

FIT2081 Week 1 - Page 2  
Introduction to Kotlin & Android

FIT2081 Week 2 - Page 59  
Kotlin Language Basics

FIT2081 Week 3 - Page 187  
Android Project Structure & UI Layouts

FIT2081 Week 4 - Page 266  
Kotlin OOP & UI Components

FIT2081 Week 5 - Page 332  
Navigation & Intents

FIT2081 Week 6 - Page 401  
Data Handling & Persistence

FIT2081 Week 7 - Page 440  
Networking & APIs

FIT2081 Week 8 - Page 536  
Coroutines & Async Programming

FIT2081 Week 9 - Page 614  
App Architecture (MVVM)

FIT2081 Week 10 - Page 695  
Advanced Compose UI & Theming

# FIT2081 Week 1

## Introduction to Kotlin & Android

# Lecture: Unit Administration & Native vs Mobile Apps

---

## Welcome!

Welcome to Mobile App Development for Semester 1 2025.

For this unit, we will use the following tools:

1. **Ed Lessons** will be used to supply all online unit learning resources,
2. **Moodle**: for online assignment submission, result release and
3. **Ed Forums**: this will be our discussion forum.

**i** Staff and tutors will attempt to respond to your forum postings within 2 business days during the week.  
Forum postings will typically not be monitored during the week-end or on holidays.

This unit will require you to participate in three different learning activities each week:

1. Lectures (1 hour)
2. Applied Labs (3 hours)
3. Peer-engagement Reflections (weekly, own-time)

See you soon,

Dr. Delvin Varghese

Chief Examiner

# Unit Administration

## Teaching Team

- Delvin Varghese (Chief Examiner & Lecturer)
- Joe Liu (Lecturer)
- Clair Pan (Admin Tutor)
- Nawfal Ali (Admin Tutor)
- Mansi Vyas (Lecture Support Tutor)

## Getting Support

Use Ed-discussion to post your questions and concerns (quick response)

- Make your post private if you want it to the teaching team only

## Unit Guide Synopsis

“This unit introduces an industrial strength programming language (with supporting software technologies and standards) and object-oriented application development in the context of mobile application development for smartphones and tablets. The approach is strictly application driven. Students will learn the syntax and semantics of the chosen language, its supporting technologies and standards, and object-oriented design and coding techniques by analysing a sequence of carefully graded, finished applications. Students will also design and build their applications.”

## Proposed Schedule

# Why Native Android Mobile Apps?

## Developing Mobile Apps

- There are 3 Fundamentally Different Types of Mobile Apps
  - They require different development skills and platforms
  - They have different capabilities, advantages and disadvantages
    - These are constantly changing as development in this extremely dynamic space continues at a pace that can easily overwhelm IT professionals
  - The Three Types are
    - Native Apps
    - Mobile Web Apps
    - Hybrid Apps

## Terms

- SDK = Software Development Kit
  - A bundle of all the software components necessary to develop and deploy on a given development platform (usually does not include an IDE)
  - We will use:
    - The Java Software Development Kit (JDK)
    - The Android SDK
      - Which uses the JDK to compile Java classes (step 1 in a 2-step compilation)
- Class Library aka Application Programming Interface (API)
  - Code that we do not write but can call (execute) to perform common but complicated tasks
    - As we will see later it's mainly in the form of classes which we can use to instantiate objects then invoke class methods on these objects to accomplish these common but complicated tasks
    - The public methods of these classes form the API of the library
  - These libraries are generally huge and perform all manner of tasks from creating a responsive UI to interacting with hardware like GPS and accelerometers etc.
- Integrated Development Environment (IDE)
  - A software environment that contains and/or orchestrates all the tools developers need to develop applications
  - e.g. very smart, language sensitive code editors, interface designers, debuggers, device telemetry monitors, device emulators, version control, etc.

## Native Apps

- Characteristics
  - App's compiled code runs directly on a device's platform
    - e.g. Android, iOS
  - Built using the SDK tools and languages provided and recommended by the platform vendor
    - e.g. Using Java (or more rarely c/c++) with Android Studio sitting on the Android SDK sitting on the JDK
      - This stack has been deployed for Windows, Mac OSX and Linux
    - e.g. Objective C/Cocoa or Swift with the XCode IDE
      - This stack only runs on Mac OSX

## Mobile Web Apps

- Characteristics
  - A Web site (built using any Web site development technology) designed for smart device displays and accessed (like any Web site) by a device's browser

## Mobile Web Apps

- Ongoing Developments
  - The Web application development space is ridiculously dynamic
    - Every week it seems some new, game-changing HTML5/CSS3 feature or JS library is released
    - These all change what is possible with a Mobile Web App
  - Some Examples
    - HTML5/CSS3 advances make it possible for a Mobile Web App to look and feel like a Native App
    - Improvement in JS execution speeds make it possible to approach the performance of a Native App
    - With a few meta elements in a Web page's header the page can be rendered full page without any browser décor and an icon placed on the devices home page that will launch the app in a way indistinguishable from a Native app
    - Browser Local storage and application caching make off-line operation possible
    - Browser JS → Native API bridges are improving fast allowing, for instance, a Mobile Web App to access a device's hardware
    - The latest technology is Service Workers which will allow push notification to a Mobile Web App - the holy grail of App engagement once thought only possible in a Native App
    - Chrome remote debugging tools allow debugging of client side JS code on the device once only the domain of native apps

# Hybrid Apps

- Characteristics
  - Written using a language and development environment other than the recommended languages for the platform but deployed as a Native App
  - Two main types:
    - A thin native app shell contains a Web app
      - The shell contains a JS → Native API bridge
      - The shell contains a native component that interacts with the platform's Web rendering engine to maintain a UI
      - e.g. Cordova/PhoneGap, [Trigger.io](#), Ionic, and Sencha
    - A cross compiler is used to convert code into a Native App executable for each required target platform (is this a Native app)
      - e.g. Xamarin, Appcelerator, Embarcadero FireMonkey, or RubyMotion
      - It's possible for developers to be able to call through transparently from their own language to the underlying platform's API giving complete native access
      - What about UI designers and other development tools?
      - Single code base: Yes! Simplicity of a Web App: No!
- More
  - It's actually possible (although not easy) to build a Hybrid Web App without any 3rd party support
  - But usually a 3rd party such as Xamarin and Ionic provide a complete IDE that:
    - Critically uses a single code base to deploy to multiple platforms
    - Creates and constantly updates the JS → Native API bridge in the thin native client it deploys on each platform
      - The quality, completeness and cross browser consistency of such bridges is critical to Hybrid Apps

## Some Important Issues

- Companies Absolutely Require a Mobile Presence
  - This usually means a presence on both major mobile platforms
- Native Apps
  - Requires scarce, relatively technically sophisticated, costly developers, one set per target platform
  - Developers have complete access to the device's features
- Mobile Web Apps
  - Requires abundant, relatively less sophisticated, cheaper Web developers,

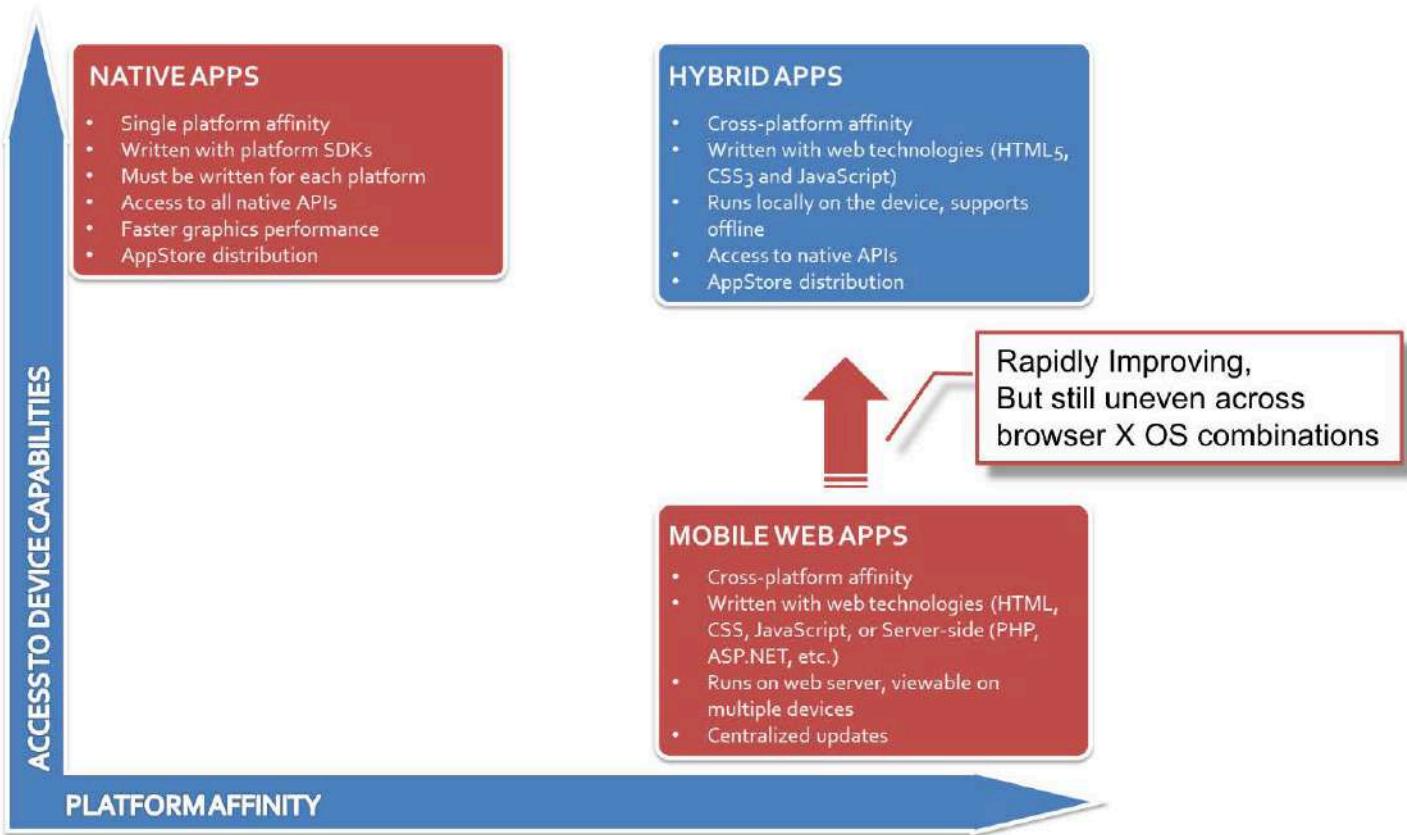
- The same “App” functions on all platforms (write once)\*
- Developer access to device features and the Native API in general depends on the quality, completeness and cross browser consistency of the browser’s JS → Native API bridge
- Hybrid Web Apps (the enclosed Web app type)
  - Requires abundant, relatively less sophisticated, cheap Web developers,
  - The same “App” functions on all platforms (write once) that the hybrid app IDE creates thin clients for \*
  - Developer access to device features and the Native API in general depends on the quality, completeness and cross browser consistency of a 3rd party’s JS → Native API bridge

## More Issues

- Cost
  - Write once is cheaper and Web developer skills are cheaper
  - Write once can be over-hyped though
    - Although WebKit (rendering engine) is virtually a standard on Mobile browsers their JS engines wrt JS → Native API are not consistent
    - Hybrid Apps are also at the mercy of the quality completeness and consistency of their IDE’s thin native clients for the various target platforms
  - UI Look and Feel (L&F)
    - Web L&F is not Android/iOS L&F
    - HTML5 allows development of a sophisticated UI (swiping etc.) but it’s often still a different user experience
  - Offline
    - In-browser data storage is limited compared to local file storage, does your App require offline functionality?
  - Discoverability and Installation
    - App store vs URL and bookmarking (do mobile users bookmark?)
    - Installation versus no installation required
  - Speed
    - Native will always shine here and for fluid, complex graphics it’s essential
    - Many important types of App do not require this kind of speed
  - Security
    - Mobile Web Apps are subject to all the security risks of normal Web Apps
    - Native Apps have none of these risks
  - Content restrictions, approval process, fees
    - Anything in an App store (Native and Hybrid) usually shares its purchase price with the store owner and must undergo a lengthy approval process (in the case of Apple’s App Store)
    - The Web is free of any of these encumbrances
  - Maintenance

- Multiple platforms to update consistently if not write once, update related delays and re-approval if in an app store (especially Apple's)
- Web updates are instantaneous to all platforms

## Characteristics / Pros / Cons



## Summary

- Surprisingly (to me at least) all 3 types seem to be thriving equally in their own niches
  - btw Native apps can include UI widgets that are essentially embedded browser windows which opens up many possibilities
    - Native and Mobile Web Apps mash-ups
    - e.g. circumventing Apple's approval process
  - Something we have not considered is that a poorly designed app is bad app no matter how its implemented
    - And the design requirements for a Mobile application are very different to those for a desktop application

# Support @ Monash

## Faculty of IT website & Student Portal

The faculty of Information Technology has developed the following resources to ensure you have a successful start to your learning journey. It is recommended to become familiar with this following:

- [Faculty of IT website](#)
- [Student portal](#)

## Ask Monash

[Ask Monash](#) is a self-service tool for you to search for information 24/7, submit your own questions for assistance and review any responses provided.

## Zoom User Guides

- [Getting started with Zoom](#)
- [Zoom induction slides](#)

## Support Services

- **Academic Support:** Book in with a learning adviser for one-on-one consultations to help you to prepare for your assessments, academic English, time management skills and academic integrity. Expert learning advisers in the [Student Academic Success \(SAS\)](#) team can help you develop learning and academic English skills to be successful in your studies. Browse [Learn HQ](#) for self serve learning and academic English resources at any time.
- **Counselling Support:** Keeping on top of our studies can be stressful and to add to that work, looking after family and all the other aspects of life so often invisible to those around us, it can be really tough. If you find that stress and health and wellbeing concerns are affecting your studies or broader life, [reach out to our counselling team](#). They are trained professionals (mostly psychologists and social workers) who can help equip us with strategies to manage the ups and downs of life, both small and large. Mental health problems are much like physical ones, and a professional is well placed to support us with an objective perspective and help us work through problems. Remember, reaching out for help is a sign of strength!
- **English Skills:** Want to gain more confidence in your English speaking in social contexts? [English Connect](#) can help you to improve your speaking and be more confident and prepared for the future.
- **Disability Support:** Some of our students experience barriers to their academic success on the basis of disability status, physical or mental health conditions, carer status, or other factors outside their control. While [extensions and special consideration](#) requests can assist with short-term issues, others may be ongoing and you are encouraged to speak with the [Disability Support Services](#) team who can assist you with adjustments (such as flexibility with assignment

deadlines, alternative exam arrangements, assistive software, note-takers, etc.) that allow equitable access to learning as well as individualised career support through the [GradWISE program](#).

# Code of Conduct and Ed Forums

We will use the Ed forum for announcements, discussions, questions, etc.

- Be **courteous**. Don't write anything you would not say face-to-face or that you would not want family or a future employer to read. Don't divulge private details about yourself or others.
- Be **helpful**, within limits. You can help each other but do not give answers to assessed assignment questions face-to-face, and the communications on Ed are subject to the usual **academic integrity rules**. Your lecturer will advise on how to safely collaborate without breaching academic integrity
- Be **generous**. We encourage you "give" at least as much as you "take", to the extent that it is possible and follows the previous rule. Ask away, but (attempt to) answer questions asked by others if you can. The teaching staff is here to help, but their availability is limited. Having the cohort's help will make a day-and-night difference in everyone's experience.
- Be **clear**. Review what you write to make sure others will understand your message and intent correctly and without effort. When we speak face to face and are misunderstood, we have an on-the-spot opportunity to rephrase our words. In writing, we must strive twice as hard to be understood, as we do not have the benefit of modifying or elaborating in real time. Finally, a clear question will get better answers faster.
- Be **self-sufficient**. Before asking questions, do your own research to ensure that the answers to your questions are not just a few clicks away from you. If you find answers, but you're unsure about them, or If you think others have the same questions, you can post your questions with the answers you have found and ask for a second opinion.
- Monash takes its **student charter** seriously. Any posts deemed inappropriate by your teaching team will be removed and treated as student **general misconduct**.

# Lecture 1 - Notes

## FIT2081 - Mobile App Development

### Week 1: Introduction to Kotlin & Android

#### 1. Welcome and Introduction

- **Lecturer:** Delvin Varghese (Chief Examiner and Unit Coordinator), Joe Liu
- **Teaching Team:**
  - Norman Chen (Lecture support and Q&A moderator)
  - Clair Pan (Admin tutor)
  - Multiple tutors for practical workshops

#### Unit Overview:

- **Unit Size:** 1,100 students at Clayton, 400 students in Malaysia
- **Unit Type:** Ultra-large unit with many tutors
- **Learning Approach:** Highly practical with a strong focus on building real-world applications

#### 2. What is FIT2081?

- **Primary Goal:** Equip students with skills to develop real-world mobile applications.
- **Core Language:** Kotlin (transitioning from Java due to its modern features)
- **Primary Framework:** Jetpack Compose for UI development
- **Learning Philosophy:** Application-driven, hands-on programming experience
- **Key Takeaway:** Android development is more than just learning Kotlin; it's about understanding mobile app architecture, ecosystem, and UI/UX principles.

#### 3. Live Poll: Student Background in Programming

**Question:** What programming languages have you used before?

- **Common Responses:** Python, Java, JavaScript, C++, R, C#, Unity, and some Kotlin.
- **Observation:** Majority have prior experience with Python and Java, but very few have used Kotlin before.

#### 4. Why Kotlin for Android Development?

- **Google's Official Language for Android:**
  - Better null safety handling (avoids NullPointerException)
  - Concise syntax compared to Java
  - Coroutines for simplified asynchronous programming

- Interoperable with existing Java codebases
- **Why Jetpack Compose for UI?**
  - Declarative UI approach (similar to React)
  - Simplifies UI development with @Composable functions
  - No need for XML-based layouts
  - More efficient UI rendering

## 5. Course Structure & Weekly Breakdown

- **Week 1:** Introduction to Kotlin & Android
- **Week 2:** Kotlin Language Basics
- **Week 3:** Android Project Structure & UI Layouts
- **Week 4:** Kotlin OOP & UI Components
- **Week 5:** Navigation & Intents
- **Week 6:** Data Handling & Persistence
- **Week 7:** Networking & APIs
- **Week 8:** Coroutines & Async Programming
- **Week 9:** MVVM Architecture
- **Week 10:** Advanced Compose UI & Theming
- **Week 11:** Deployment & Final Project Showcase
- **Week 12:** Cross-Platform App Development & Quiz

## 6. Course Logistics and Assessments

- **Assessment Components:**
  - **Assignment 1:** Basic App Development (20%)
  - **Assignment 2:** Advanced App Development (30%)
  - **In-Semester Quiz:** Theory-based (20%)
  - **Peer Engagement Activities:** Participation-based (10%)
  - **App Critique Report:** Evaluating an existing app (10%)
- **Submission Policy:** Extensions permitted except for quizzes and peer engagement.
- **Use of AI:** Generative AI tools allowed but must be declared.

## 7. Practical Workshop Setup

### Live Demo: Setting up Android Studio

- **Step 1:** Install & configure Android Studio
- **Step 2:** Create a simple "Hello World" Android app
- **Step 3:** Explore project structure, Gradle dependencies, and emulator setup
- **Step 4:** Introduction to @Composable functions and preview tools

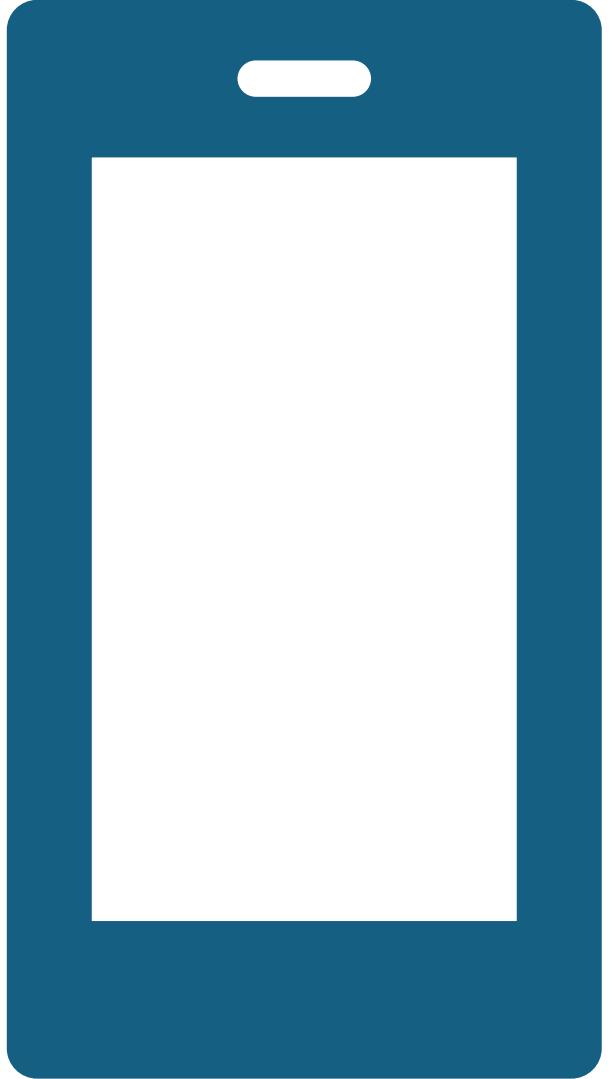
## 8. Resources and Support

- **Primary Learning Platform:** Ed Discussion Forum
  - Post all questions here instead of emailing

- Public and private posts available
- **Moodle:** Used mainly for submissions
- **Consultation Sessions:** Available from Week 2
- **Tutors:** 20+ tutors available for help in labs and Ed discussion

## 9. Closing and Next Steps

- **Next Lecture:** Deep dive into Kotlin syntax and debugging techniques
- **Action Items:**
  - Install Android Studio and set up your environment
  - Familiarize yourself with the course materials on Ed
  - Participate in the Ed discussion and introduce yourself
- **Q&A Session:** Open floor for student questions and clarifications



# Preparing for Life as an Android Developer

Delvin Varghese

# The Road Ahead

- Our Unit Structure: You'll have a 1-hour online lecture each week, occasionally supported by a short recorded session, plus 3-hour labs for hands-on practice. This format helps balance theory with real-world coding tasks
- Action-Packed Learning: We'll focus on practical demos to keep you engaged and help you immediately apply new concepts. Android is massive, so our priority is to build a solid foundation rather than cover everything.
- Expectations: This journey demands active participation. You'll be required to examine, modify, and fully understand each provided code snippet, as any part could appear in quizzes or final assessments.

# Slide Sets & Code Notes

---

- Purpose of Slides: Our slides summarize key Android/Kotlin concepts in a concise format. They won't replace deeper reading or coding practice, but offer quick references when you get stuck.
- Coupled with Code Demos: You'll see actual app snippets and Kotlin techniques that bridge theory and practice. Pay close attention to these demos to see how abstract concepts translate into working code.
- Brief Yet Impactful: The slide sets are intentionally short so you can spend more time coding. Detailed explanations appear in code comments and references, ensuring you learn by doing.

# Android – A Vast Ecosystem

---

- Why Only the Fundamentals?: We'll focus on core activities, views, and Compose fundamentals to get you deployment-ready without overwhelming you.
- Real-World Influence: Android apps power billions of devices globally, making every skill here transferrable to real industry projects.
- Long-Term Learning: Mastering the essentials lays the groundwork for diving into advanced topics like architecture components, coroutines, and more after this unit.

# Kotlin & Compose Advantages

- Modern & Concise: Kotlin's null-safety and brevity cut down on crashes and boilerplate. Compose uses Kotlin to define UIs declaratively, reducing complexity.
- Replaces XML Layouts: Compose eliminates clunky XML for layout creation. Everything happens in Kotlin, creating a more unified development experience.
- Strong Community: Google's official support plus an active community means updates, libraries, and forums are widely available.

# Pre-Reading & Background Materials

- Developer Docs: The official Android site ([developer.android.com](https://developer.android.com)) is huge. Sometimes straightforward, other times dense—but a crucial resource for your career.
- Supplemental Guides: We'll point you to blog posts and tutorials, but they build on the fundamentals from lectures and labs.
- Persistence is Key: Official documentation might seem overwhelming, but repeated exposure and practice will strengthen your resilience as a developer.

# Labs – A Deep Dive into Code

---

- Hands-On Practice: Each 3-hour lab features code to debug, expand, or repurpose. This is where you'll truly hone your Kotlin and Android skills.
- Expect to Tinker: Learning by experimentation is key. Don't be afraid to break things and ask plenty of questions.
- Also our assessments are super-practical!

# Exploring Official Documentation

---

- Life in the Workplace: Documentation mastery is a universal skill. Whether exploring new frameworks or corporate policies, the pattern is the same: read, interpret, and experiment.
- No Panic Zone: At first, it can feel daunting. Learn to skim for keywords, try simple demos, and piece solutions together gradually.
- Teach Yourself to Fish: We won't translate everything for you. Our aim is to build your ability to handle tough documentation confidently.

# Getting Help – Stack Overflow & Beyond



Stack Overflow: A go-to Q&A platform for Android and Kotlin. Most everyday development questions have been answered in detail there.



Peer & Tutor Support: Don't forget your course instructors and classmates; explaining your problem clarifies your own understanding.



Don't Suffer in Silence: Investigate on your own first, but reach out when you hit a wall. Collaboration is integral to professional development.



GenAI tools i.e. Google Gemini or ChatGPT can also be very useful for debugging! (but you don't need me to tell you this ;-))

# Documentation as a Lifelong Skill

---

- Parallel Industries: Whether it's an obscure cloud service or complex governance policy, your method—breaking things down, testing, iterating—remains the same.
- Persistence & Pattern Recognition: Over time, you'll identify structural patterns in docs, making it easier to find the sections you need.
- Professionalism: Employers love developers who can self-educate. Demonstrating quick adaptation and learning is a major career booster.

# Embrace the Challenge

- Adapt & Evolve: Mobile development changes constantly. Embrace continuous learning to stay relevant as new frameworks or APIs emerge.
- Ownership of Your Skills: We won't spoon-feed every answer. Testing and refining your understanding is an essential part of growing as a developer.
- Looking Forward: In the coming weeks, we'll cover Kotlin and Android essentials. Stay curious, use all available resources, and remember—you're learning to fish, not just receiving fish.

# Lab: Installing Android Studio | Hello World

---

## Overview

### Learning Objectives

- Set up the Android development environment
- Create and run a basic Android application
- Understand the Android project structure
- Implement simple user interface elements and event handling
- Practice basic Kotlin syntax in an Android context

### Prerequisites

- Basic programming experience (Java or Python)
- A computer capable of running Android Studio (8GB RAM minimum recommended)

---

# 1. How to install Android Studio

## Installing Android Studio

Follow the official installation guide: <https://developer.android.com/studio/install>

### **Key installation tips:**

- Select "Standard" installation to get recommended components
- Allow sufficient disk space (at least 8GB)
- If you encounter issues, consult the troubleshooting guide in the resources section

**Verification task:** After installation, launch Android Studio and confirm you can see the welcome screen.

# Update: For those with slow laptops..

If you are running Android Studio on a laptop with limited RAM, a slower processor, or lower storage speed, you may experience performance slowdowns. This is especially true when using the Android Emulator, which requires its own dedicated RAM and CPU resources to function efficiently.

## What Can You Do?

To improve your experience:

Close any unnecessary applications while using Android Studio.

Reduce the allocated RAM for the emulator in AVD Manager.

Use a lower-resolution emulator or switch to a physical Android device.

## Debugging on a Physical Android Device

One way to avoid emulator lag is to **debug directly on a real Android device** using a USB data cable. To do this:

1. **Enable Developer Mode** on your phone by tapping "Build Number" 7 times in **Settings > About Phone**.
2. **Enable USB Debugging** in **Developer Options**.
3. **Connect your device** via USB and select "Allow" when prompted.
4. Run your app directly on the phone from Android Studio.

For more information:

[https://documentation.xojo.com/topics/debugging/android/android\\_debugging\\_on\\_device.html](https://documentation.xojo.com/topics/debugging/android/android_debugging_on_device.html)

<https://developer.android.com/studio/run/device>

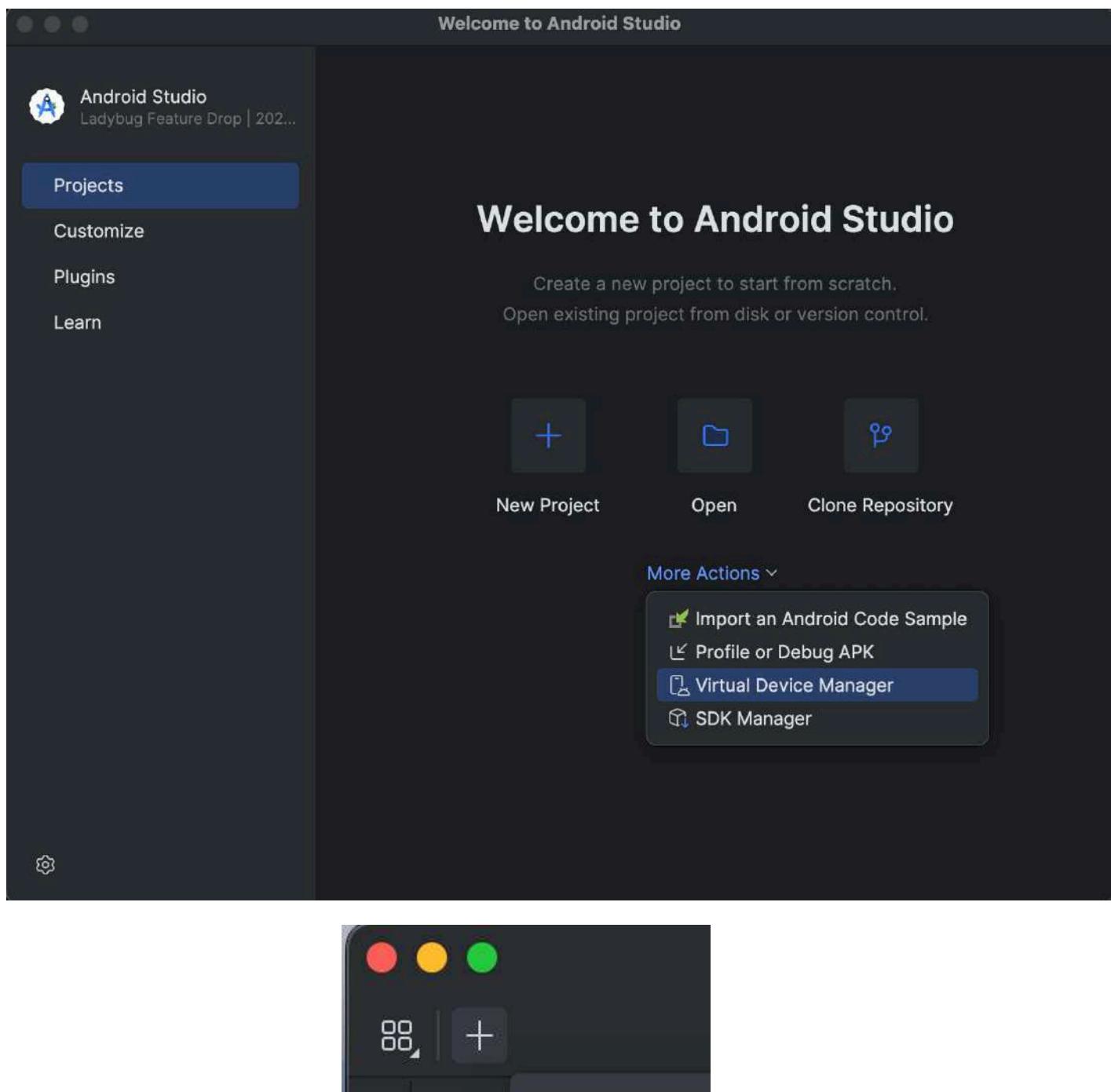
## 2. Create an Emulator

**Why this matters:** Emulators allow you to test your applications without a physical device, simulating various screen sizes and Android versions.

==

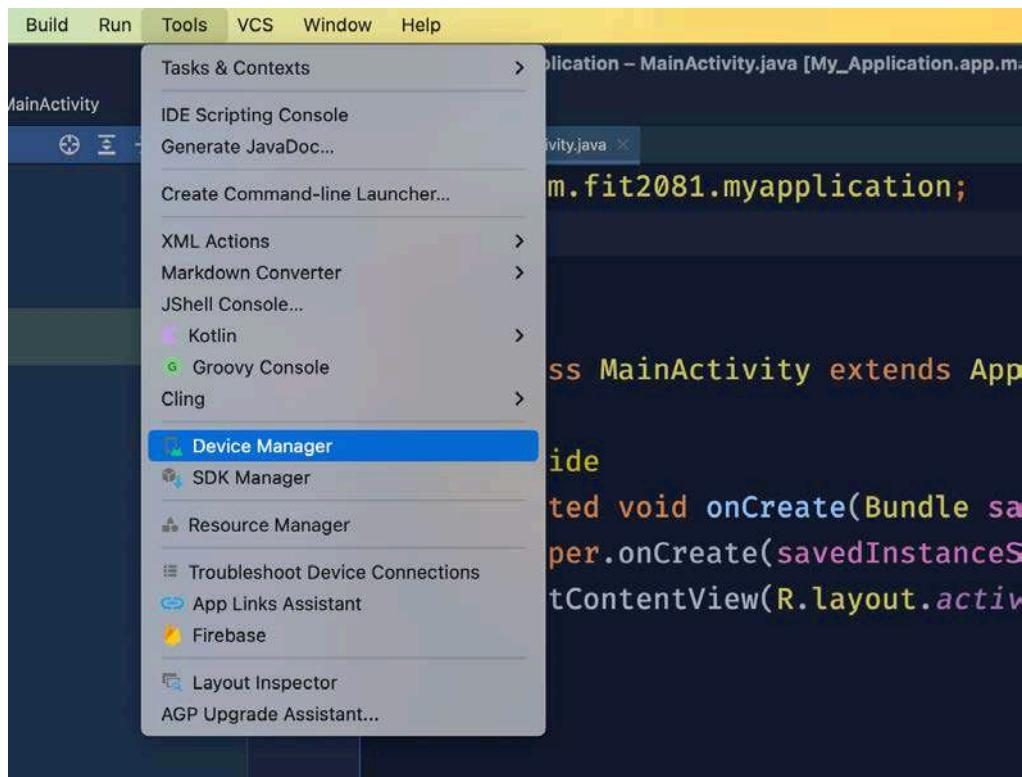
Ok, let's begin!

- Open your Android Studio
- More actions > Virtual Device Manager and click on the "+" icon in the top left corner to add a new device.

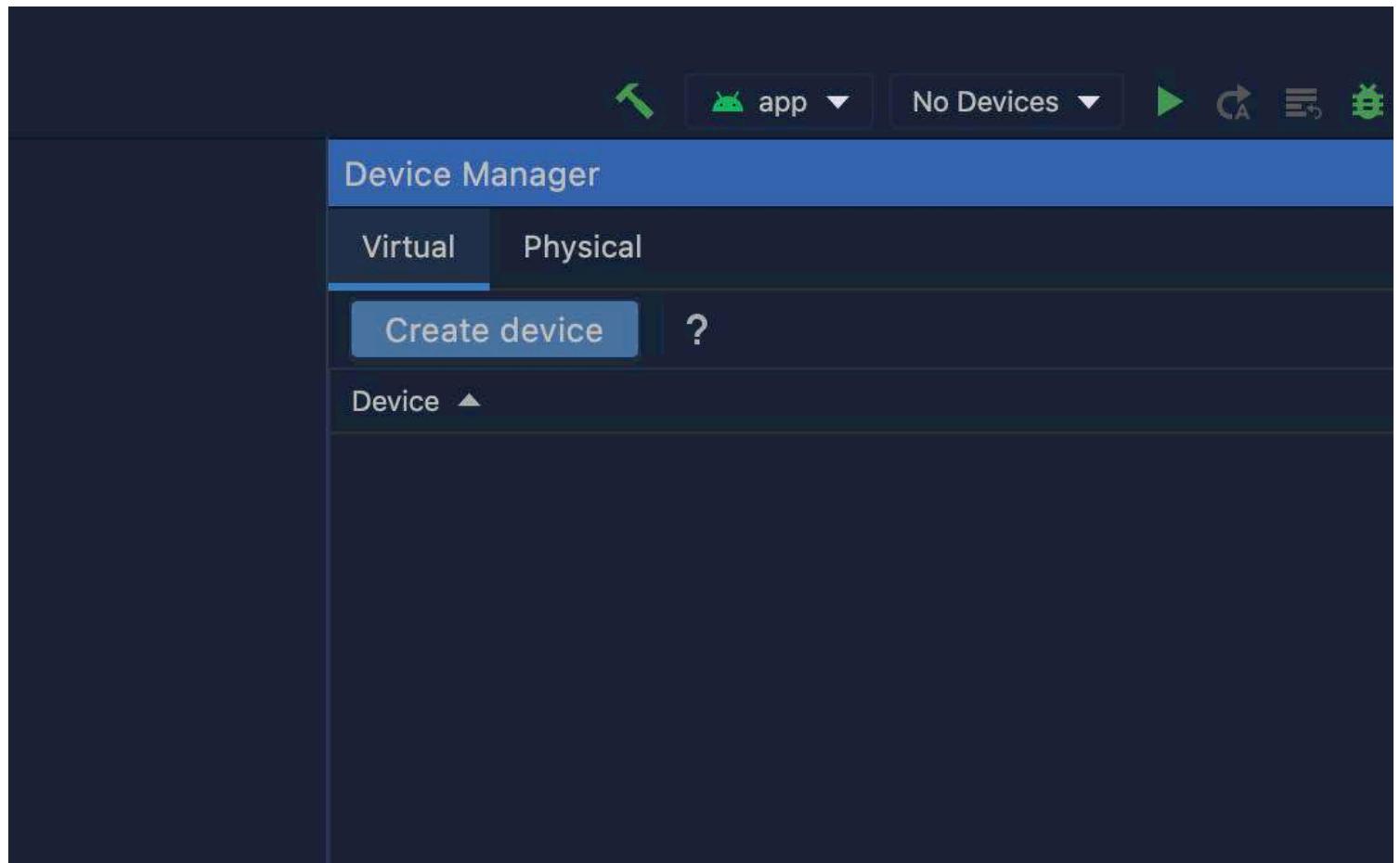


If you have already created a new project, the way to access the "Android Virtual Device" is below.

- From Tools--> Open Device Manager



- From the device manager panel, click on Create Device



- Select the hardware and click Next. Ensure the Play Store is available on the selected device.

Virtual Device Configuration

## Select Hardware

Choose a device definition

Category	Name	Play Store	Size	Resolution	Density
Phone	Small Phone	▶	4.65"	720x1280	xhdpi
Tablet	Medium Phone	▶	6.4"	1080x2400	420dpi
Wear OS	Pixel 9 Pro XL	▶	6.8"	1344x2992	xxhdpi
Desktop	Pixel 9 Pro Fold	▶	8.0"	2076x2152	390dpi
TV	Pixel 9 Pro	▶	6.3"	1280x2856	xxhdpi
Automotive	Pixel 9	▶	6.24"	1080x2424	420dpi
Legacy	Pixel 8a	▶	6.1"	1080x2400	420dpi
	Pixel 8 Pro	▶	6.7"	1344x2992	xxhdpi
	Pixel 8	▶	6.17"	1080x2400	420dpi
	Pixel Fold	▶	7.58"	1840x2208	420dpi

New Hardware Profile Import Hardware Profiles Refresh Clone Device...

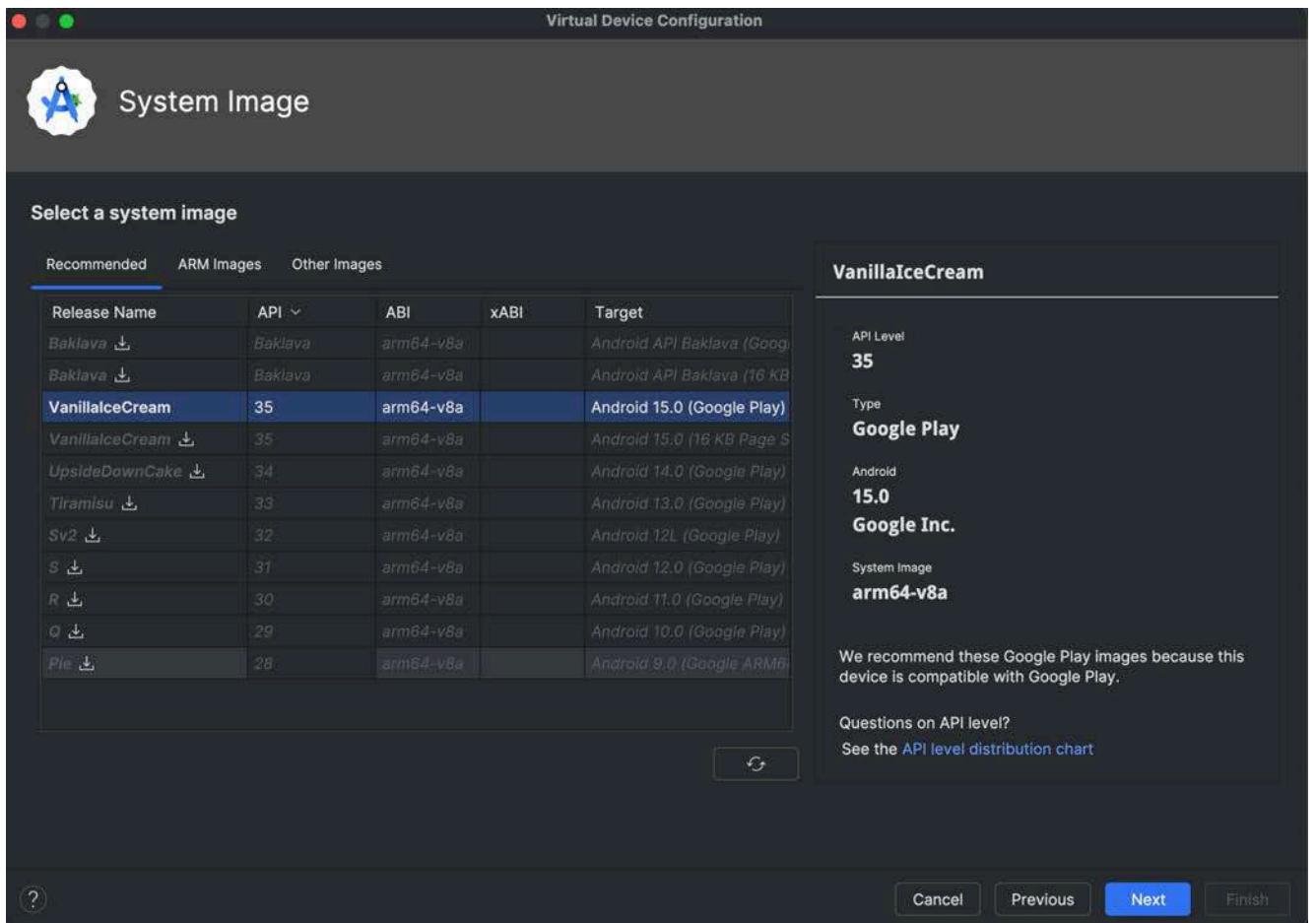
?

Cancel Previous Next Finish

**Pixel 9**

Size: large  
Ratio: long  
Density: 420dpi

- Select the system image (OS). Ensure the following:
  - API Level >= 35
  - ABI = x86\_64 (Win) or arm64\_v8a (Mac)
  - Target = Android (Google Play)



**i** If you haven't, you must download the image first by clicking on the download icon (circled in red)

- Select Next, then finish.

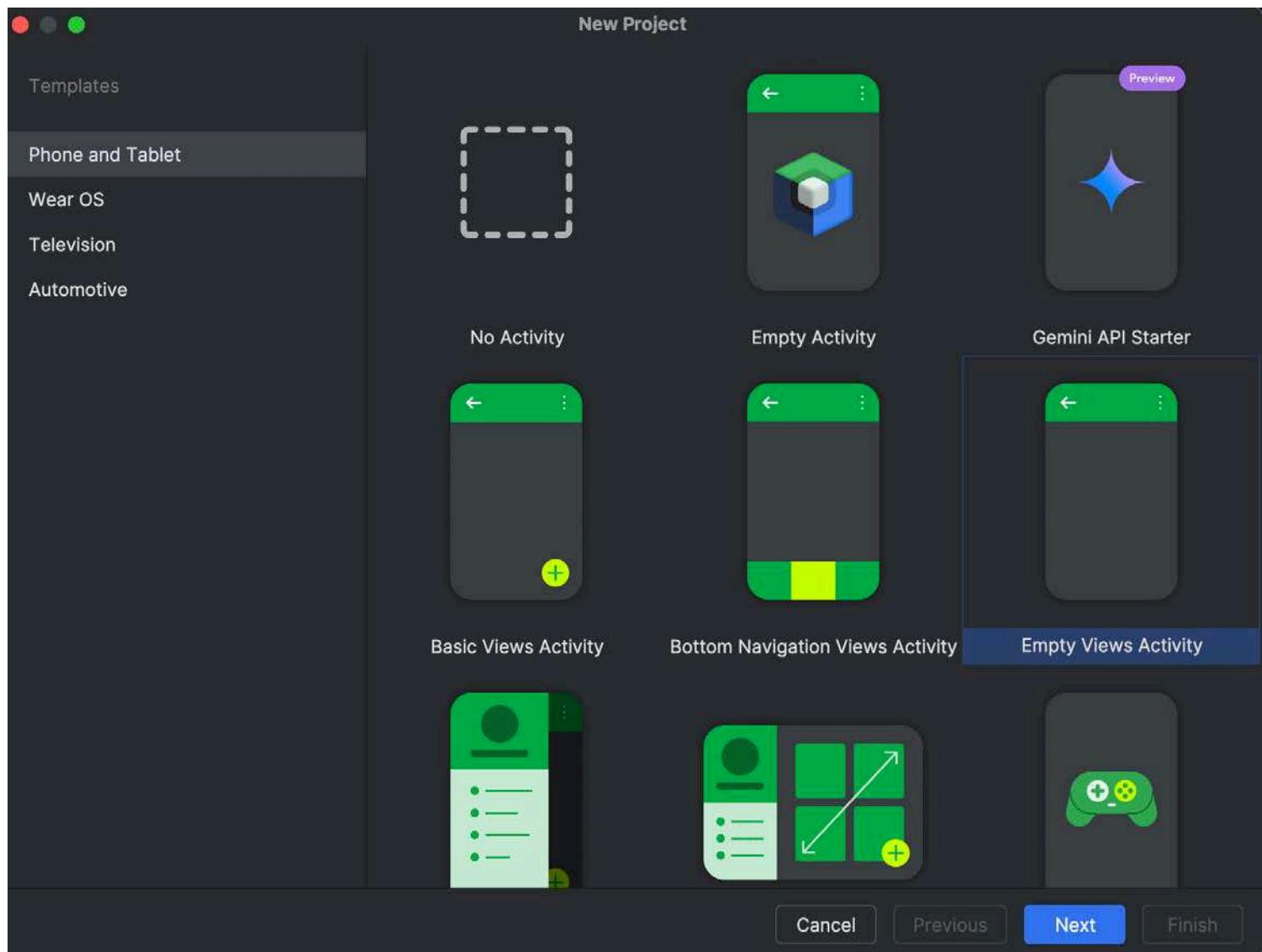
==

**Checkpoint:** Successfully create and launch your emulator. You should see the Android home screen.

### 3. Create First Project

## Create First Project

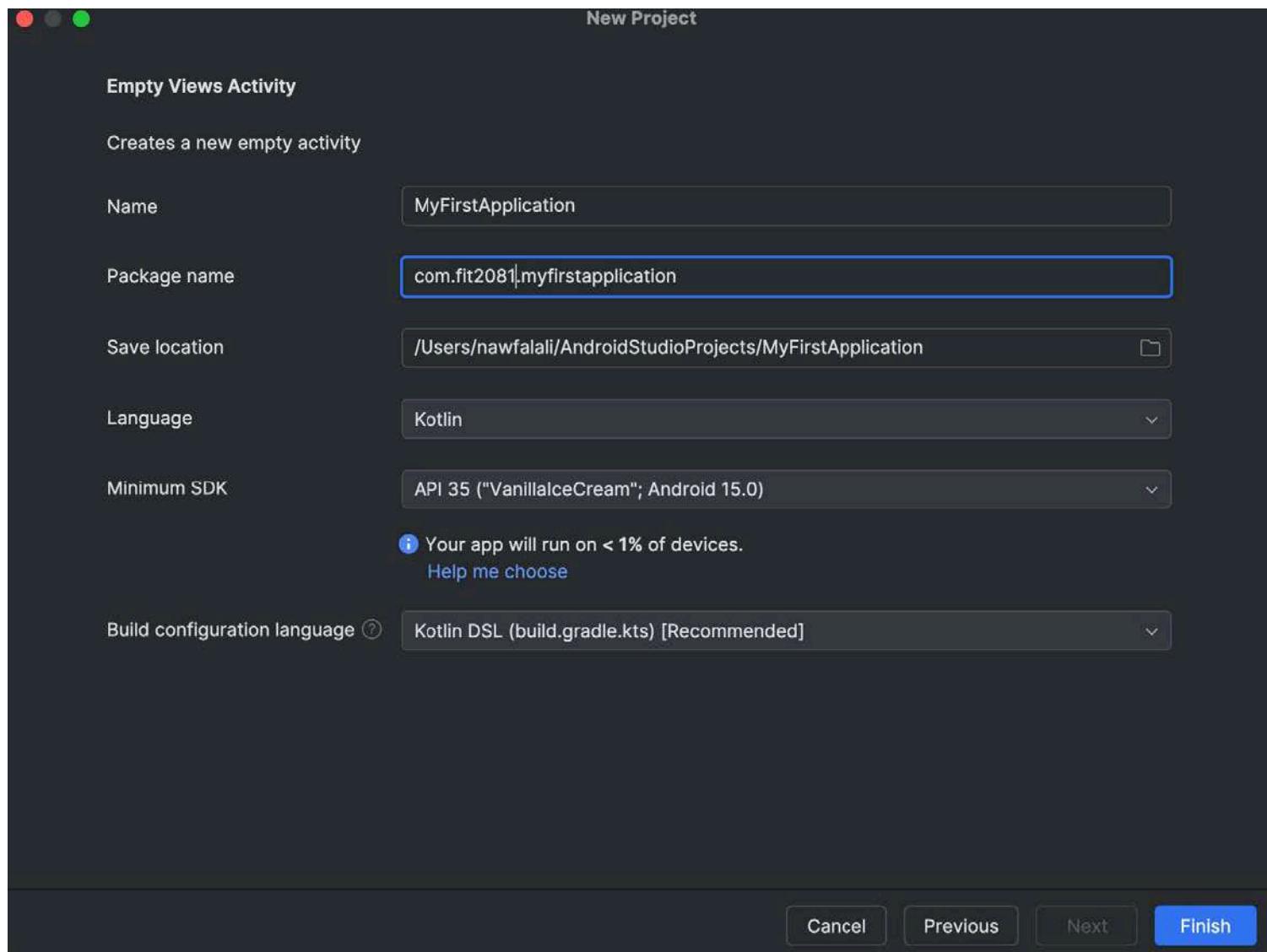
Open Android Studio and select New Project → Empty Views Activity → Next



Specify the following:

- Name of the project: for example (MyFirstApplication)
- Package name using the dot notation: com.fit2081.myfirstapplication
- Location: Folder where project files will reside.
- Language: Kotlin
- Minimum SDK: 35

Click Finish



**X** You might get a message "Gradle sync in progress" and that lasts for several minutes -- this is completely normal!

=====

## Exploring the Project Structure

Take time to explore and understand these key components:

- **app/src/main/java**: Contains your Kotlin code files
- **app/src/main/res/layout**: Contains XML layout files
- **app/src/main/res/values**: Contains resource files like strings, colors
- **app/build.gradle**: Project dependencies and build configuration

**Why this matters:** Understanding the project structure helps you locate and modify different components of your application efficiently.

=====

## Running Your Application

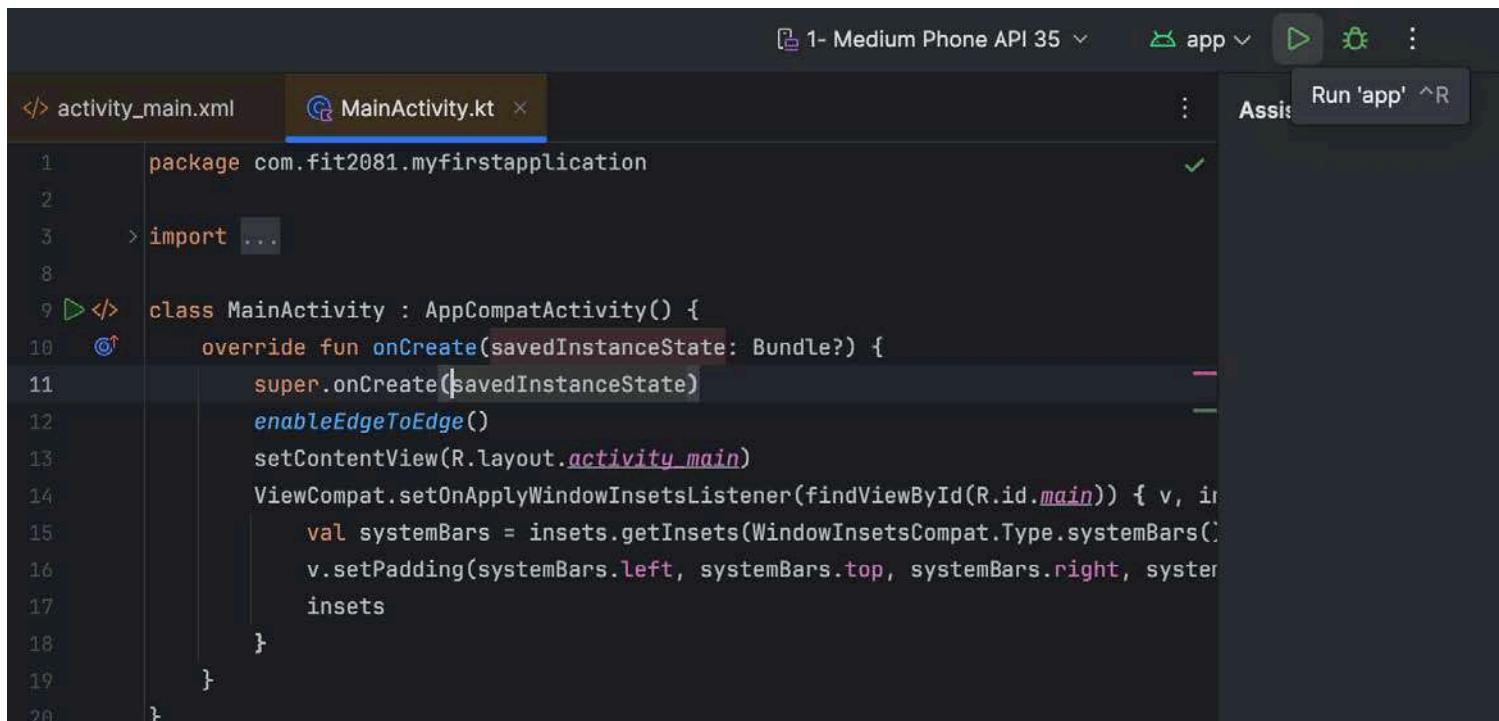
1. Click the green triangle (Run) button in the toolbar
2. Select your emulator as the deployment target
3. Wait for the app to build and launch

### Reflection questions:

- What files were automatically generated for you?
- How does the project structure differ from Java projects you've worked with?
- What is the entry point of your application?

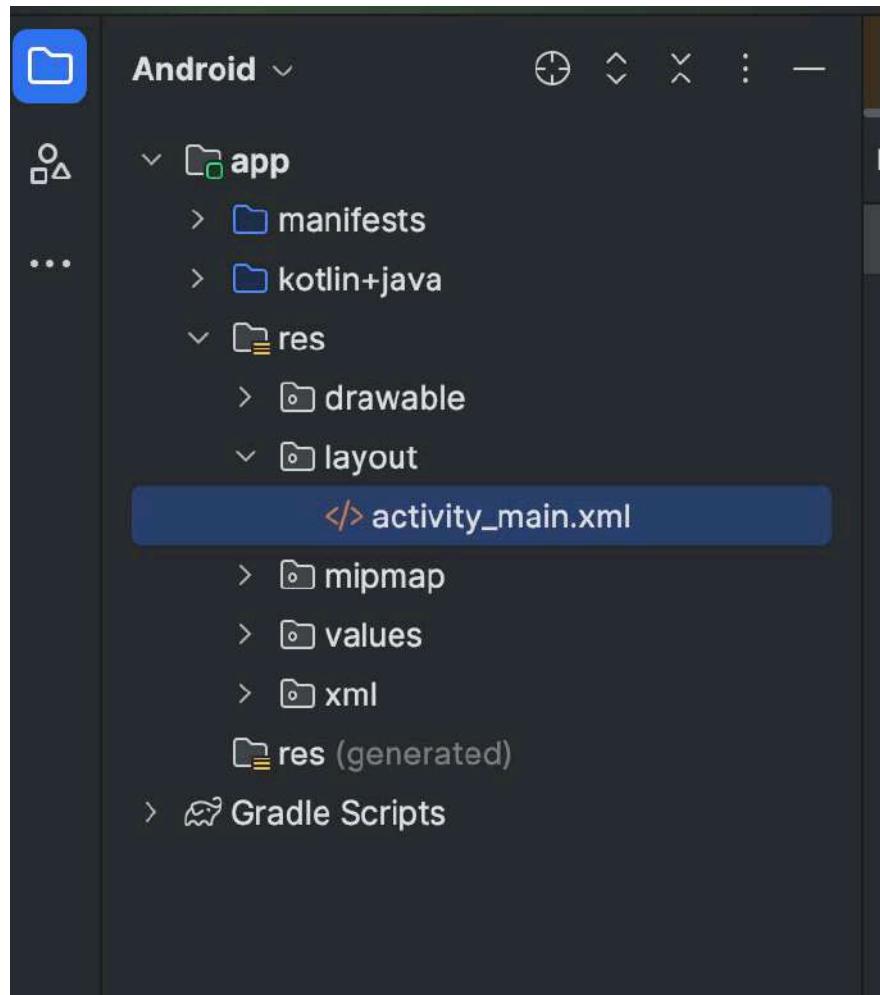
=====

To run your application and deploy it to your emulator, click the green triangle.

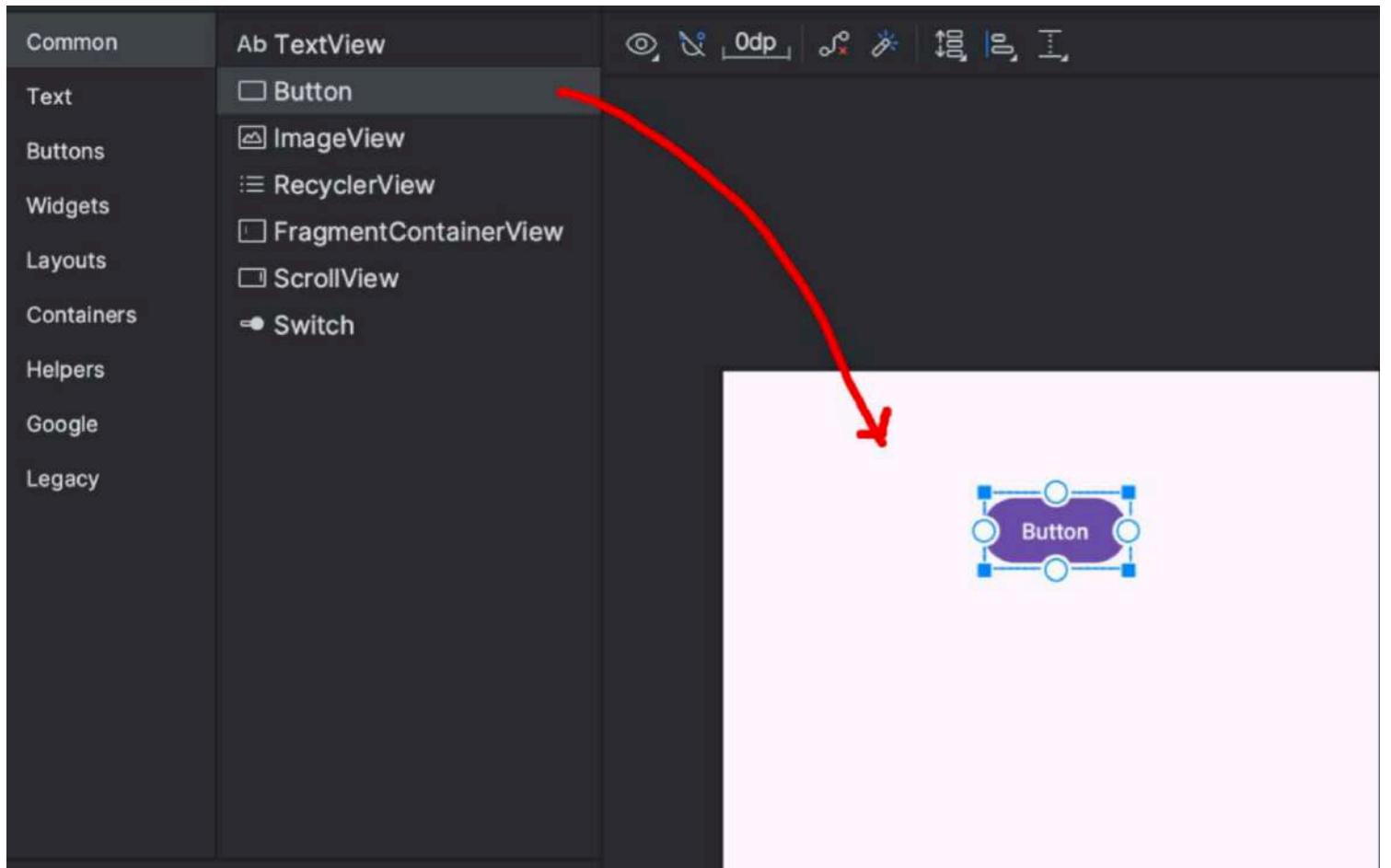


```
1 package com.fit2081.myfirstapplication
2
3 import ...
4
5 class MainActivity : AppCompatActivity() {
6     override fun onCreate(savedInstanceState: Bundle?) {
7         super.onCreate(savedInstanceState)
8         enableEdgeToEdge()
9         setContentView(R.layout.activity_main)
10        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) { v, i
11            val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())
12            v.setPadding(systemBars.left, systemBars.top, systemBars.right, system
13            insets
14        }
15    }
16 }
```

To add more user interface (UI) elements, go to the project explorer, expand the tree until you find the file activity\_main.xml



From the palette, drag UI elements and drop them on your activity (white area).



After each update, rerun your application to see the changes on your emulator.

---

## Why is My Button Jumping to the Top-Left?

If you've just added a button and it's moving unexpectedly to the top-left of the screen, here's why:

- **By default, Android Studio places elements at (0,0)** – which is the top-left of the screen.
- **Without constraints, UI elements don't "know" where to position themselves.** They need explicit placement rules.
- **We'll be covering layout constraints next** – including how to properly position buttons, text, and other UI components.

# Intermission

Well done on getting this far!

## Understanding XML Layouts vs. Compose

In today's lab, we'll explore both XML-based layouts and Jetpack Compose:

- XML layouts separate UI design (XML) from logic (Kotlin)
- Compose combines UI and logic in Kotlin

Today, we'll focus on the traditional XML approach first, just to show you the difference between the older (XML) and newer style (Compose).

But from next week's lab, we only focus on Jetpack Compose.

## 4. Add a New Button

### Instructions in brief

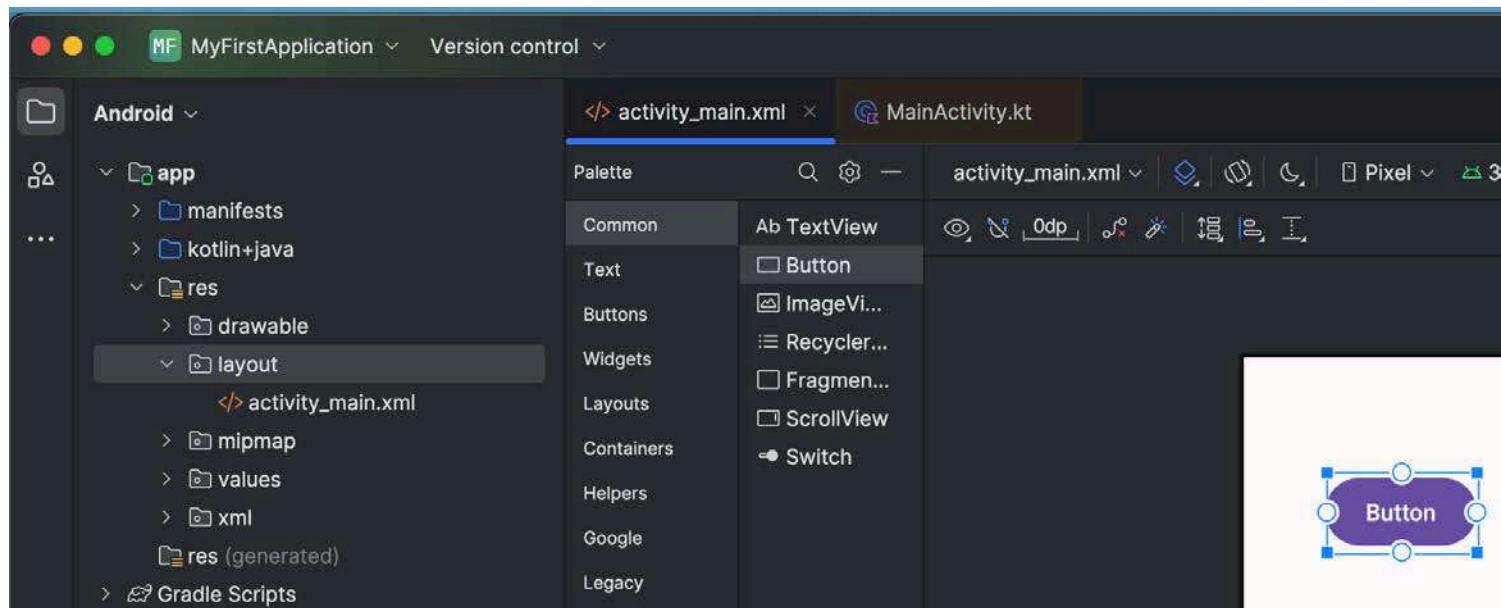
#### Adding a Button

1. Open app/src/main/res/layout/activity\_main.xml
2. Switch to Design view if not already selected
3. From the Palette, drag a Button onto the design surface
4. In the Attributes panel:
  - o Set the button's ID to "clickMeButton"
  - o Change the text to "Click Me"
6. Add constraints by:
  - o Selecting the button
  - o Dragging the constraint handles to the edges of the parent layout
  - o Alternatively, click the "Infer Constraints" icon

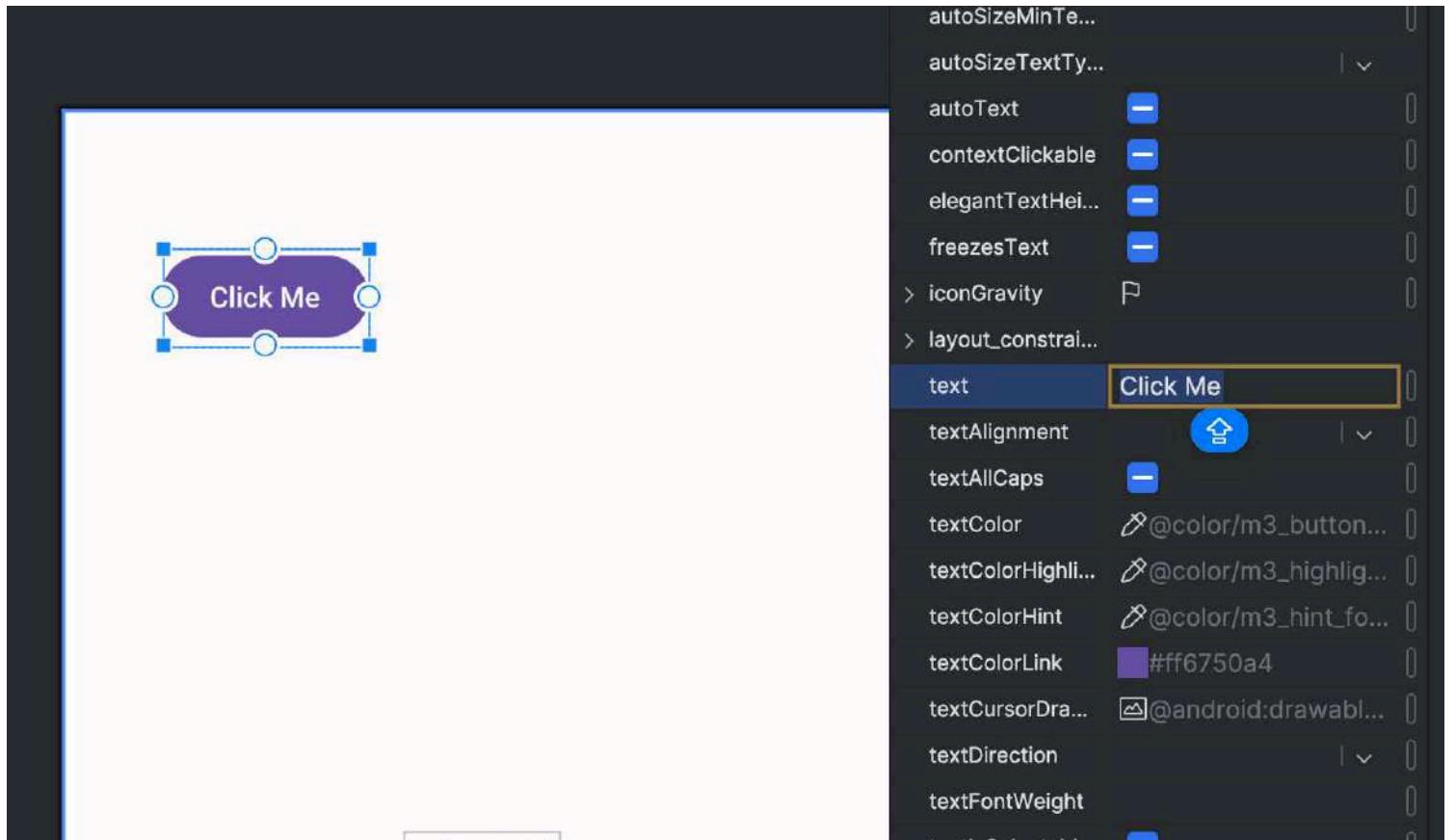
### More detail

Follow the steps below using the same app created in the previous slide.

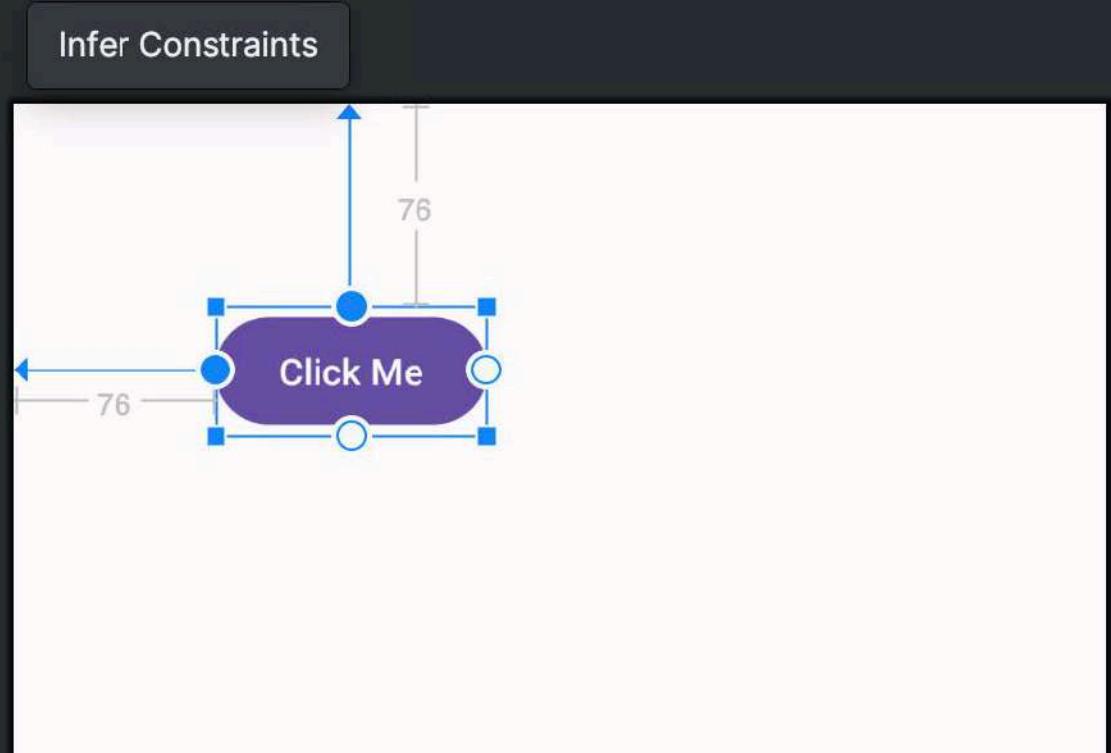
1. Drag a button to the white area representing your device screen.



2. Change the button label to "Click Me" using the Attributes window on the right. You would need to select the button to see its properties.



3. Drag the button's top and left white dots to the edge of the device screen to add constraints to the widget. These constraints tell Android how to position the UI element on the layout. You can also select the UI element, click the Infer Constraints icon, and let Android Studio apply the appropriate constraints:



4. Once you run your app, you should see a newly added button on the screen of your virtual device. Great job!

=====

**Understanding constraints:** Constraints determine how UI elements are positioned relative to each other and the parent container. They ensure your UI adapts to different screen sizes.

# 5. Implementing Event Handling

**Why this matters:** Event handling is fundamental to interactive applications. Understanding how to respond to user actions is essential for mobile development.

Since you've already done a foundational programming unit, event handling should already be familiar to you. We will build on that understanding to show you how event handling works in Android.

## Creating an event handler method

After completing the previous step, either use the same app or create a new project and adapt the following method to your MainActivity.kt.



Dont just copy-paste the code! Make sure to compare the following code side-by-side with your Android Project and type the code by-hand..

```
package com.fit2081.myfirstapplication

import android.os.Bundle
import android.widget.Button
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {
    //We removed some boilerplate codes for the sake of simplicity

    private var clickCounter: Int = 0 //counts the number of clicks
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        //We removed some boilerplate code for the sake of simplicity..

        clickCounter=0; //initialize the counter
        val myButton: Button = findViewById(R.id.button) //find the button by its id

        myButton.setOnClickListener { //set a click listener on the button
            handleClick() //call the handleClick method when the button is clicked
        }
    }

    private fun handleClick(){
        clickCounter++ //increment the counter
        showToast("button clicked $clickCounter times") //show a toast message with the click count
    }

    private fun showToast(message: String) {
```

```
        Toast.makeText(this, message, Toast.LENGTH_SHORT).show() //show a toast message with the given message
    }
}
```

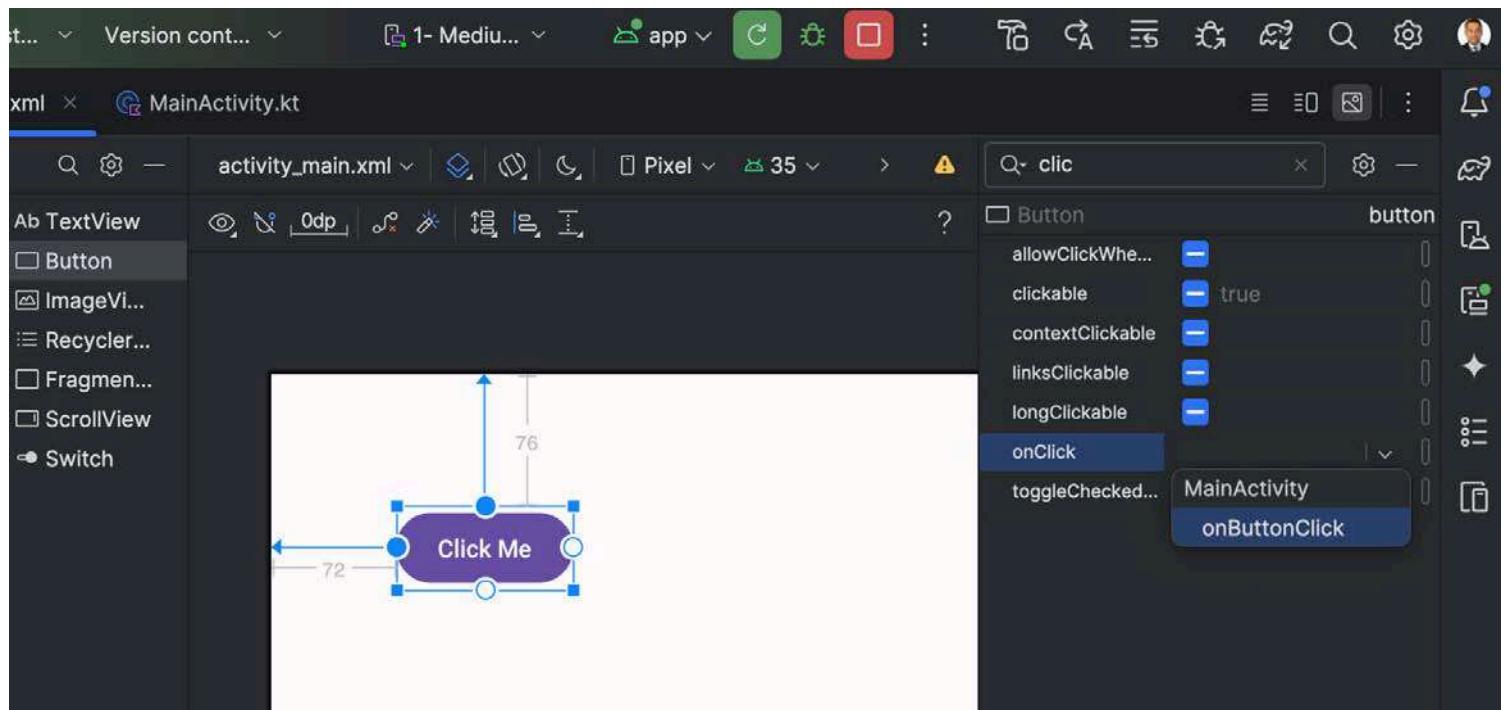
The purpose of lines 18-23 is to register the click events listener that invokes the method handleClick every time a click happens.

There is another way to link a method (function) to a button using the GUI.

- Rewrite your function to be an event handler (public and accept a view as a parameter)

```
public fun onButtonClick(view: View) {
    clickCounter++ //increment the counter
    showToast("button clicked $clickCounter times") //show a toast message with the click count
}
```

- Filter the attributes window by searching "click", as shown below.
- Find the onClick attribute and, using the dropdown, select the newly added method.

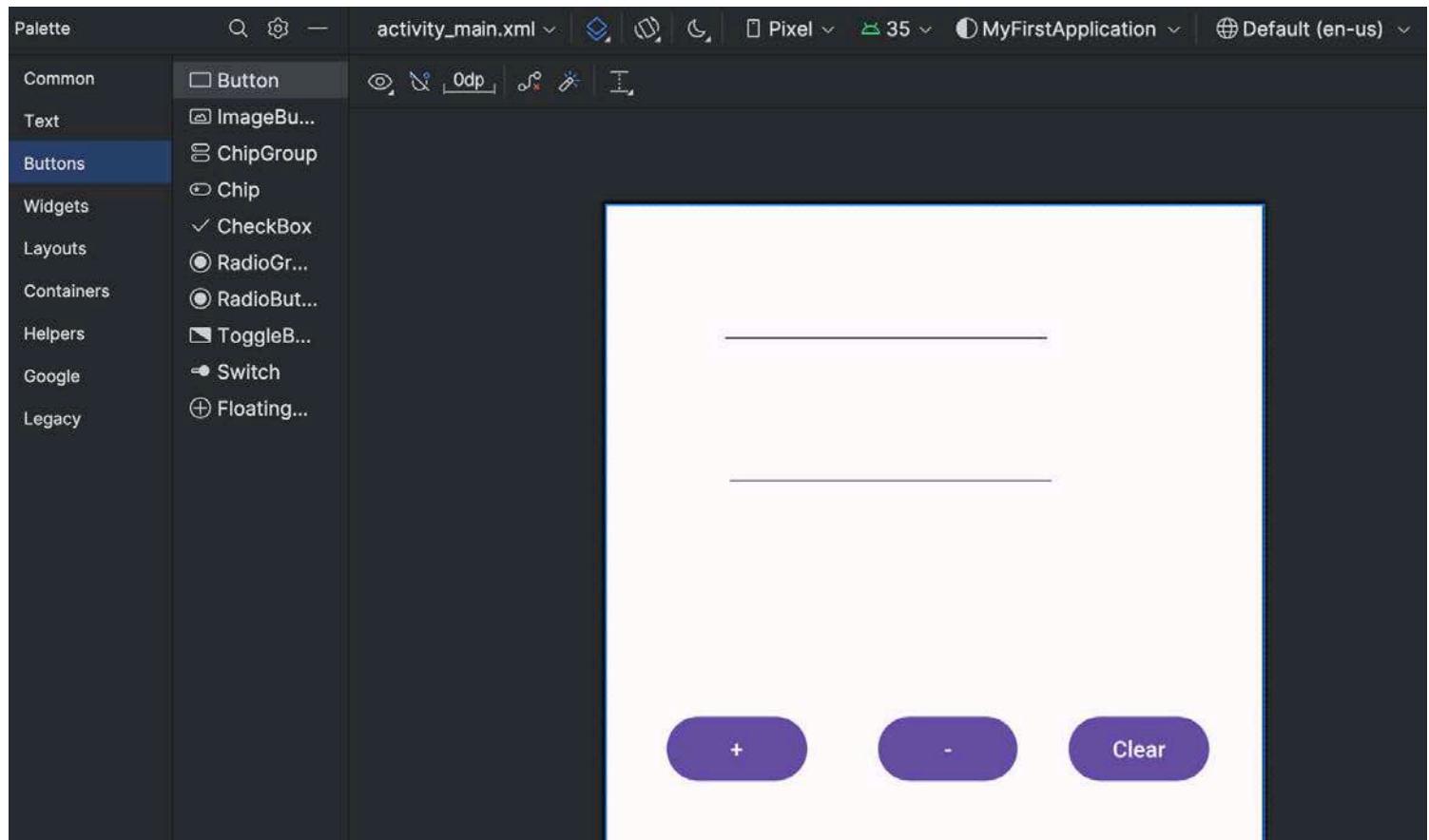


Attention: Don't use both methods together.

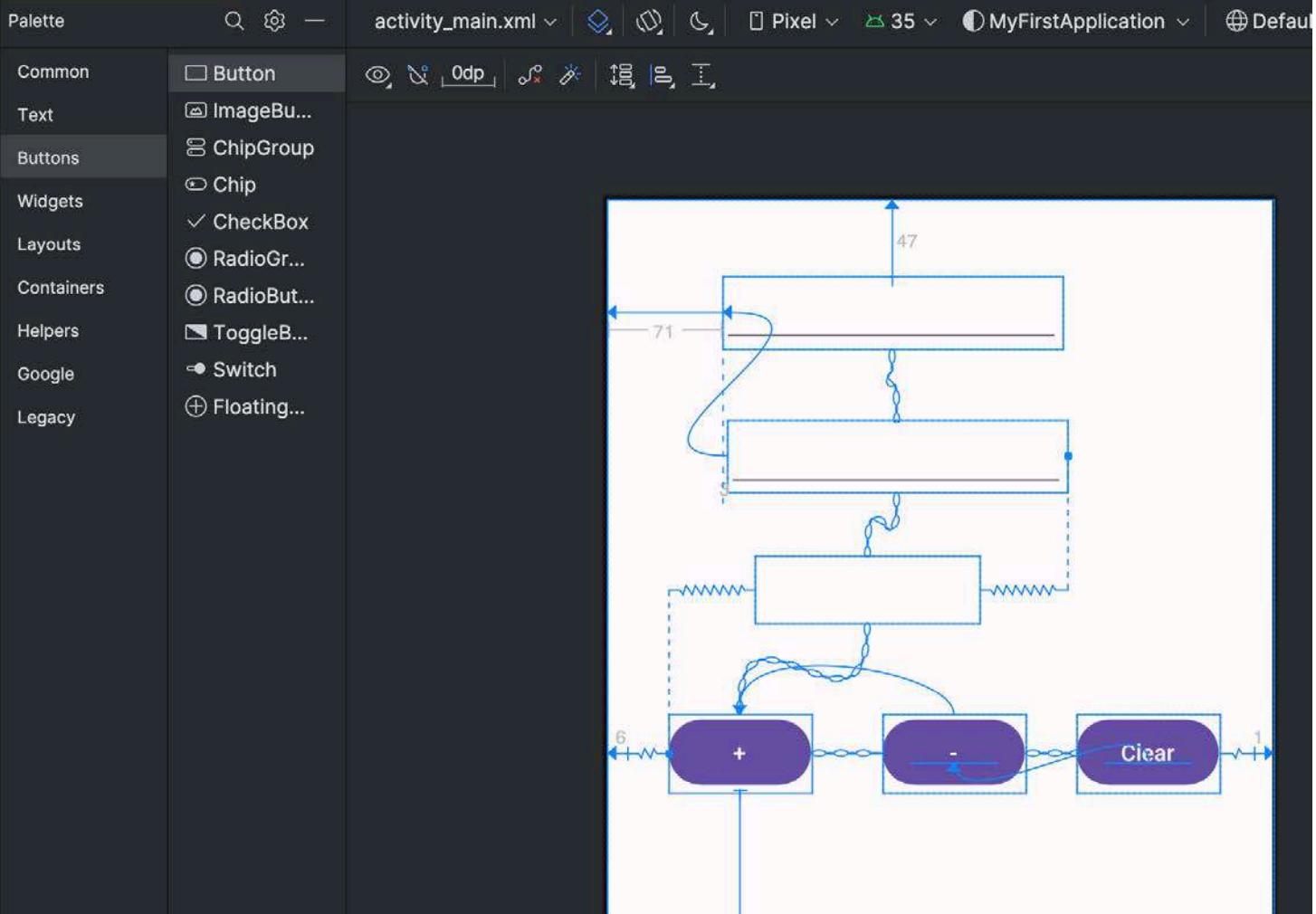
## 6. Simple Calculator

Now, let's develop a calculator with two arithmetic operations.

- Create a new project → New Project → Empty Views Activity
- Give a name and keep the default values for the others.
- Add Two edit texts (for the operands), one text view (for the results) and three buttons (two for the operations and one for clearing the screen). It should be like:



You can use the Infer Constraints button to add constraints



## Activity Source Code (Logic)

Firstly, create references to the six UI elements.

```
//references to the two edit text numbers
lateinit var firstNumberEditText: EditText
lateinit var secondNumberEditText: EditText

//references to the three buttons (two operations and one clear)
lateinit var addButton: Button
lateinit var subButton: Button
lateinit var clearButton: Button

//reference to the text view to display the result
lateinit var resultTextView: TextView
```

Now, inside the `onCreate` function, create references to the UI elements:

```
firstNumberEditText = findViewById(R.id.number1_id)
secondNumberEditText = findViewById(R.id.number2_id)
```

```
//connect the references to three buttons
addButton = findViewById(R.id.add_id)
subButton = findViewById(R.id.sub_id)
clearButton = findViewById(R.id.clear_id)

resultTextView = findViewById(R.id.result_id)
```

Develop a function that does the arithmetic operations:

```
private fun calculateResult(opt: String) {
    val num1Str = firstNumberEditText.text.toString()
    val num2Str = secondNumberEditText.text.toString()

    if (num1Str.isEmpty() || num2Str.isEmpty()) {
        resultTextView.text = "Please enter both numbers"
        return
    }

    var result: Int = 0
    val num1 = num1Str.toInt()
    val num2 = num2Str.toInt()
    if (opt == "add") {
        result = num1 + num2
    } else if (opt == "sub") {
        result = num1 - num2
    }
    resultTextView.text = "Result: $result"
}
```

Another one is to clear the UI

```
private fun clearAll() {
    firstNumberEditText.text.clear()
    secondNumberEditText.text.clear()
    resultTextView.text = ""
}
```

Now, add to onCreate the buttons' listeners:

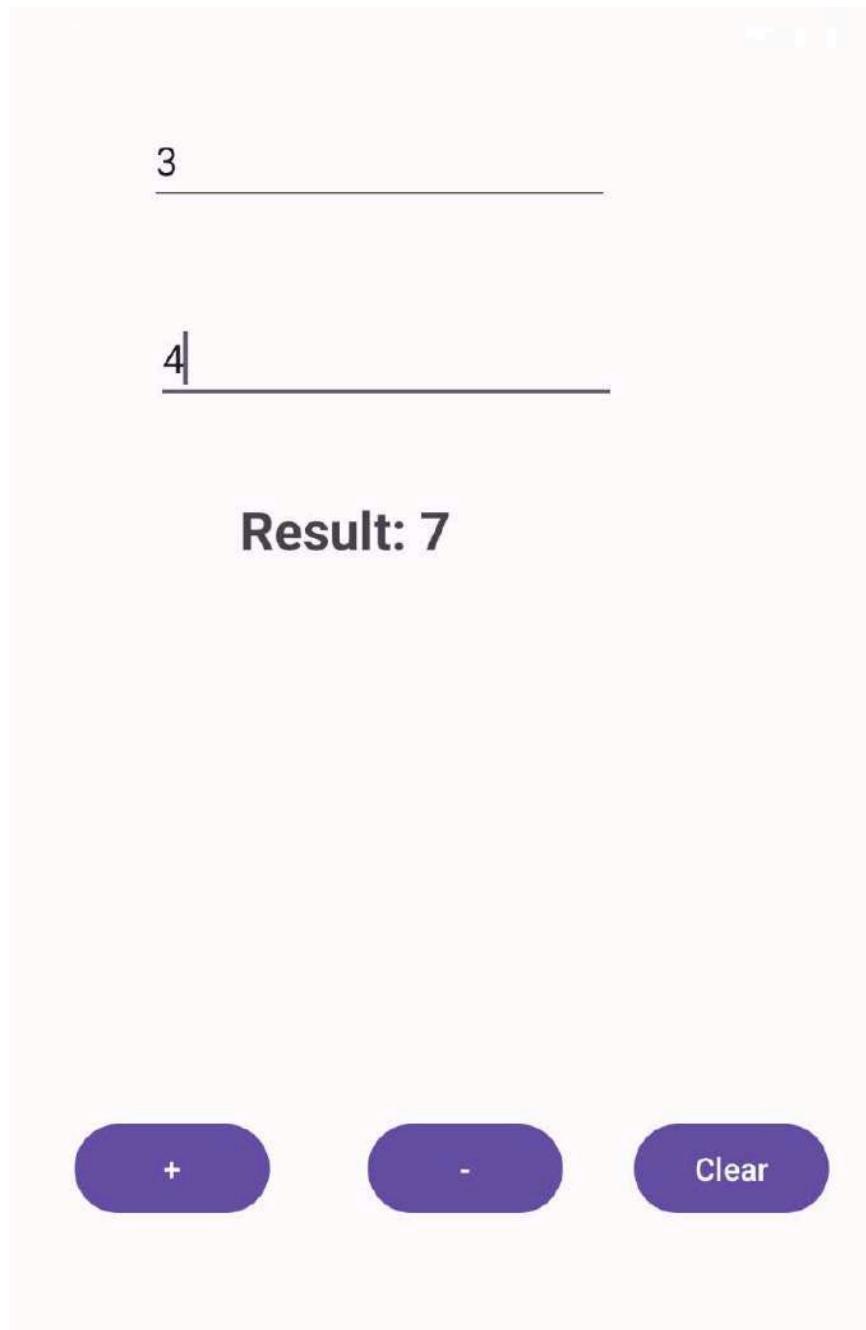
```
addButton.setOnClickListener {
    calculateResult("add")
}

subButton.setOnClickListener {
    calculateResult("sub")
```

```
}
```

```
clearButton.setOnClickListener {
    clearAll()
}
```

Run your app, and you should get the following screens:



3 \_\_\_\_\_

4 \_\_\_\_\_

**Result: 7**

+

-

Clear

## 7. Simple Calculator (Compose)

Overview:

If time permits, let's look at how the same functionality can be implemented using Jetpack Compose:

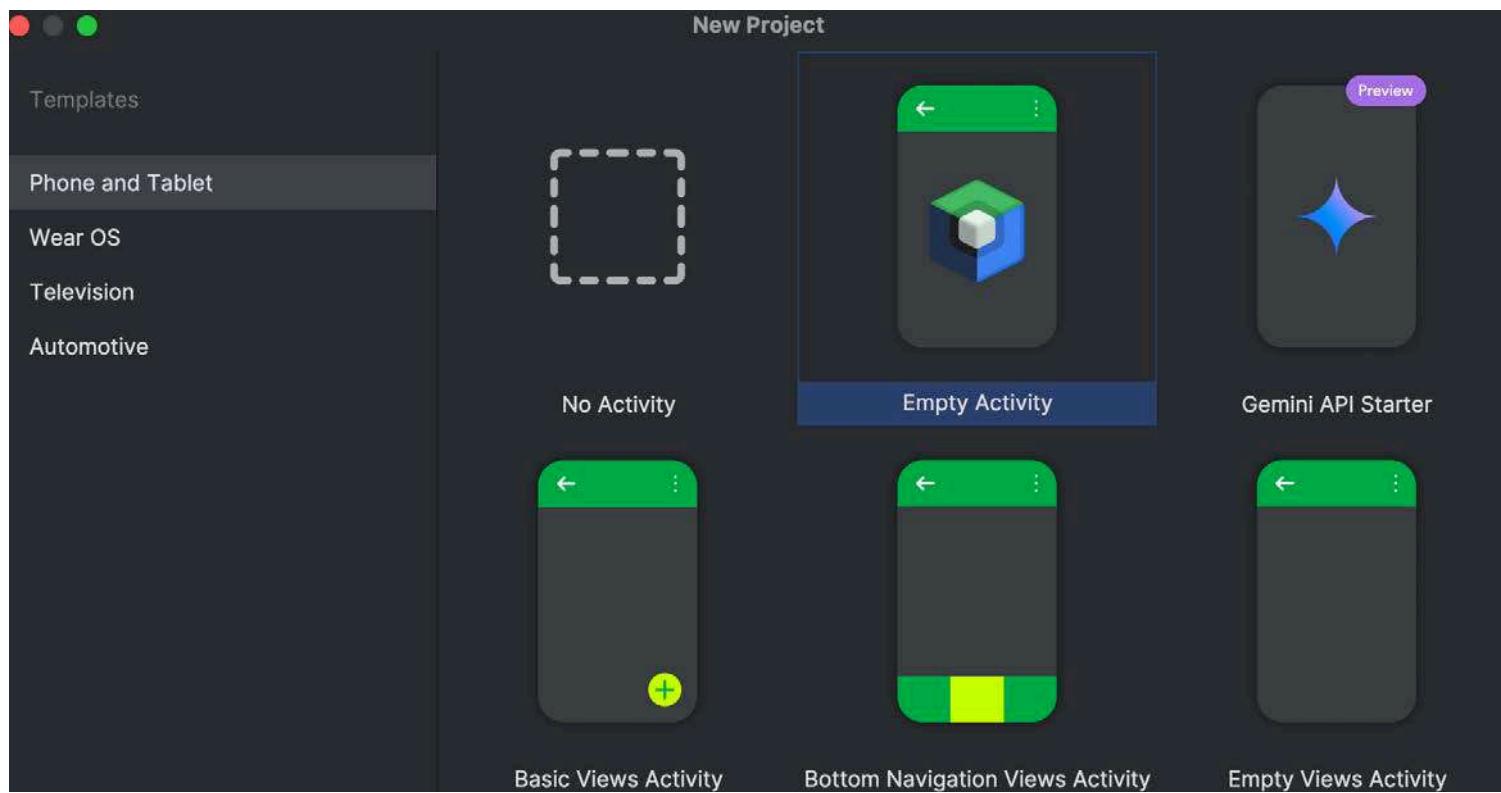
1. Create a new project with Empty Activity template (not Empty Views Activity)
2. Examine the generated MainActivity.kt file
3. Notice how UI elements are defined directly in Kotlin code

=====

Now, let's redevelop our simple calculator app using a new approach called "JetPack Compose". In this approach, we will design the UI elements and layout inside the Kotlin file file. In other words, there will be no XML file in our project.

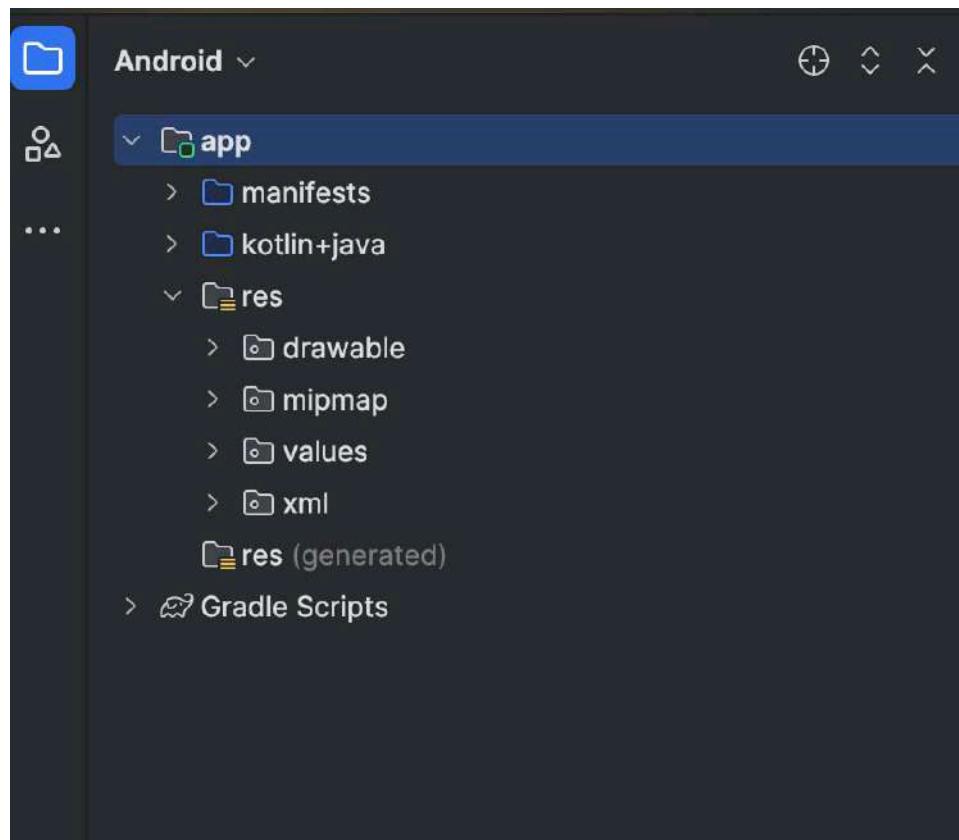
 For Flutter/Dart developers, you should be familiar with this approach

Let's get started by [\*\*creating a new project\*\*](#), but this time with 'Empty Activity' as shown below:

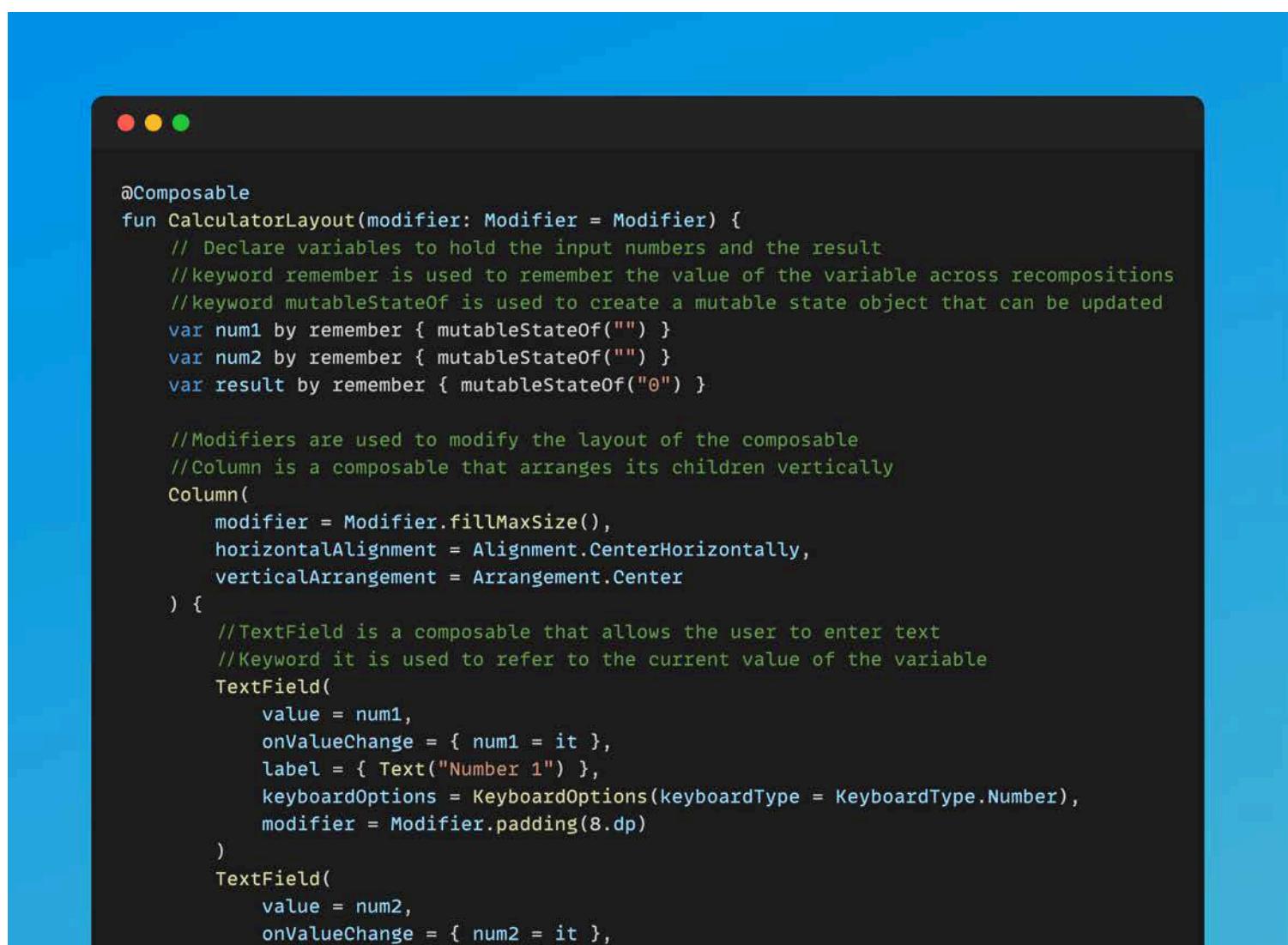


Give it a name and keep other attributes to their default values.

At first glance, you will see no layout folder in the res folder as we have no XML files, as we mentioned earlier.



Now, in the Kotlin file, add a new Composable function called 'CalculatorLayout' that is responsible for the activity's layout.



```
@Composable
fun CalculatorLayout(modifier: Modifier = Modifier) {
    // Declare variables to hold the input numbers and the result
    // Keyword remember is used to remember the value of the variable across recompositions
    // Keyword mutableStateOf is used to create a mutable state object that can be updated
    var num1 by remember { mutableStateOf("") }
    var num2 by remember { mutableStateOf("") }
    var result by remember { mutableStateOf("0") }

    // Modifiers are used to modify the layout of the composable
    // Column is a composable that arranges its children vertically
    Column(
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        // TextField is a composable that allows the user to enter text
        // Keyword it is used to refer to the current value of the variable
        TextField(
            value = num1,
            onValueChange = { num1 = it },
            label = { Text("Number 1") },
            keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
            modifier = Modifier.padding(8.dp)
        )
        TextField(
            value = num2,
            onValueChange = { num2 = it },
            label = { Text("Number 2") },
            keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
            modifier = Modifier.padding(8.dp)
        )
    }
}
```

```

        label = { Text("Number 2") },
        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
        modifier = Modifier.padding(8.dp)
    )

// Text is a composable that displays text
Text(
    text = "Result: $result",
    style = TextStyle(fontSize = 24.sp),
    textAlign = TextAlign.Center,
    modifier = Modifier.padding(16.dp)
)
// Row is a composable that arranges its children in a horizontal sequence
Row(
    horizontalArrangement = Arrangement.SpaceEvenly,
    modifier = Modifier.fillMaxWidth()
) {
    Button(onClick = {
        calculate("+", num1, num2)?.let {
            result = it
        }
    }) {
        Text("+")
    }
    Button(onClick = {
        calculate("-", num1, num2)?.let {
            result = it
        }
    }) {
        Text("-")
    }
    Button(onClick = {
        num1 = ""
        num2 = ""
        result = "0"
    }) {
        Text("Clear")
    }
}
}
}

```

As you can see, this function declares three variables of type strings that are used to hold the two operand numbers and the result. Then, the function adds a Column that positions the three TextField vertically. This Column element takes as input several parameters representing its attributes. We used to provide these attributes in the XML file. After that we added three TextFields and their attributes that are provided as parameters.

In order to position the three buttons horizontally, we used the Row element. Like the Column, we passed some attributes as parameters and added the three buttons as children. Each button takes the click event handler as input and defines a Text (formally TextView) representing its caption.

Now, there is a need to develop another function that does the arithmetic operations.

```
/*
 * Performs basic arithmetic calculations based on the provided operator and numbers.
 *
 * @param op The operation to perform, either "+" or "-".
 * @param num1Str The first number as a string.
 * @param num2Str The second number as a string.
 * @return The result of the calculation as a string, or null if the input is invalid.
 * If the operation is unsupported, it will return 0.
 * This function is designed to handle addition and subtraction,
 */
fun calculate(op: String, num1Str: String, num2Str: String): String? {
    //toDoubleOrNull is a function that converts a string to a double
    val number1 = num1Str.toDoubleOrNull()
    val number2 = num2Str.toDoubleOrNull()

    //This if statement is used to check if the input is valid
    if (number1 == null || number2 == null) {
        return null
    }
    //This when statement is used to perform the calculation based on the operator.
    // It is similar to a switch statement in other programming languages

    return when (op) {
        "+" → (number1 + number2).toString()
        "-" → (number1 - number2).toString()
        "*" → (number1 * number2).toString()
        else → "0"
    }
}
```

codesnap.dev

The method above is invoked by the event handlers of the first two buttons in the Row element.

Some screenshots of the output:

Number 1

Number 2

Result: 0

+

-

Clear

Number 1

3

Number 2

4

Result: 7.0

+

-

Clear

Here is the code of the MainActivity.kt file for your reference.

```
package com.fit2081.mycalcccompose

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.*
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.*
import androidx.compose.ui.text.input.KeyboardType
import androidx.compose.ui.unit.dp
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.material3.TextField
import androidx.compose.ui.Alignment
import androidx.compose.ui.text.TextStyle
import androidx.compose.ui.unit.sp
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.foundation.text.KeyboardOptions

import com.fit2081.mycalcccompose.ui.theme.MyCalcComposeTheme

/**
 * The main activity for the calculator app.
 */
class MainActivity : ComponentActivity() {
    /**
     * Called when the activity is starting. This is where most initialization should go.
     *
     * @param savedInstanceState If the activity is being re-initialized after previously being shut down
     *                         then this Bundle contains the data it most recently supplied.
     */
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        enableEdgeToEdge()
        //The setContent block defines the layout for the activity
        setContent {
            // Set the theme for the calculator app
            MyCalcComposeTheme {
                //The Scaffold component provides a basic structure for the calculator app
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    CalculatorLayout(
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }

    /**
     * Composable function that defines the layout of the calculator.
     *
     * @param modifier Modifier to apply to the layout.
     * This function sets up two input fields for numbers, a result display, and buttons for basic operations.
     */
    @Composable
    fun CalculatorLayout(modifier: Modifier = Modifier) {
        // Declare variables to hold the input numbers and the result
        //keyword remember is used to remember the value of the variable across recompositions
        //keyword mutableStateOf is used to create a mutable state object that can be updated
        var num1 by remember { mutableStateOf("") }
        var num2 by remember { mutableStateOf("") }
        var result by remember { mutableStateOf("0") }
    }
}
```

```

//Modifiers are used to modify the layout of the composable
//Column is a composable that arranges its children vertically
Column(
    modifier = Modifier.fillMaxSize(),
    horizontalAlignment = Alignment.CenterHorizontally,
    verticalArrangement = Arrangement.Center
) {
    //TextField is a composable that allows the user to enter text
    //Keyword it is used to refer to the current value of the variable
    TextField(
        value = num1,
        onValueChange = { num1 = it },
        label = { Text("Number 1") },
        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
        modifier = Modifier.padding(8.dp)
    )
    TextField(
        value = num2,
        onValueChange = { num2 = it },
        label = { Text("Number 2") },
        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
        modifier = Modifier.padding(8.dp)
    )
}

//Text is a composable that displays text
Text(
    text = "Result: $result",
    style = TextStyle(fontSize = 24.sp),
    textAlign = TextAlign.Center,
    modifier = Modifier.padding(16.dp)
)
//Row is a composable that arranges its children in a horizontal sequence
Row(
    horizontalArrangement = Arrangement.SpaceEvenly,
    modifier = Modifier.fillMaxWidth()
) {
    Button(onClick = {
        calculate("+", num1, num2)?.let {
            result = it
        }
    }) {
        Text("+")
    }
    Button(onClick = {
        calculate("-", num1, num2)?.let {
            result = it
        }
    }) {
        Text("-")
    }
    Button(onClick = {
        num1 = ""
        num2 = ""
        result = "0"
    }) {
        Text("Clear")
    }
}
}

/**
 * Performs basic arithmetic calculations based on the provided operator and numbers.
 *
 * @param op The operation to perform, either "+" or "-".
 * @param num1Str The first number as a string.
 * @param num2Str The second number as a string.
 * @return The result of the calculation as a string, or null if the input is invalid.
 * If the operation is unsupported, it will return 0.
 * This function is designed to handle addition and subtraction,
 */
fun calculate(op: String, num1Str: String, num2Str: String): String? {
    //toDoubleOrNull is a function that converts a string to a double
    val number1 = num1Str.toDoubleOrNull()
    val number2 = num2Str.toDoubleOrNull()
}

```

```
//This if statement is used to check if the input is valid
if (number1 == null || number2 == null) {
    return null
}

//This when statement is used to perform the calculation based on the operator.
// It is similar to a switch statement in other programming languages

return when (op) {
    "+" → (number1 + number2).toString()
    "-" → (number1 - number2).toString()
    "*" → (number1 * number2).toString()
    else → "0"
}
}
```

codesnap.dev

## Discussion points:

- How does state management differ in Compose vs. traditional View-based UI?
- What are the advantages of each approach?
- Which approach do you find more intuitive?

## 8. Challenge Exercises

If you have completed Activities 1-7, feel free to try out these challenge exercises to really stretch your Android skills!

1. Extend your calculator by creating a display that shows both the calculation history and current input
2. Implement a customizable interface for your calculator - Allow users to change color schemes (light, dark, high contrast).
3. (Very advanced) - Add haptic feedback options for button presses.
4. Create a domain-specific calculator. Build an alternate calculator for one of these domains.
  1. Health calculator - BMI -> Users can enter their weight and height, you need to display their BMI. Show a disclaimer at the top of the page that BMI is an inaccurate and misleading measure of someone's health status.
  2. Unit calculator - create a calculator for converting between "cms" to "inches" (and vice versa) & "kgs" to "lbs" (and vice versa).

# FIT2081 Week 2

# Kotlin Language Basics

# Lecture: Android, API Levels, Fragmentation

---

## Overview: Lecture Topics

Here are the **detailed lecture notes** for your students, updated to reflect the focus on **Kotlin and Jetpack Compose** and removing outdated references.

## FIT2081 – Lecture 2: Introduction to Android Development & Kotlin

### Week 2 – Understanding the Android Ecosystem & Setting Up Your Development Environment

#### 1. Introduction

Welcome to Week 2! This lecture will introduce you to **Android development using Kotlin and Jetpack Compose**. By the end of this session, you should:

- Understand the **Android ecosystem and its evolution**.
- Learn **why Kotlin** is the preferred language for Android development.
- Grasp the **importance of Jetpack Compose** and how it differs from traditional XML-based UI design.
- Get comfortable with **Android Studio and emulator setup**.
- Understand the **basic project structure in Android Studio**.

#### 2. The Android Ecosystem

##### What is Android?

Android is an **open-source mobile operating system** developed by Google, running on **billions of devices worldwide**. It powers:

- Smartphones
- Tablets
- Wearables (WearOS)
- Smart TVs (Android TV)
- IoT devices (Android Things)

##### Why Kotlin for Android Development?

Kotlin is now the **official language for Android development** (since 2017). It offers:

- Concise & Readable Code** → Less boilerplate than Java
- Null Safety** → Prevents NullPointerExceptions
- Coroutines for Asynchronous Programming** → Handles background tasks efficiently
- Interoperability** → Works seamlessly with existing Java libraries

Google recommends Kotlin-first development for Android apps due to its simplicity and efficiency.

## 3. Recap: The Shift from XML to Jetpack Compose

### What is Jetpack Compose?

Jetpack Compose is a **modern UI toolkit** for building native Android interfaces **declaratively**. Instead of XML layouts, UI components are built using **Kotlin functions**.

### Comparison of XML vs Jetpack Compose

### Example UI Code Comparison

#### XML-Based UI (Old)

```
<Button  
    android:text="Click Me"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:onClick="handleClick"/>
```

#### Jetpack Compose UI (New)

```
@Composable  
fun MyButton() {  
    Button(onClick = { /* Handle Click */ }) {  
        Text("Click Me")  
    }  
}
```

**Compose is faster and requires fewer lines of code!**

## 4. Setting Up Your Development Environment

### Installing Android Studio & SDKs

Download **Android Studio** from [developer.android.com/studio](https://developer.android.com/studio)

Install the necessary **SDKs & Emulator**

Create a new project using **Jetpack Compose**

### Android Project Structure Overview

When you create a new project, your structure will look like this:

```
app/
└── src/
    ├── main/
    │   ├── java/com/example/myapp/ (Kotlin source files)
    │   ├── res/ (Resources like images, colors, and themes)
    │   └── AndroidManifest.xml (App configuration)
    └── test/ (Unit tests)
├── build.gradle (Project dependencies)
└── settings.gradle (Gradle settings)
```

### Key Project Files

**MainActivity.kt** → The entry point of your app

**AndroidManifest.xml** → Declares app permissions and components

**build.gradle** → Manages dependencies and SDK versions

**themes.xml (Optional)** → Defines theme colors and styles

## 5. Running Your First App

### Step 1: Create a New Project

- Choose "**Empty Compose Activity**"
- Select **Kotlin** as the language
- Set the **minimum SDK** (usually API Level 26+)

### Step 2: Explore the Default Compose Code

When you open **MainActivity.kt**, you'll see:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
```

```

        super.onCreate(savedInstanceState)
        setContentView {
            MyApp()
        }
    }

}

@Composable
fun MyApp() {
    Text("Hello, Android!")
}

```

### Explanation:

- `ComponentActivity()` → Entry point for your app.
- `setContent {}` → Wraps the UI inside a **Composable function**.
- `@Composable` → Marks a function that defines UI elements.

## Step 3: Run on Emulator or Physical Device

Select an emulator (e.g., **Pixel 5 API 33**)

Click **Run ►** in Android Studio

If using a physical device, enable **USB Debugging**

## 6. Understanding State in Jetpack Compose

Jetpack Compose uses **State Management** to update the UI automatically.

### Example: Counter App

```

@Composable
fun CounterApp() {
    var count by remember { mutableStateOf(0) }

    Column(
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text("Count: $count", fontSize = 24.sp)
        Button(onClick = { count++ }) {
            Text("Increment")
        }
    }
}

```

```
}
```

## Key Concepts:

- `remember { mutableStateOf(0) }` → **State variable** that triggers recomposition when changed.
- `Button(onClick = { count++ })` → Updates the state when clicked.
- `Text("Count: $count")` → UI updates automatically when `count` changes.

## 7. Navigating Between Screens with Jetpack Compose

Unlike traditional **Intents**, Compose uses the **Navigation Component**.

### Setting Up Navigation

Add dependency in `build.gradle`

```
dependencies {
    implementation "androidx.navigation:navigation-compose:2.5.3"
}
```

Define Navigation in Compose:

```
@Composable
fun MyAppNav() {
    val navController = rememberNavController()

    NavHost(navController, startDestination = "home") {
        composable("home") { HomeScreen(navController) }
        composable("details") { DetailsScreen() }
    }
}
```

## 8. Debugging & Logcat

### Using Logcat to Debug in Android Studio

Add a **log message** to check variables:

```
Log.d("MainActivity", "Button clicked!")
```

Open **Logcat Panel** → Filter logs using `MainActivity`      Run the app and observe logs.

## 9. Summary & Next Steps

## Key Takeaways

- Kotlin is the recommended language for Android development
- Jetpack Compose replaces XML for UI design
- State management automatically updates UI
- Navigation is simplified with Compose's Navigation Component
- Use Logcat for debugging

## Next Steps

### Workshop Practice:

- Write a simple Compose UI with a **TextField, Button, and State**
- Try **navigating between two screens** using **NavHost**

# What is Android?

- It's an Operating System + ...
  - "An operating system (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs"
  - Notes
    - In the case of the Android OS, the top level programs are called Apps
      - Some Apps come with the OS
      - Google Apps + OEM and/or Telco Apps
    - Android includes a run-time environment for Apps
      - How it manages their coexistence will be of particular interest to us
    - Like most OSs, Android consists of several layers of software
      - Lower layers (in C/C++) perform lower-level functions for higher layers (in Java) that therefore do not have to concern themselves with lower-level details (e.g. connect to the internet)
    - Architectured as a Linux Kernel + a Software Stack
      - Linux kernel OS
      - Android Software Stack
- [...] the Linux kernel provides a multitasking execution environment allowing multiple processes to execute concurrently. Therefore, it would be easy to assume that each Android application simply runs as a process directly on the Linux kernel. In actual fact, each application running on an Android device does so within its own instance of the ~~Dalvik~~ [Android Run Time (ART) since Lollipop] virtual machine [within its own Linux Kernel Process].
  - Running applications in virtual machines provides several advantages. Firstly, applications are essentially sandboxed, in that they cannot detrimentally interfere (intentionally or otherwise) with the operating system, other applications or directly access the device hardware. Secondly, this enforced level of abstraction makes applications platform neutral in that they are never tied to any specific hardware.
- The ~~Dalvik~~ ART virtual machine was developed by Google and relies on the underlying Linux kernel for low-level functionality. It is more efficient than the standard Java VM in terms of memory usage and specifically designed to allow multiple instances to run efficiently within the resource constraints of a mobile device."

# Android Characteristics

- Linux-based
- Lightweight (wrt device resources)
- Designed for touchscreen mobile devices
  - e.g. smartphones and tablets
    - Other devices include laptops/netbooks, smart TVs, wristwatches, headphones, games consoles etc.
  - So, optimised to operate within the typical limitations of such devices (processing, power, memory, screen size...)
  - So, designed to support Apps using typical hardware features of such devices.
    - e.g. touchscreen, WiFi, accelerometers, gyroscopes, proximity sensors ...
- Open Source (Apache License)
  - Free source code available for modification and redistribution
  - The most permissive open-source license without even the need to push modifications back into the open-source community
    - So enhancements do not need to be open source
    - Very business/start-up friendly
    - Fragmentation???

# Who “Owns” Android?

- Google
  - Purchased from Android Inc. in 2005 and made Open Source
  - They create and release new versions of Android
    - These can include updates to:
      - The software stack
        - Including the API (the main focus of Android App developers)
      - The Linux Kernel
        - which is now on a separate branch to the mainline Linux Kernel following architectural changes to the kernel by Google
      - Bundled Apps + Development tools + ...
    - Android users are significantly fragmented among several versions of Android
      - This creates forward and backward compatibility complications for Android App developers and users
        - with iOS, this problem is much less severe
    - They maintain the largest Android App store
      - Google Play (formerly Android Market)
      - A distribution site for Apps and other media published by Google

# The Android Ecosystem

- What is the Android Ecosystem?
  - A set of interdependent, evolving components that together enable the creation and distribution of Android Apps
    - Cooperation between these components is essential to the well-being of the Android Ecosystem
      - Many problems have arisen as a result of the lack of cooperation (see next slide)
    - Compare this with the iOS Ecosystem, where nearly all components are controlled by Apple, which ensures cooperation.
  - Components include hardware manufacturers, core and development software (IDEs) creators, distribution channels (marketplaces etc.), app developers and their communities, telcos, etc ...
    - i.e. the OHA + other App developers + marketplaces + users

# Android Versions

- SDK Platform
  - The entire SDK
  - Including a version of the framework API
  - Incremented using n.n.n format
- API Level
  - “API Level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform”
  - Framework API
    - “The Android platform provides a framework API that applications can use to interact with the underlying Android system. The framework API consists of the following:
      - A core set of packages and classes
      - A set of XML elements and attributes for declaring a manifest file
      - A set of XML elements and attributes for declaring and accessing resources
      - A set of Intents
      - A set of permissions that applications can request, as well as permission enforcements included in the system

## API Level

 SDK version means API version here, which of course belongs to a major Platform version

The minimum SDK version determines the lowest level of Android that your app will run on. You typically want to target as many users as possible, so you would ideally want to support everyone -- with a minimum SDK version of 1. However, that has some disadvantages, such as a lack of features, and very few people use devices that are old anymore. Your choice of minimum SDK level should be a tradeoff between the distribution of users you wish to target and the features your application will need. Click each Android Version/API level for more information.

## API Distribution

Platform Version	API	Distribution
Android 4.4 (KitKat)	19	0.7%
Android 5.0 (Lollipop)	21	0.3%
Android 5.1 (Lollipop)	22	1.8%
Android 6.0 (Marshmallow)	23	2.8%
Android 7.0 (Nougat)	24	1.9%
Android 7.1 (Nougat)	25	1.8%
Android 8.0 (Oreo)	26	2.6%
Android 8.1 (Oreo)	27	6.9%
Android 9.0 (Pie)	28	13.2%
Android 10.0 (Q)	29	19.5%
Android 11.0 (R)	30	24.4%
Android 12.0 (S)	31	18.9%
Android 13.0 (T)	33	5.0%

# API Fragmentation

- Developers must deal with fragmentation
- Device Fragmentation
  - From watches to giant TV screens
  - What range do you want to target?
  - What range is it feasible to target?
  - Android supports fluid UI design + UI related resource selection based on device characteristics when the fluid design is stretched too far

# API Fragmentation

- "Compared to its chief rival mobile operating system, namely iOS, Android updates are typically slow to reach actual devices.
- For devices not under the Nexus brand, updates often arrive months from the time the given version is officially released. This is caused partly due to the extensive variation in hardware of Android devices, to which each update must be specifically tailored, as the official Google source code only runs on their flagship Nexus phone.
- Porting Android to specific hardware\* is a time- and resource-consuming process for device manufacturers, who prioritize their newest devices and often leave older ones behind. Hence, older smartphones are frequently not updated if the manufacturer decides it is not worth their time, regardless of whether the phone is capable of running the update.
- This problem is compounded when manufacturers customize Android with their own interface and apps, which must be reapplied to each new release.
- Additional delays can be introduced by wireless carriers who, after receiving updates from manufacturers, further customize and brand Android to their needs and conduct extensive testing on their networks before sending the update out to users."

# Forward and Backward Compatibility

- Forward Compatibility
  - Old apps running on new platform versions
  - "Android applications are generally forward-compatible with new versions of the Android platform. Because almost all changes to the framework API are additive"
    - "...except in isolated cases where the application uses a part of the API that is later removed for some reason."
  - This is essential considering OTA (over the air) platform updates
- Backward Compatibility

- New apps (using new features?) running on old platform versions
- “Android applications are not necessarily backward compatible with versions of the Android platform older than the version against which they were compiled. Each new version of the Android platform can include new framework APIs, such as those that give applications access to new platform capabilities or replace existing API parts.”

# Android Components

## Activities

- “Android applications are created by bringing together one or more components known as Activities. An activity is a single, standalone module of application functionality which usually correlates directly to a single user interface screen and its corresponding functionality”
- [Activities have lifecycles as they are partially or fully hidden or killed by the OS. Developers must code callbacks (methods that are auto-fired as partial- and full-hiding and kill events occur) to respond appropriately)]
- “An activity represents a single screen with a user interface. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others. As such, a different app can start any one of these activities (if the email app allows it). For example, a camera app can start the activity in the email app that composes new mail in order for the user to share a picture.
  - An activity is implemented as a subclass of Activity, and you can learn more about it in the Activities developer guide.”

## Services

- “Android Services are processes that run in the background and do not have a user interface. They can be started and subsequently managed from Activities, Broadcast Receivers or other Services. Android Services are ideal for situations where an application needs to continue performing tasks but does not necessarily need a user interface to be visible to the user [e.g. music player app]”
- “A service is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface. For example, a service might play music in the background while the user is in a different app, or it might fetch data over the network without blocking user interaction with an activity. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it.
  - A service is implemented as a subclass of Service and you can learn more about it in the Services developer guide.”

## Content Providers

- “Content Providers implement a mechanism for the sharing of data between applications. Any application can provide other applications with access to its underlying data through the

implementation of a Content Provider including the ability to add, remove and query the data (subject to permissions). Access to the data is provided via a Universal Resource Identifier (URI) defined by the Content Provider. Data can be shared in the form [of] a file or an entire SQLite database.

- The native Android applications include a number of standard Content Providers allowing applications to access data such as contacts and media files.
- The Content Providers currently available on an Android system may be located using a Content Resolver.”
- “A content provider manages a shared set of app data. You can store the data in the file system, an SQLite database, on the web, or any other persistent storage location your app can access. Through the content provider, other apps can query or even modify the data (if the content provider allows it). For example, the Android system provides a content provider that manages the user's contact information. As such, any app with the proper permissions can query part of the content provider (such as `ContactsContract.Data`) to read and write information about a particular person.
- Content providers are also useful for reading and writing data that is private to your app and not shared. [...].
- A content provider is implemented as a subclass of `ContentProvider` and must implement a standard set of APIs that enable other apps to perform transactions. For more information, see the Content Providers developer guide.”

## What is the Content Resolver?

- The Content Resolver is the single, global instance in your application that provides access to your (and other applications') content providers. The Content Resolver behaves exactly as its name implies: it accepts requests from clients and resolves these requests by directing them to the content provider with a distinct authority. To do this, the Content Resolver stores a mapping from authorities to Content Providers. This design is important, as it allows a simple and secure means of accessing other applications' Content Providers.
- The Content Resolver includes the CRUD (create, read, update, delete) methods corresponding to the abstract methods (insert, query, update, delete) in the Content Provider class. The Content Resolver does not know the implementation of the Content Providers it is interacting with (nor does it need to know); each method is passed an URI that specifies the Content Provider to interact with.

## What is a Content Provider?

- Whereas the Content Resolver provides an abstraction from the application's Content Providers, Content Providers provide an abstraction from the underlying data source (i.e. a SQLite database). They provide mechanisms for defining data security (i.e. by enforcing read/write

permissions) and offer a standard interface that connects data in one process with code running in another process.

- Content Providers provide an interface for publishing and consuming data, based around a simple URI addressing model using the content:// schema. They enable you to decouple your application layers from the underlying data layers, making your application data-source agnostic by abstracting the underlying data source."

## Broadcast Receivers

- "Broadcast Receivers are the mechanism by which applications are able to respond to Broadcast Intents. A Broadcast Receiver must be registered by an application and configured with an Intent Filter to indicate the types of broadcast in which it is interested [\*]. When a matching intent is broadcast, the receiver will be invoked by the Android runtime regardless of whether the application that registered the receiver is currently running. The receiver then has 5 seconds in which to complete any tasks required of it (such as launching a Service, making data updates or issuing a notification to the user) before returning. Broadcast Receivers operate in the background and do not have a user interface."
- "A broadcast receiver is a component that responds to system-wide broadcast announcements. Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured. Apps can also initiate broadcasts—for example, to let other apps know that some data has been downloaded to the device and is available for them to use. Although broadcast receivers don't display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs. More commonly, though, a broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work. For instance, it might initiate a service to perform some work based on the event.
  - A broadcast receiver is implemented as a subclass of BroadcastReceiver and each broadcast is delivered as an Intent object. For more information, see the BroadcastReceiver class."

# Activating Components - Intents

- Intents
  - “Three of the four component types—activities, services, and broadcast receivers—are activated by an asynchronous message called an intent. Intents bind individual components to each other at runtime (you can think of them as the messengers that request an action from other components), whether the component belongs to your app or another.”
  - An intent is created with an Intent object, which defines a message to activate either a specific component or a specific type of component—an intent can be either explicit or implicit, respectively.”
- Activity Intents
  - “Intents are the mechanism by which one activity is able to launch another [activity or service] and implement the flow through the activities that make up an application. Intents consist of a description of the operation to be performed and, optionally, the data on which it is to be performed.
  - Intents can be explicit, in that they request the launch of a specific activity by referencing the activity by class name, or implicit by stating either the type of action to be performed or providing data of a specific type on which the action is to be performed. In the case of implicit intents, the Android runtime will select the activity to launch that most closely matches the criteria specified by the Intent using a process referred to as Intent Resolution.”
- Broadcast Intents
  - Another type of Intent, the Broadcast Intent, is a system wide intent that is sent out to all applications that have registered an “interested” Broadcast Receiver. The Android system, for example, will typically send out Broadcast Intents to indicate changes in device status such as the completion of system start up, connection of an external power source to the device or the screen being turned on or off.
- Inter-App Activation
  - “A unique aspect of the Android system design is that any app can start another app’s component.”
  - “When the system starts a component, it starts the process for that app (if it’s not already running) and instantiates the classes needed for the component.”
  - “Because the system runs each app in a separate process with file permissions that

restrict access to other apps, your app cannot directly activate a component from another app. The Android system, however, can. So, to activate a component in another app, you must deliver a message to the system that specifies your intent to start a particular component. The system then activates the component for you."

- An app must give its permission for any of its components to be activated by an external intent
- Content Providers (not activated by intents)
  - "Unlike activities, services, and broadcast receivers, content providers are not activated by intents. Rather, they are activated when targeted by a request from a ContentResolver. The content resolver handles all direct transactions with the content provider so that the component that's performing transactions with the provider doesn't need to and instead calls methods on the ContentResolver object. This leaves a layer of abstraction between the content provider and the component requesting information (for security)."

# Application Manifest, Resources, & Context

## Manifest File

- It's an XML file
  - It details all of an App's components, their capabilities and more
- Includes
  - A declaration of all components in the application
    - Including for each, any capabilities wrt implicit inter-App intents
  - In most cases if a component is not declared the system can't see it
- "The primary task of the manifest is to inform the system about the app's components."
- "The manifest does a number of things in addition to declaring the app's components, such as the following:
  - Identifies any user permissions the app requires, such as Internet access or read-access to the user's contacts.
  - Declares the minimum API Level required by the app, based on which APIs the app uses.
  - Declares hardware and software features used or required by the app, such as a camera, bluetooth services, or a multitouch screen.
  - Declares API libraries the app needs to be linked against (other than the Android framework APIs), such as the Google Maps library."

## Resources

- "In addition to the manifest file and the Dex files that contain the byte code, an Android application package [APK] will also typically contain a collection of resource files. These files contain resources such as the strings, images, fonts and colors that appear in the user interface together with the XML representation of the user interface layouts. By default, these files are stored in the /res sub-directory of the application project's hierarchy."
- "An Android app is composed of more than just code—it requires resources that are separate from the source code, such as images, audio files, and anything relating to the visual presentation of the app. For example, you should define animations, menus, styles, colors, and the layout of activity user interfaces with XML files. Using app resources makes it easy to update various characteristics of your app without modifying code and—by providing sets of alternative resources—enables you to optimize your app for a variety of device configurations (such as different languages and screen sizes)."

- “For every resource that you include in your Android project, the SDK build tools define a unique integer ID, which you can use to reference the resource from your app code or from other resources defined in XML. For example, if your app contains an image file named logo.png (saved in the res/drawable/ directory), the SDK tools generate a resource ID named R.drawable.logo, which you can use to reference the image and insert it in your user interface.”
- “One of the most important aspects of providing resources separate from your source code is the ability for you to provide alternative resources for different device configurations. For example, by defining UI strings in XML, you can translate the strings into other languages and save those strings in separate files. Then, based on a language qualifier that you append to the resource directory's name (such as res/values-fr/ for French string values) and the user's language setting, the Android system applies the appropriate language strings to your UI.”

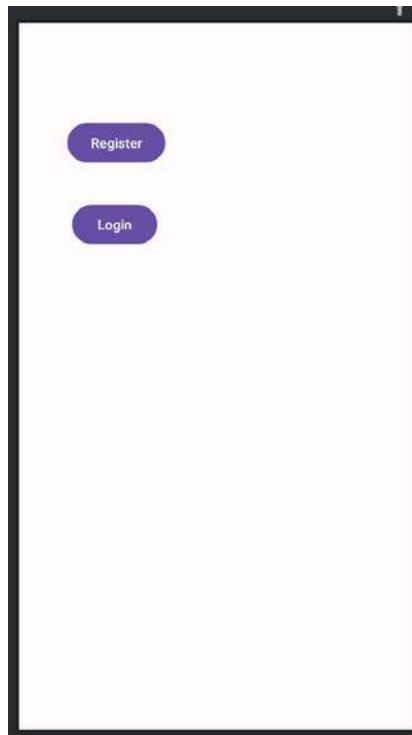
## Context

- “Interface to global information about an application environment. This is an abstract class whose implementation is provided by the Android system. It allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, broadcasting and receiving intents, etc.”
- “When an application is compiled, a class named R is created that contains references to the application resources. The application manifest file and these resources combine to create what is known as the Application Context. This context, represented by the Android Context class, may be used in the application code to gain access to the application resources at runtime. In addition, a wide range of methods may be called on an application's context to gather information and make changes to the application's environment at runtime.”

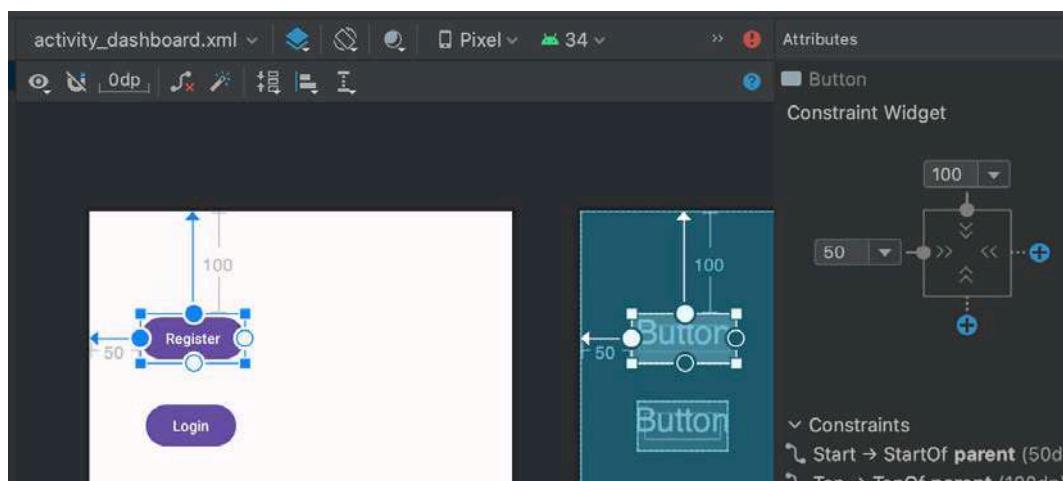
# Appendix: How to launch another activity in Java/XML

To give you an idea of how

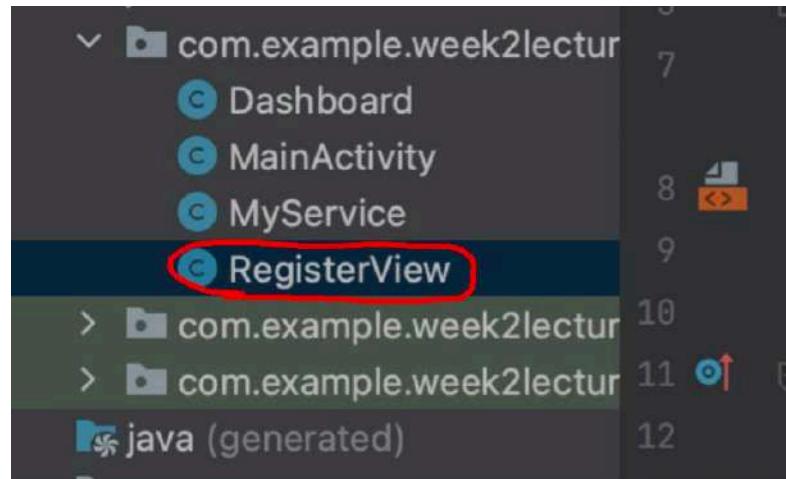
1. Create a Blank Activity with the following buttons



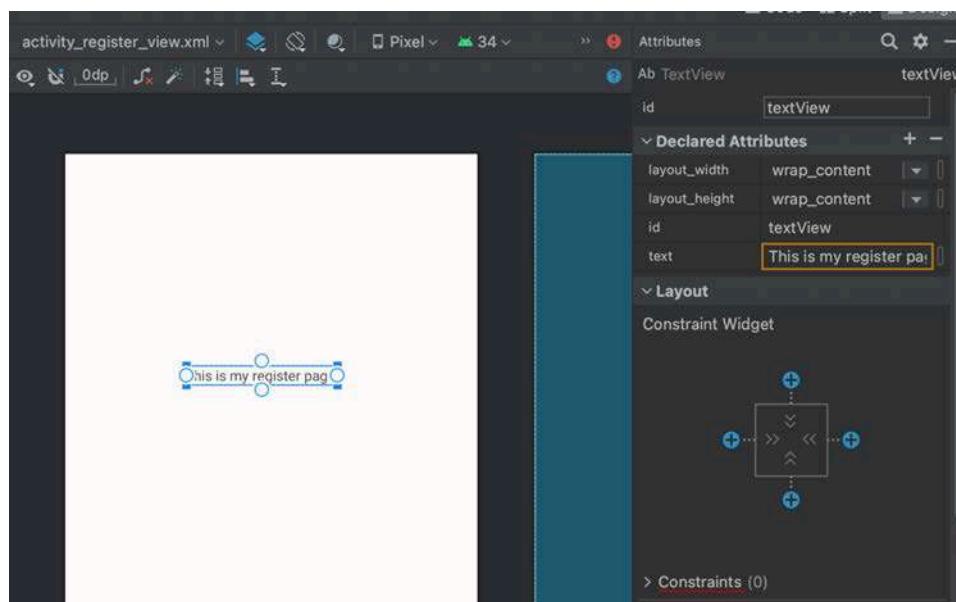
2. Click the Register button, and add constraints...



3. Create a new Blank Activity called RegisterView



4. In RegisterView, add a TextView and show a Toast (for confirmation purposes).

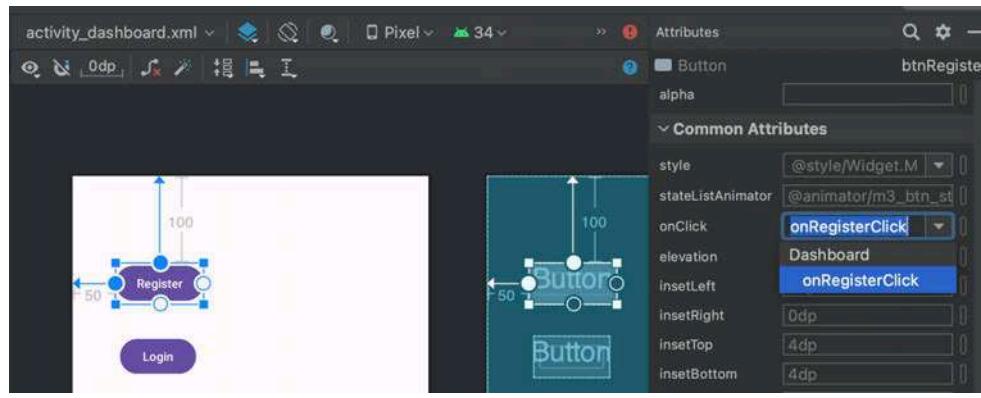


Add a toast to the bottom of the onCreate() method:

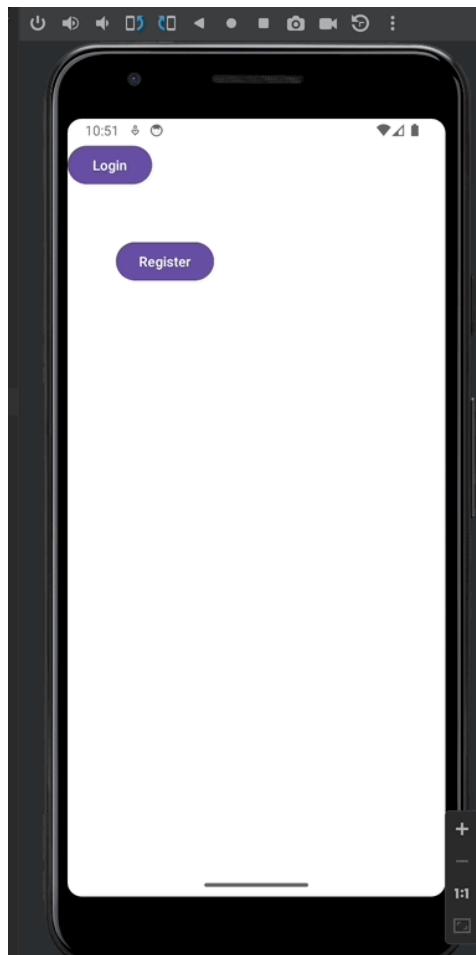
```
Toast.makeText(this, "Welcome to Register page", Toast.LENGTH_LONG).show();
```

5. Create a method for the onClick method and add this

```
public void onRegisterClick(View view){  
    Intent intent = new Intent(this, RegisterView.class);  
    startActivity(intent);  
}
```



You should now have successfully created a multi-activity app!



# Lecture 2: Code from Live Coding

**i** Note: this is the full application code building on from what was done in the Week 2 Lecture. We didn't complete the entire application in the lecture but for your benefit it is provided here.

**This is what we are going to build:**

## Simple Quiz Application



[W2Lecturev1-AndroidStudio.zip](#)

### MainActivity.kt

```
package com.fit20812025.myapplication
```

```
import android.content.Intent
import android.os.Bundle
import android.widget.Toast
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.Image
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.width
import androidx.compose.material3.Button
import androidx.compose.material3.Checkbox
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.unit.dp
import com.fit20812025.myapplication.ui.theme.W2Lecturev1Theme

private const val q1a_answer = true
private const val q1b_answer = false
private const val q2_answer = "Samsung"

class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            W2Lecturev1Theme {
                Scaffold(
                    modifier = Modifier.fillMaxSize()
                ) { innerPadding ->
                    QuizScreen(modifier = Modifier.padding(innerPadding))
                }
            }
        }
    }
}
```

```
@Composable
fun QuizScreen(modifier: Modifier = Modifier) {
    var optionOneChecked by remember { mutableStateOf(false) }
    var optionTwoChecked by remember { mutableStateOf(false) }
    var questionTwoAnswer by remember { mutableStateOf("") }
    Column(
        modifier = modifier.padding(16.dp).fillMaxSize(),
        horizontalAlignment = androidx.compose.ui.Alignment.CenterHorizontally
    ) {
        QuizHeader()
        Spacer(modifier = Modifier.height(16.dp))

        // Pass down current checkbox states + callbacks
        QuestionOne(
            optionOneChecked = optionOneChecked,
            onOptionOneCheckedChange = { newValue ->
                optionOneChecked = newValue
            },
            optionTwoChecked = optionTwoChecked,
            onOptionTwoCheckedChange = { newValue ->
                optionTwoChecked = newValue
            }
        )
        Spacer(modifier = Modifier.height(16.dp))

        // Pass down Q2's answer state + callback
        QuestionTwo(
            answer = questionTwoAnswer,
            onAnswerChange = { newAnswer ->
                questionTwoAnswer = newAnswer
            }
        )
        Spacer(modifier = Modifier.height(16.dp))

        SubmitButton( optionOneChecked, optionTwoChecked, questionTwoAnswer )
    }
}
```

```
@Composable
fun QuizHeader() {
    // Suppose you have a drawable resource named "ic_logo"
    Image(
        painter = painterResource(id = R.drawable.fit2081_logo_yellow),
        contentDescription = "Quiz Logo",
        modifier = Modifier.size(100.dp)
    )
}
```

```
@Composable
fun QuestionOne(
    optionOneChecked: Boolean,
    onOptionOneCheckedChange: (Boolean) -> Unit,
```

```

optionTwoChecked: Boolean,
onOptionTwoCheckedChange: (Boolean) -> Unit
) {
    // The question text
    Text(text = "Q1: Which of these statements are true about Kotlin?")

    Spacer(modifier = Modifier.height(8.dp))

    Row {
        // Option 1
        Checkbox(
            checked = optionOneChecked,
            onCheckedChange = { onOptionOneCheckedChange(it) }
        )
        Spacer(modifier = Modifier.width(4.dp))
        Text(text = "It runs on the JVM")

        Spacer(modifier = Modifier.width(24.dp))

        // Option 2
        Checkbox(
            checked = optionTwoChecked,
            onCheckedChange = { onOptionTwoCheckedChange(it) }
        )
        Spacer(modifier = Modifier.width(4.dp))
        Text(text = "It is only used for iOS")
    }
}

@Composable
fun QuestionTwo(
    answer: String,
    onAnswerChange: (String) -> Unit
) {
    Text(text = "Q2: Which company produced the most Android smartphones in 2024")
    OutlinedTextField(
        value = answer,
        onValueChange = { onAnswerChange(it) },
        label = { Text("Answer") },
        modifier = Modifier.fillMaxWidth()
    )
}

@Composable
fun SubmitButton(
    optionOneChecked: Boolean,
    optionTwoChecked: Boolean,
    questionTwoAnswer: String
) {
    val context = LocalContext.current
    Button(onClick = {

        if (optionOneChecked == q1a_answer && optionTwoChecked == q1b_answer && questionTwoAnswer == Toast.makeText(context, "Correct", Toast.LENGTH_SHORT).show()
    })
}

```

```

        } else if (optionOneChecked != q1a_answer){
            Toast.makeText(context, "Incorrect: Q1a", Toast.LENGTH_SHORT).show()
        } else if (optionTwoChecked != q1b_answer){
            Toast.makeText(context, "Incorrect: Q1b", Toast.LENGTH_SHORT).show()
        } else if (questionTwoAnswer != q2_answer){
            Toast.makeText(context, "Incorrect: Q2", Toast.LENGTH_SHORT).show()
        }

        // Calculate score (0-3)
        var score = 0
        if (optionOneChecked == q1a_answer) score++
        if (optionTwoChecked == q1b_answer) score++
        if (questionTwoAnswer == q2_answer) score++

        // Create and start intent with the score
        val intent = Intent(context, ScoreSummary::class.java)
        intent.putExtra("QUIZ_SCORE", score)
        context.startActivity(intent)
    }) {
        Text(text = "Submit")
    }
}
}

```

## ScoreSummary.kt

```

package com.fit20812025.myapplication

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Slider
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableFloatStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import com.fit20812025.myapplication.ui.theme.W2Lecturev1Theme

class ScoreSummary : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

```

```
enableEdgeToEdge()

// Get the score from the intent
val score = intent.getIntExtra("QUIZ_SCORE", 0)

setContent {
    W2Lecturev1Theme {
        Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
            Column(
                modifier = Modifier.padding(innerPadding),
                horizontalAlignment = androidx.compose.ui.Alignment.CenterHorizontally
            ){
                Spacer(modifier = Modifier.padding(16.dp))
                Greeting()
                Spacer(modifier = Modifier.padding(16.dp))
                SliderDisplay(score = score)
                Spacer(modifier = Modifier.padding(16.dp))
                ScoreDisplay(
                    score = score,
                    modifier = Modifier.padding(16.dp)
                )
            }
        }
    }
}
```

```
@Composable
fun Greeting(modifier: Modifier = Modifier) {
    Text(
        text = "Thank you for playing my quiz! Your score is:",
        modifier = modifier
    )
}
```

```
@Composable
fun SliderDisplay(score: Int, modifier: Modifier = Modifier) {
    // Convert score to float for the slider (0-3 to 0f-1f)
    val initialSliderValue = score.toFloat() / 3f
    var sliderPosition by remember { mutableFloatStateOf(initialSliderValue) }
    Column{
        Slider(
            value = sliderPosition,
            onValueChange = { sliderPosition = it },
            modifier = modifier.fillMaxWidth().padding(horizontal = 48.dp),
            enabled = false
        )
    }
}
```

```
@Composable
fun ScoreDisplay(score: Int, modifier: Modifier = Modifier) {
```

```
    Text(  
        text = "$score / 3",  
        modifier = modifier  
    )  
}
```



MONASH  
University

**FIT2081**  
**Mobile application**  
**development (MAD)**



## **Breaking down Android: APIs & Fragmentation**

**Week 2**

Delvin Varghese

## Checklist

- Android Studio installed and running
- AVD/Emulator working
- Empty application created

## Learning objectives for today

### 1. What is Android and how does it work?

- Android Software Stack
- Virtual Machines in Android

### 1. Android versions, API Levels

- Fragmentation

### 1. Android Components – The Building Blocks

- Activities, Services, Content Providers, Broadcast Receivers

# **1. What is Android and how does it work?**

## Describing Android..

**Lightweight, linux-based OS designed for touchscreen devices**

- Optimised to run within device limitations
- Open Source



**Google owns and maintains it..**

- Since 2005..
- USP: Google Play store (largest Android App store)



Linux-based

Lightweight (wrt device resources)

Designed for touchscreen mobile devices

e.g. smartphones and tablets

- Other devices include laptops/netbooks, smart TVs, wristwatches, headphones, games consoles etc.

So, optimised to operate within the typical limitations of such devices (processing, power, memory, screen size...)

So, designed to support Apps using typical hardware features of such devices.

- e.g. touchscreen, WiFi, accelerometers, gyroscopes, proximity sensors ...

Open Source (Apache License)

Free source code available for modification and redistribution

The most permissive open-source license without even the need to push modifications back into the open-source community

- So enhancements do not need to be open source
- Very business/start-up friendly
- Fragmentation???

# Who “Owns” Android?

Google

Purchased from Android Inc. in 2005 and made Open Source

They create and release new versions of Android

These can include updates to:

The software stack

- Including the API (the main focus of Android App developers)

The Linux Kernel

- which is now on a separate branch to the mainline Linux Kernel following architectural changes to the kernel by Google

Bundled Apps + Development tools + ...

Android users are significantly fragmented among several versions of Android

This creates forward and backward compatibility complications for Android App developers and users

- with iOS, this problem is much less severe

They maintain the largest Android App store

- Google Play (formerly Android Market)
- A distribution site for Apps and other media published by Google

## The Android Ecosystem

**Lots of components that work together..**

- Hardware manufacturers
- IDE creators
- Marketplaces
- App developers
- Telecoms



## The Android Ecosystem

What is the Android Ecosystem?

A set of interdependent, evolving components that together enable the creation and distribution of Android Apps

- Cooperation between these components is essential to the well-being of the Android Ecosystem
  - Many problems have arisen as a result of the lack of cooperation (see next slide)
- Compare this with the iOS Ecosystem, where nearly all components are controlled by Apple, which ensures cooperation.

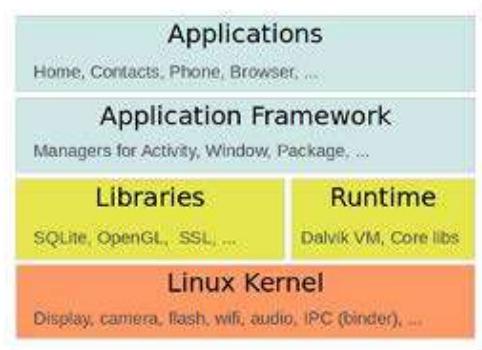
Components include hardware manufacturers, core and development software (IDEs) creators, distribution channels (marketplaces etc.), app developers and their communities, telcos, etc ...

- i.e. the OHA + other App developers + marketplaces + users

# It's an Operating System...

## Manages hardware and software..

- Apps are at the top
- Hardware at the bottom
- Multiple layers - each with different responsibilities



Source: <https://dzone.com/articles/android-software-stack-and>

It's an Operating System + ...

"An operating system (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs"

Notes

In the case of the Android OS, the top level programs are called Apps

- Some Apps come with the OS
- Google Apps + OEM and/or Telco Apps

Android includes a run-time environment for Apps

- How it manages their coexistence will be of particular interest to us

Like most OSs, Android consists of several layers of software

- Lower layers (in C/C++) perform lower-level functions for higher layers (in Java) that therefore do not have to concern themselves with lower-level details (e.g. connect to the internet)

Architected as a Linux Kernel + a Software Stack

- Linux kernel OS

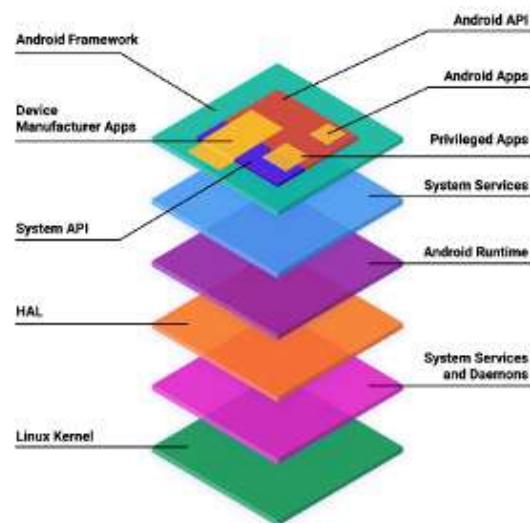
- Android Software Stack

**Next slide: Virtual machine**

## Apps run inside virtual machines..

### What is Android and how does it work?

- Android Software Stack
- Virtual Machines in Android



Source: <https://source.android.com/docs/core/architecture>

the Linux kernel provides a multitasking execution environment allowing multiple processes to execute concurrently. Therefore, it would be easy to assume that each Android application simply runs as a process directly on the Linux kernel. In actual fact, each application running on an Android device does so within its own instance of the **Dalvik** [Android Run Time (ART) since Lollipop] virtual machine [within its own Linux Kernel Process].

- Running applications in virtual machines provides several advantages. Firstly, applications are essentially sandboxed, in that they cannot detrimentally interfere (intentionally or otherwise) with the operating system, other applications or directly access the device hardware. Secondly, this enforced level of abstraction makes applications platform neutral in that they are never tied to any specific hardware.

The **Dalvik** ART virtual machine was developed by Google and relies on the underlying Linux kernel for low-level functionality. It is more efficient than the standard Java VM in terms of memory usage and specifically designed to allow multiple instances to run efficiently within the

resource constraints of a mobile device."

- Full OS wrapped in a container, like a software, running inside another machine/OS..
- Java VM (history) -> designed by Sun (the creators of Java) to run Java programs..
- So when Google acquired Android, and they realised that Java was bought by Oracle (the big software company), they decided to create an open source VM called Dalvik..
- WHY?
  - o Security: 20 years ago if you wrote a C application, it could kill another application, consume all available memory etc..
- Android application runtime must support a diverse set of devices and that applications must be sandboxed for security, performance, and reliability, a virtual machine seems like an obvious choice. Register based VM's will avoid unnecessary memory access and consume instructions efficiently. Every Android application will be running its own process, with its own instance of the Dalvik VM. Dalvik is made in a way so that a device can run multiple VMs efficiently.
- ART (Android Run Time) is the new Android runtime being used since Android L (Lollipop). Dalvik will be phased out being replaced by ART. Although it was present in Android Kit Kat as an experiment from before, ART had lot of incompatibilities with lots of applications. This move is beneficial for Google as Oracle has already many lawsuits against them related to Java patents.

<https://source.android.com/docs/core/architecture>

## **2. Android Versions & API Levels**

## Android Versions

### SDK Platform

- Incremented using the n.n.n format

### API Levels

- An integer value - unique identifier for the framework API..

The screenshot shows the "Release History" section of the Go website. At the top right is the Android logo. Below it is a table of contents for the release history. The main area lists releases from go1.0 to go1.22, each with a date and revision information.

Release Policy	Date	Revisions
go1.22.0 (released 2024-02-06)	2024-02-06	Minor revisions
go1.21.0 (released 2023-08-06)	2023-08-06	Minor revisions
go1.20 (released 2023-02-01)	2023-02-01	Minor revisions
go1.19 (released 2022-08-02)	2022-08-02	Minor revisions
go1.18 (released 2022-03-10)	2022-03-10	Minor revisions
go1.17 (released 2021-08-10)	2021-08-10	Minor revisions
go1.16 (released 2021-02-10)	2021-02-10	Minor revisions
go1.15 (released 2020-08-10)	2020-08-10	Minor revisions
go1.14 (released 2020-02-20)	2020-02-20	Minor revisions
go1.13 (released 2019-09-03)	2019-09-03	Minor revisions
go1.12 (released 2019-02-25)	2019-02-25	Minor revisions
go1.11 (released 2018-08-24)	2018-08-24	Minor revisions
go1.10 (released 2018-02-10)	2018-02-10	Older releases
go1.9 (released 2017-08-24)	2017-08-24	
go1.8 (released 2017-02-10)	2017-02-10	
go1.7 (released 2016-08-10)	2016-08-10	
go1.6 (released 2016-02-10)	2016-02-10	
go1.5 (released 2015-08-10)	2015-08-10	
go1.4 (released 2014-12-10)	2014-12-10	
go1.3 (released 2014-06-10)	2014-06-10	
go1.2 (released 2013-12-01)	2013-12-01	
go1.1 (released 2013-05-10)	2013-05-10	
go1 (released 2012-03-28)	2012-03-28	

Figure: Release History for Go. Go is an open-source programming language supported by Google

# Android Versions

### SDK Platform

The entire SDK  
Including a version of the framework API  
Incremented using n.n.n format

### API Level

"API Level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform"

### Framework API

- "The Android platform provides a framework API that applications can use to interact with the underlying Android system. The framework API consists of the following:
  - A core set of packages and classes
  - A set of XML elements and attributes for declaring

- a manifest file
- A set of XML elements and attributes for declaring and accessing resources
- A set of Intents
- A set of permissions that applications can request, as well as permission enforcements included in the system

## API Levels

**Its always a trade-off...**

- Optimised to run within device limitation
- Open Source



Source: <https://commonsware.com/>

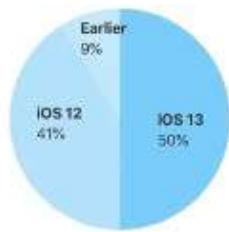
The minimum SDK version determines the lowest level of Android that your app will run on. You typically want to target as many users as possible, so you would ideally want to support everyone -- with a minimum SDK version of 1. However, that has some disadvantages, such as a lack of features, and very few people use devices that are old anymore. Your choice of minimum SDK level should be a tradeoff between the distribution of users you wish to target and the features your application will need. Click each Android Version/API level for more information.

## API Fragmentation..

**Its always a trade-off...**

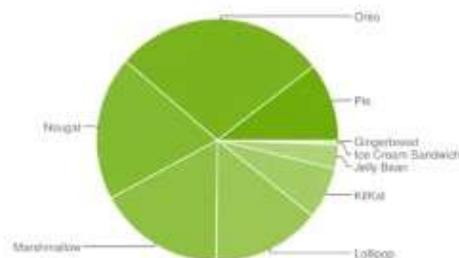
- Optimised to run within device limitations
- Open Source

50% of all devices use iOS 13.



As measured by the App Store on October 15, 2019.

?% of all devices use Android 10.



Data collected during a 7-day period ending on May 7, 2019.  
Any versions with less than 0.7% distribution are not shown.

October 17, 2019 11:18 AM

Source: <https://venturebeat.com/mobile>

# API Fragmentation

Developers must deal with fragmentation

Device Fragmentation

- From watches to giant TV screens
- What range do you want to target?
- What range is it feasible to target?
- Android supports fluid UI design + UI related resource selection based on device characteristics when the fluid design is stretched too far

## API Fragmentation

- “Compared to its chief rival mobile operating system, namely iOS, Android updates are typically slow to reach actual devices.

- For devices not under the Nexus brand, updates often arrive months from the time the given version is officially released. This is caused partly due to the extensive variation in hardware of Android devices, to which each update must be specifically tailored, as the official Google source code only runs on their flagship Nexus phone.
- Porting Android to specific hardware\* is a time- and resource-consuming process for device manufacturers, who prioritize their newest devices and often leave older ones behind. Hence, older smartphones are frequently not updated if the manufacturer decides it is not worth their time, regardless of whether the phone is capable of running the update.
- This problem is compounded when manufacturers customize Android with their own interface and apps, which must be reapplied to each new release.
- Additional delays can be introduced by wireless carriers who, after receiving updates from manufacturers, further customize and brand Android to their needs and conduct extensive testing on their networks before sending the update out to users.”

## Forward and Backward Compatibility

### Forward Compatibility

- Old apps running on new platform versions
- “Android applications are generally forward-compatible with new versions of the Android platform. Because almost all changes to the framework API are additive”
  - “...except in isolated cases where the application uses a part of the API that is later removed for some reason.”
- This is essential considering OTA (over the air) platform updates

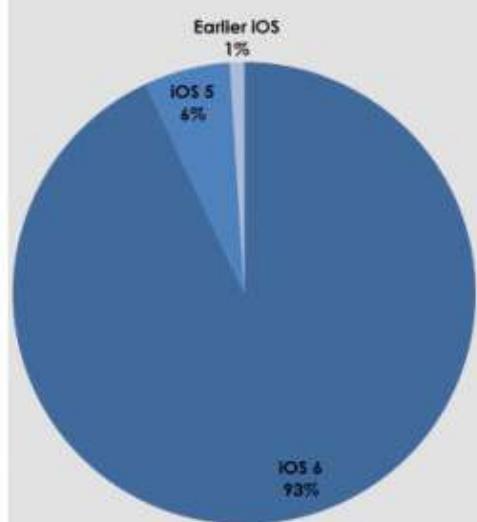
### Backward Compatibility

- New apps (using new features?) running on old platform versions
- “Android applications are not necessarily backward compatible with versions of the Android platform older than the version against which they were compiled. Each new version of the Android platform can include new framework APIs, such as those that give applications access to new platform capabilities

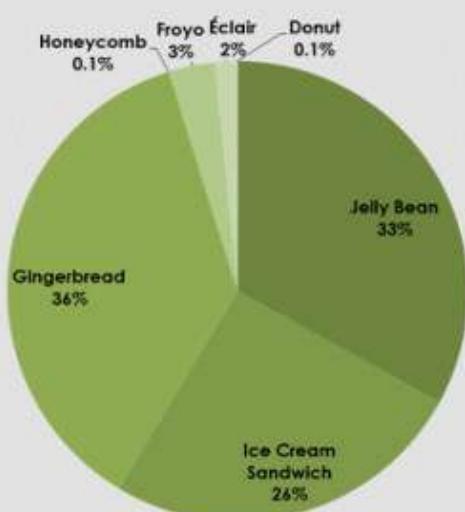
or replace existing API parts."

## Version Fragmentation by Leading Operating Systems

Apple iOS



Android



July 5, 2013 at 10:37 am |

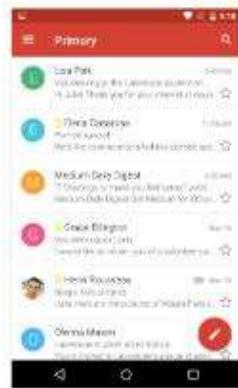
Source: <https://martech.org/how-advertisers-can-combat-mobile-fragmentation/>

### **3. Android Components - The Building Blocks**

## Activities

### Applications are made up on Activities.

- A single module..
- Each activity has a lifecycle..
- Simple definition: an activity represents a single screen with a single user interface”



## Activities

“Android applications are created by bringing together one or more components known as Activities. An activity is a single, standalone module of application functionality which usually correlates directly to a single user interface screen and its corresponding functionality”

[Activities have lifecycles as they are partially or fully hidden or killed by the OS. Developers must code callbacks (methods that are auto-fired as partial- and full-hiding and kill events occur) to respond appropriately]

“An activity represents a single screen with a user interface. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others. As such, a different app can start any one of these activities (if the email app allows it). For example, a camera app can start the activity in the email app

that composes new mail in order for the user to share a picture.

- An activity is implemented as a subclass of Activity, and you can learn more about it in the Activities developer guide."

## Manifest File

### An XML file with configurations..

- Every component declared here..
- Informs the system about app components
- Declares minimum API use, permissions



Source: <https://www.scaler.com/topics/android-manifest-file/>

## Manifest File

It's an XML file

- It details all of an Apps components , their capabilities and more
- Includes

- A declaration of all components in the application
  - Including for each, any capabilities wrt implicit inter-App intents
- In most cases if a component is not declared the system can't see it

"The primary task of the manifest is to inform the system about the app's components."

"The manifest does a number of things in addition to declaring the app's components, such as the following:

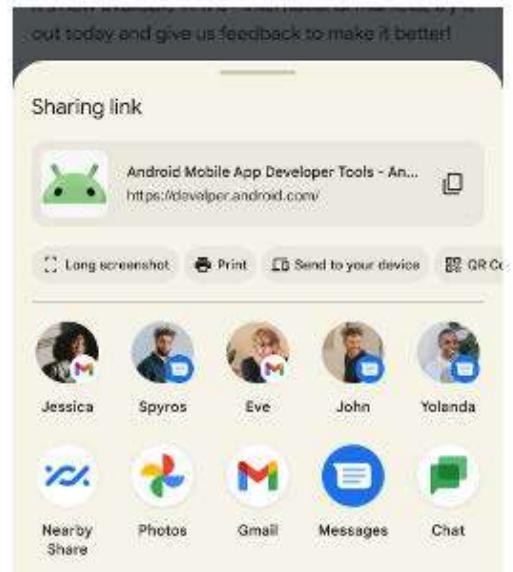
- Identifies any user permissions the app requires, such as Internet access or read-access to the user's contacts.
- Declares the minimum API Level required by the app, based on which APIs the app uses.
- Declares hardware and software features used or required by

- the app, such as a camera, bluetooth services, or a multitouch screen.
- Declares API libraries the app needs to be linked against (other than the Android framework APIs), such as the Google Maps library.”

## Activating Components - Intents

### Intent = asynchronous message

- Useful for activating activities, services or broadcast receivers.
- **Activity Intents** enable one activity to launch another activity
- **Broadcast Intent** - system-wide intent sent to all apps that have registered an “interested” broadcast receiver.
- **Inter-App Activation** - allowing another app to start your app’s component.



Source: <https://developer.android.com/training/sharing/send>

# Activating Components - Intents

## Intents

- “Three of the four component types—activities, services, and broadcast receivers—are activated by an asynchronous message called an intent. Intents bind individual components to each other at runtime (you can think of them as the messengers that request an action from other components), whether the component belongs to your app or another.”
- An intent is created with an Intent object, which defines a message to activate either a specific component or a specific type of component—an intent can be either explicit or implicit, respectively.”

## Activity Intents

- “Intents are the mechanism by which one activity is able to launch another [activity or service] and implement the flow through the activities that make up an application. Intents consist of a description of the operation to be performed and, optionally, the data on which it is to be performed.
- Intents can be explicit, in that they request the launch of a specific activity by referencing the activity by class name, or implicit by stating either the type of action to be performed or providing data of a specific type on which the action is to be performed. In the case of implicit intents, the Android runtime will select the activity to launch that most closely matches the criteria specified by the Intent using a process referred to as Intent Resolution.”

## Broadcast Intents

- Another type of Intent, the Broadcast Intent, is a system wide intent that is sent out to all applications that have registered an “interested” Broadcast Receiver. The Android system, for example, will typically send out Broadcast Intents to indicate changes in device status such as the completion of system start up, connection of an external power source to the device or the screen being turned on or off.

## Inter-App Activation

“A unique aspect of the Android system design is that any app can start another app’s component.”

“When the system starts a component, it starts the process for that app (if it’s not already running) and instantiates the classes needed for the component.”

“Because the system runs each app in a separate process with file permissions that restrict access to other apps, your app cannot directly activate a component from another app. The Android system, however, can. So, to activate a component in another app, you must deliver a message to the system that specifies your intent to start a particular component. The system then activates the component for you.”

- An app must give its permission for any of its components to be activated by an external intent
- **For example, if you build a social app that can share**

**messages or photos with the user's friends, support the ACTION\_SEND intent. Then, when users initiate a "share" action from another app, your app appears as an option in the chooser dialog (also known as the disambiguation dialog), as shown in figure 1.**

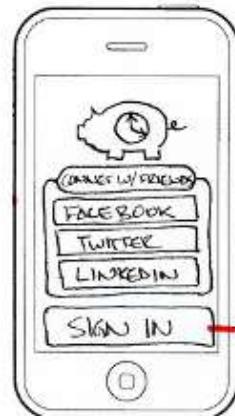
Content Providers (not activated by intents)

- “Unlike activities, services, and broadcast receivers, content providers are not activated by intents. Rather, they are activated when targeted by a request from a ContentResolver. The content resolver handles all direct transactions with the content provider so that the component that's performing transactions with the provider doesn't need to and instead calls methods on the ContentResolver object. This leaves a layer of abstraction between the content provider and the component requesting information (for security).”

## Designing With Compose

### Simplifying Jetpack Compose

1. Start with a Sketch
2. Group UI elements into composables
3. Choose Layout Structures
4. Beautify
5. Pseudocode
6. Implement + Manage State



[Photo](#) by Unknown Author is licensed under [CC BY-SA-NC](#)

- Designing a screen in Jetpack Compose is about **planning, structuring, and refining**. Sketch first, break your UI into modular composable functions, choose appropriate layouts, and manage state so Compose can do its magic. Think of it like building a house: you

have blueprints (layout planning), construction (implementation with Rows, Columns, Boxes), décor (MaterialTheme), and final checks (preview, testing on devices).

- **Key Takeaways:**
- Start with a sketch or wireframe.
- Use composables to group logical screen elements.
- Leverage layout containers (Row, Column, Box).
- Keep spacing and alignment in mind.
- Adopt Material Design components for a consistent

look.

- Manage data changes with proper state handling.
- Preview and test on different devices for responsiveness.
- With a systematic approach (and a bit of creativity), your Jetpack Compose UI can be both **functional** and **beautiful**.
- **Start with a Sketch**
- Just like you wouldn't build a house without a rough floor plan:
  1. **Draw a rough outline** on paper or a whiteboard.

- 2. Identify major rooms**  
(headers, main content, footers).
  - 3. Mark where you'll place furniture** (buttons, text fields, images).
- **Key Takeaway:** A quick sketch helps solidify your design goals, focus on structure, and minimize guesswork once you start coding.
  - **Group Your UI Elements into Composables**
  - In a house, each room has a purpose—kitchen, bedroom, living room. In Compose, each

section of your UI can be a **composable function**:

- **HeaderSection()** might display the app title or top navigation.
- **MainContent()** could hold your primary data or user inputs.
- **FooterButtons()** might include confirm or cancel actions.
- **Why do this?**
- Encourages **modular design** (it's easier to work on separate pieces).
- Makes your code more **readable** and **maintainable**.

- **Choose Your Layout Structures**
- Once you've decided on your rooms, you need to place them. Jetpack Compose offers several layout “blueprints”:
- **Row**: items in a horizontal line (like putting kitchen cabinets side by side).
- **Column**: items in a vertical stack (like stacking floors).
- **Box**: overlays or layers items on top of each other (like placing a rug, then furniture).
- **Best Practices**:
- Use Modifier.padding() for spacing around items.
- Leverage alignment

parameters (e.g.,  
horizontalAlignment =  
Alignment.CenterHorizontally)  
to fine-tune positioning.

- **Add Spacing and Alignment**
- Think of spacing like the corridors and walkways in a house—you need them for a comfortable layout:
- Use Arrangement (e.g.,  
Arrangement.SpaceBetween)  
to specify how items spread  
out in a Row or Column.
- Use Modifier.padding() or  
Modifier.size() to ensure  
elements aren't crammed  
together.

- This step ensures your “house” is livable and flows well visually.
- **Use Material Design Components**
- Now it’s time to pick out the furniture and decor:
- **Buttons, Text, Card, etc.** come from the Jetpack Compose Material library.
- **MaterialTheme** gives you **colors, typography, and shapes** aligned with modern Android design principles.
- **Real-World Analogy:**
- If your “house” were a real property, these Material

components are like reliable, pre-made furniture that's tested for safety and comfort.

- **Manage State**
- A home needs utilities and wiring. Similarly, your app needs a way to handle data changes:
- Store dynamic values (e.g., user input, network data) in a `ViewModel` or using `remember/mutableStateOf`.
- Whenever the “state” changes, the UI **automatically** re-draws (“recomposes”) to stay in

sync.

- **Why it Matters:**
- Simplifies data flow. You don't have to manually call "refresh" methods—Compose handles it.

## Let's make an app!

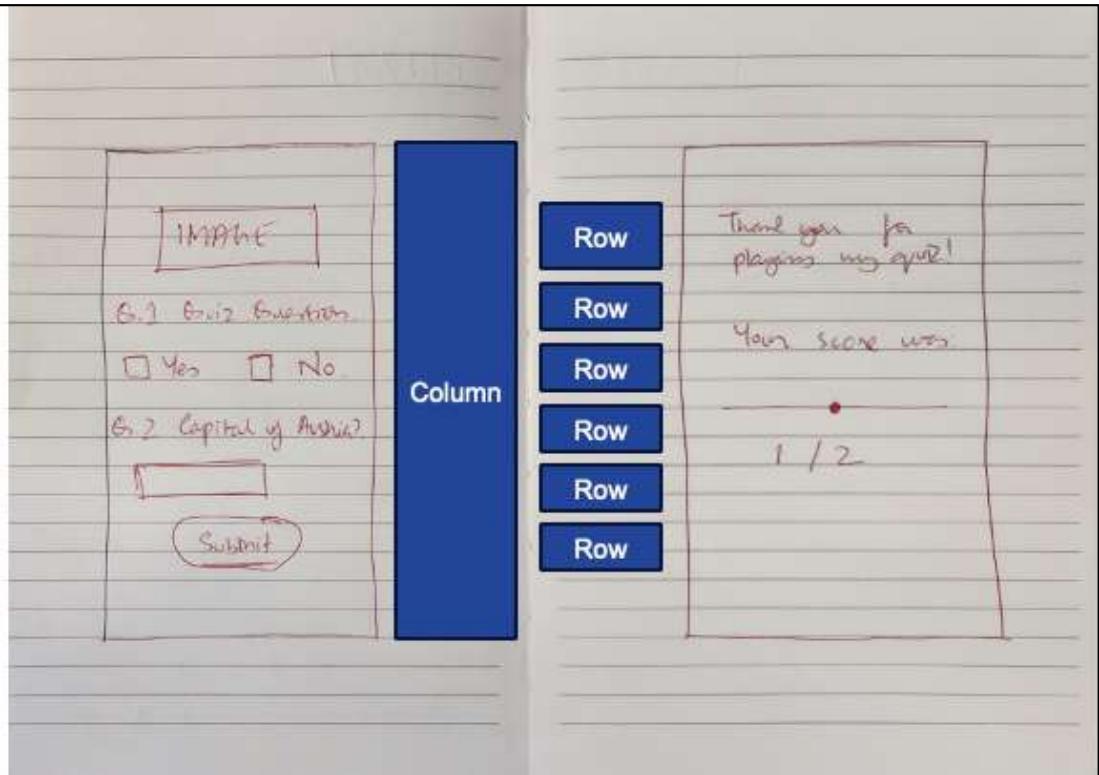


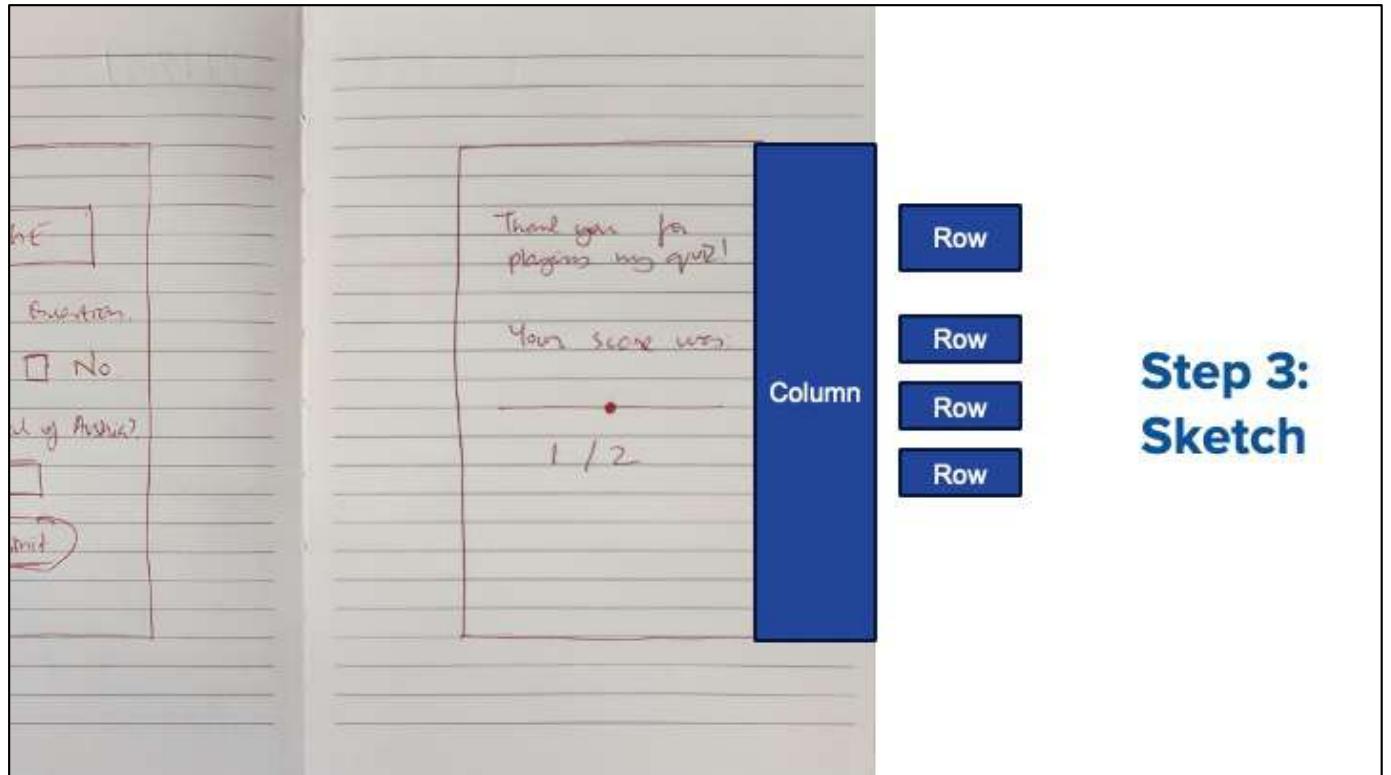
[This Photo](#) by Unknown Author is licensed under [CC BY-SA-NC](#)

## Step 1: Sketch

<p>IMAGE</p> <p>Q.1 Quiz Question</p> <p><input type="checkbox"/> Yes   <input type="checkbox"/> No</p> <p>Q.2 Capital of Austria?</p> <p><input type="text"/></p> <p>Submit</p>	<p>Thank you for playing my quiz!</p> <p>Your score was</p> <p>1 / 2</p>
--	--

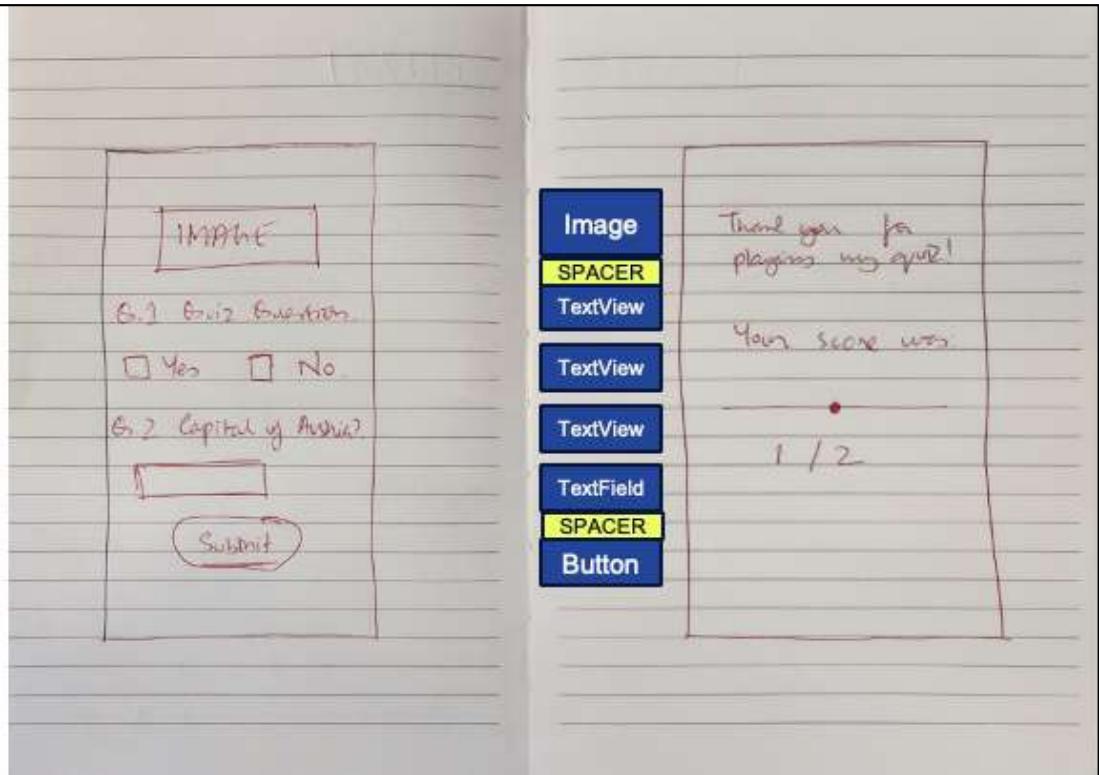
### Step 3: Sketch



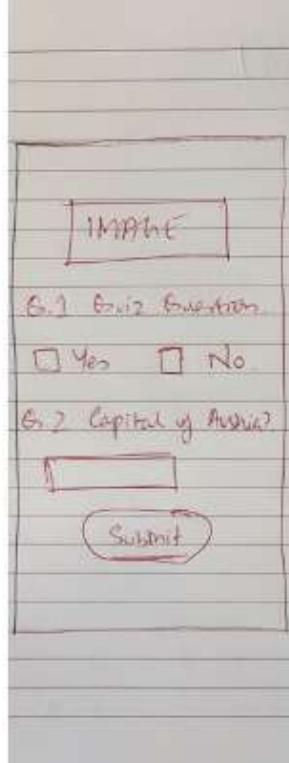


### Step 3: Sketch

## Step 4: Beautify



## Step 5: Pseudo-code



```
@Composable  
fun QuizScreen(){  
    Column(){  
        QuizHeader()  
        Spacer()  
        QuestionOne()  
        Spacer()  
        QuestionTwo()  
        Spacer()  
        SubmitButton()  
    }  
}
```

## Understanding Syntax

In Kotlin, **Unit** is a special type that represents "no value" or "void".

It's similar to void in Java, but with an important distinction: Unit is an actual type with a single instance (also called Unit), while void in Java isn't a type you can use as a value.

```
@Composable
fun QuestionOne(
    optionOneChecked: Boolean,
    onOptionOneCheckedChange: (Boolean) -> Unit,
    optionTwoChecked: Boolean,
    onOptionTwoCheckedChange: (Boolean) -> Unit
) {
    // ...
}
```

The notation (Boolean) -> Unit describes a function type with:

- A parameter of type Boolean
- A return type of Unit

```
Checkbox(
    checked = optionOneChecked,
    onCheckedChange = { onOptionOneCheckedChange(it) }
)
```

- Here, `onOptionOneCheckedChange` is a parameter that accepts a function. This function:
  1. Takes one Boolean parameter
  2. Returns Unit (meaning it doesn't return any meaningful value)

- This is Kotlin's way of defining a callback function that will be called when something happens (in this case, when the checkbox state changes).
- When the checkbox is clicked:
  1. The onCheckedChange event fires
  2. The lambda {  
onOptionOneCheckedChange  
(it) } is executed
  3. This calls your passed-in function with the new boolean value
  4. The parent component (which provided this function) can

react accordingly

## Understanding Syntax

In Kotlin, Unit is a special type that represents "no value"

```
or "void"
QuestionOne(
    optionOneChecked = optionOneChecked,
    onOptionOneCheckedChange = { newValue ->
        optionOneChecked = newValue
    },
    // ...
)
```

```
@Composable
fun QuestionOne(
    optionOneChecked: Boolean,
    onOptionOneCheckedChange: (Boolean) -> Unit,
    optionTwoChecked: Boolean,
    onOptionTwoCheckedChange: (Boolean) -> Unit
) {
    // ...
}
```

with:

- A parameter of type Boolean
- A return type of Unit

```
Checkbox(
    checked = optionOneChecked,
    onCheckedChange = { onOptionOneCheckedChange(it) }
)
```

- The lambda { newValue -> optionOneChecked = newValue } is a function that:
  1. Takes a Boolean parameter (named newValue)
  2. Updates the state variable optionOneChecked
  3. Implicitly returns Unit (since

# assignment expressions in Kotlin return Unit)

- **Why Unit Instead of Void?**
- You might wonder why Kotlin uses Unit instead of void.  
There are several reasons:
  1. **Consistency:** In Kotlin, everything is an expression and has a type. Unit allows functions that don't return a meaningful value to still have a consistent type.
  2. **Generics:** Unit can be used in generic contexts where a concrete type is required.
  3. **Functional programming:** In functional programming

(which Kotlin supports well),  
having a single value to  
represent "no value" is more  
elegant and useful than  
having no value at all.

**THE END\_**

## FIT2081 – Week 2 Lecture Notes: Breaking Down Android – APIs & Fragmentation

### Week 2

 Topic: Android Software Stack, API Levels, and Android Components

 Lecturer: Delvin Varghese

---

## Learning Objectives for Today

By the end of this lecture, you should understand:

1. **What is Android and how does it work?**
    - **Android Software Stack:** How Android is structured and why it uses a layered architecture.
    - **Virtual Machines in Android:** The role of Dalvik and ART in app execution.
  2. **Android Versions & API Levels**
    - **Understanding Fragmentation:** Why Android devices run different OS versions and how this affects development.
  3. **Android Components – The Building Blocks**
    - **Activities, Services, Content Providers, and Broadcast Receivers:** Core components of Android apps and how they interact.
- 

## Pre-Class Checklist

Before continuing with today's content, ensure that:

- ✓ **Android Studio** is installed and running.
  - ✓ **AVD/Emulator** is set up and functional.
  - ✓ An empty **Android application** has been created successfully.
- 

## Section 1: What is Android and How Does It Work?

### ♦ **Android Overview**

Android is a **Linux-based** operating system designed primarily for **touchscreen devices**. It has been optimized to:

- **Operate efficiently** within hardware limitations like limited processing power, battery life, and memory.
- **Support common hardware features** such as **touchscreen gestures, WiFi, sensors (accelerometer, gyroscope, proximity sensors)**.
- Be **open-source**, which allows manufacturers to modify and distribute it under the **Apache License**.

#### ◆ Who Owns Android?

- **Google** acquired Android from Android Inc. in **2005**.
- Google is responsible for:
  - **Developing and releasing new Android versions.**
  - **Updating the API framework** (used by developers to build apps).
  - **Maintaining the Google Play Store**, the largest Android app marketplace.

#### ◆ The Android Ecosystem

- Android is an **open ecosystem**, meaning it consists of **multiple stakeholders**:
  - **Device manufacturers (Samsung, Google, OnePlus, etc.)**
  - **App developers** who create apps for the Play Store.
  - **Network carriers (Optus, Telstra, etc.)** who sometimes modify Android for their networks.
  - **Development tools & software** like Android Studio and Jetpack libraries.

#### Comparison with iOS:

Unlike Apple, which controls **both the hardware and software** in iOS, Android's **openness results in fragmentation**—meaning different devices run different versions of Android at any given time.

---

## Section 2: Android Software Stack & Virtual Machines

#### ◆ Understanding the Android Software Stack

Android is designed as a **multi-layered software stack** to efficiently manage hardware and applications. The key layers include:

1. **Linux Kernel** – The foundation of Android, responsible for hardware interaction, memory management, and process scheduling.
2. **Hardware Abstraction Layer (HAL)** – Provides an interface between hardware drivers and higher-level APIs.
3. **Android Runtime (ART/Dalvik)** – The execution environment for apps.
4. **Application Framework** – Exposes APIs for developers to build apps.
5. **Applications** – User-installed apps that interact with the OS.

#### ◆ Virtual Machines & Dalvik/ART

Instead of running apps **directly on the Linux kernel**, Android runs them inside **virtual machines**.

- **Dalvik VM (Pre-Android 5.0 Lollipop)**
  - Created by Google to run **Java-based Android apps**.
  - Designed to be **lightweight** and optimized for mobile hardware.
- **Android Runtime (ART) (Android 5.0 onwards)**
  - **Improves performance** by compiling apps ahead of time (AOT) instead of just-in-time (JIT).
  - **Reduces memory consumption** and speeds up execution.
  - **Why the shift?** Dalvik had **performance limitations** and legal issues with Oracle (which owns Java).

#### 📌 Security & Performance Considerations:

- Virtual machines **sandbox apps**, preventing them from accessing each other's data.
  - **Each app runs in its own VM**, avoiding crashes caused by other apps.
- 

## 📌 Section 3: Android Versions & API Levels

#### ◆ Android SDK Platform & API Levels

Each version of Android comes with:

- **A unique SDK (Software Development Kit)** for developers.
- **API Level** – A number assigned to each framework revision.
- **Backward and Forward Compatibility:**
  - Developers **set a minimum SDK level** (oldest version their app supports).
  - Choosing an **older minimum SDK** supports more devices **but limits features**.

#### ◆ API Fragmentation & Its Challenges

Unlike iOS, which updates across all Apple devices at once, Android updates **reach devices at different times** due to:

1. **Device manufacturers modifying Android** before releasing it.
2. **Carriers testing updates before allowing them**.
3. **Older devices not receiving updates** due to hardware limitations.

#### 📌 Impact on Developers:

- **Apps may behave differently on different devices**.
- **New features might not work on older versions**.

- Testing across multiple API levels is necessary.

 **Example:**

- A Samsung phone (3 years old) may still run Android 11, while a Google Pixel phone may have Android 14.
  - Developers need to handle UI inconsistencies, security updates, and API deprecations.
- 

 **Section 4: Android Components – The Building Blocks**

◆ **Key Android Components**

1. **Activities** – Represent a single screen of an app.
  - Example: The main screen of WhatsApp that shows your chats.
  - Activities **have lifecycles** (onCreate, onPause, onResume, onDestroy).
2. **Services** – Perform background operations (without UI).
  - Example: Spotify playing music in the background.
3. **Broadcast Receivers** – Respond to system-wide events.
  - Example: An app responding to "battery low" notifications.
4. **Content Providers** – Allow apps to share data securely.
  - Example: The Contacts app providing access to phone numbers for messaging apps.

◆ **Android Manifest File**

Every Android app must declare:

- All components (Activities, Services, etc.).
- Permissions required (e.g., Internet access, Camera).
- Minimum and target API levels.

◆ **Intents & Inter-App Communication**

- **Explicit Intent:** Calls a specific component.
    - Example: Opening a new screen within the same app.
  - **Implicit Intent:** Lets the system determine the best app to handle an action.
    - Example: Choosing Gmail or WhatsApp to share an image.
-

## Section 5: Designing a UI in Jetpack Compose

### ◆ Jetpack Compose Approach

Jetpack Compose **simplifies UI development** by using a **declarative programming model**.

#### 1. Start with a Sketch

- Draw rough wireframes before coding.

#### 2. Use Composables

- Break UI into reusable **Composable functions**.

#### 3. Choose Layout Structures

- **Row:** Horizontal elements (e.g., a navbar).
- **Column:** Vertical elements (e.g., form fields).
- **Box:** Overlapping elements (e.g., a button over an image).

#### 4. Apply Material Design

- Use **Material Components** for a consistent UI.

#### 5. Manage State Efficiently

- Use `remember { mutableStateOf() }` to track UI changes dynamically.

## Example: A Quiz App

- Layout: `Column` for vertical arrangement.
  - Components: `TextView`, `Button`, `TextField`.
- 

## Section 6: Key Takeaways

- Android is **fragmented**, requiring careful API selection.
  - Apps run inside **ART (Android Runtime) virtual machines**.
  - **Jetpack Compose** enables modern UI development.
  - Android apps consist of **Activities, Services, Broadcast Receivers, and Content Providers**.
- 

## Next Steps

- ✓ Complete **Lab 2**: Hands-on practice with API levels and UI design.

---

## Lecture Recording & Notes

-  Will be available on **Ed Forum**
-  Post Questions on Ed Forum 

# Lab: Login | Navigation | Multi-Screen Apps

---

## Overview

### Intro

In this lab, we will build a basic **login screen** using **Jetpack Compose**, where a user can enter their **username and password** to authenticate. If the credentials are correct, the app will display a success message. If the login fails, an error message will appear using a **Toast notification**.

This lab will introduce several **core Android development concepts**:

**User Input Handling** – Creating interactive text fields for username and password entry.

**State Management** – Using Jetpack Compose's `remember` and `mutableStateOf` to store user input dynamically.

**Conditional UI Updates** – Displaying success or error messages based on login attempts.

**Navigation Between Screens** – Implementing a second screen (`Dashboard Activity`) and linking it to the login process.

**UI Structuring with Scaffold & Surface** – Understanding how to organize Compose UI components properly.

By the end of this lab, you will have created a **fully functional login screen** with simple **navigation** and **error handling**, giving you a strong foundation in Android development with Jetpack Compose. Let's get started!

## Resources

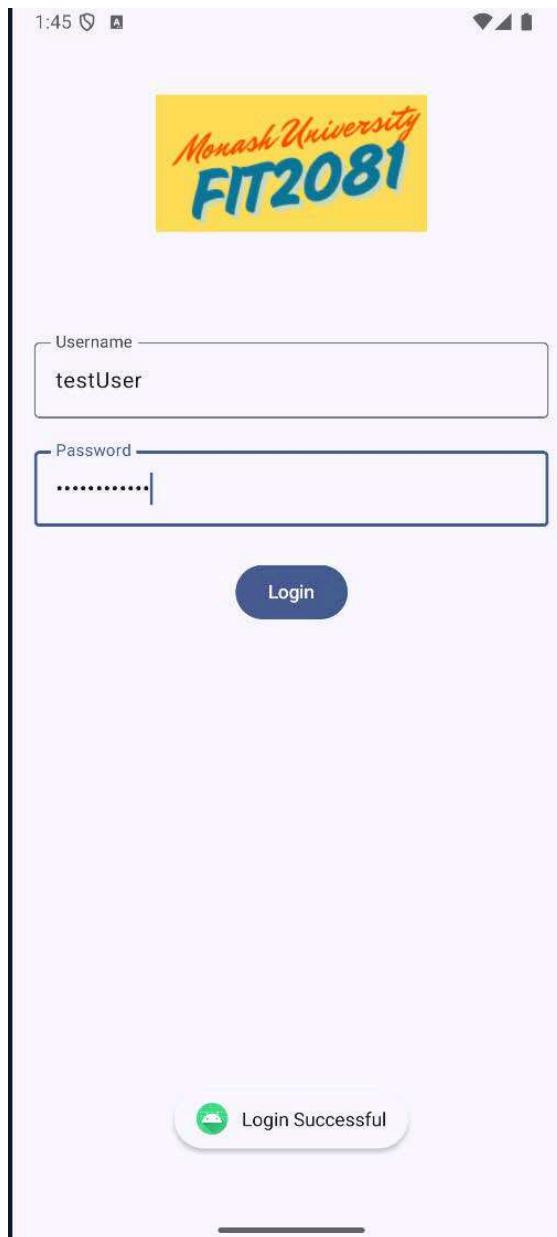
Feel free to use your own images, but if you'd like to use the same images as used in this Lab, download the images here:



[W2-FIT2081-Image-Resources.zip](#)

## Screenshots of the app that we will build.

When the user enters the correct username/password - we will display a success message at the bottom.



Gracefully handling invalid logins by using error message short notifications (Toast displayed at bottom of screen).



Username

invalidUser

Password

.....

A blue rounded rectangular button containing the word "Login" in white text.



Incorrect Credentials

# 1. Develop Login Screen

Now, let's get serious and develop a login screen where the user enters their username and password for authentication.

**i** Note: We are not going to store our users in a database at this stage. Instead, we will compare the values to pre-defined (hard-coded) values for the sake of simplicity.

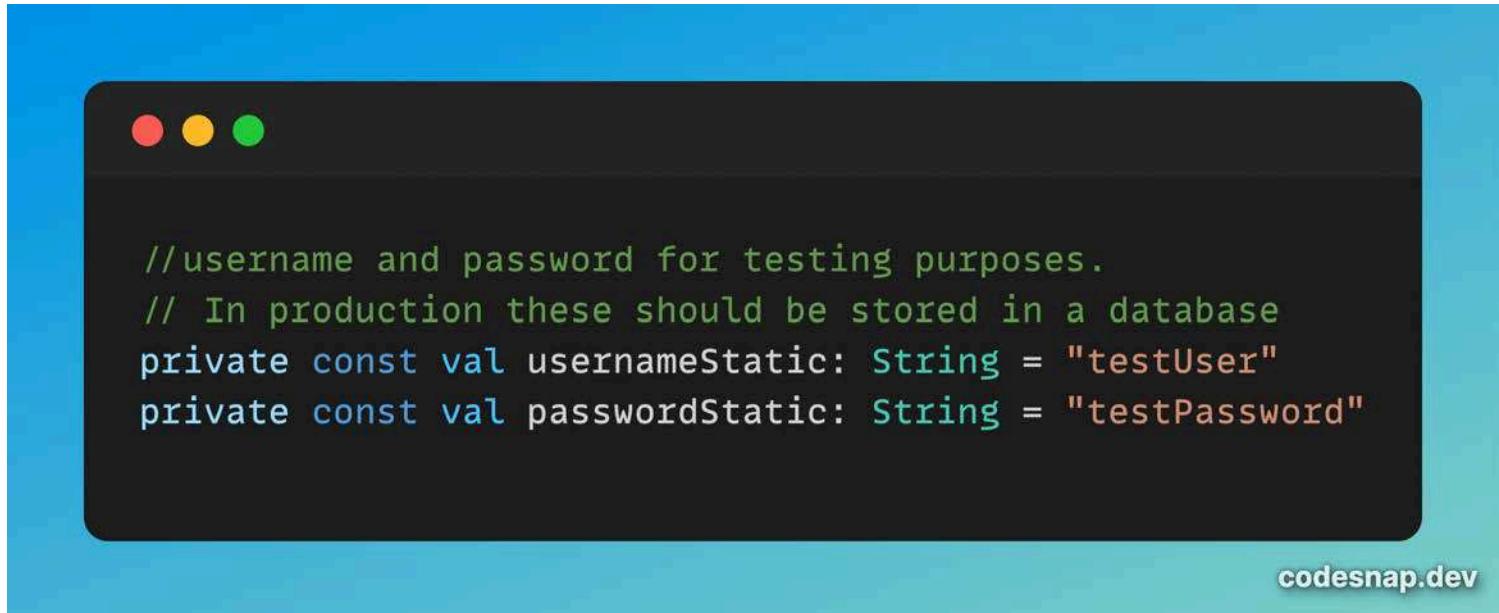
## Step 1a Create a new project

1. Go and create a project
2. Select Empty Activity
3. Call it 'LabWeek2'.

**i** If you are not sure how to do these steps, go back to Week 1 Labs..

## Step 1b: Add some strings to store the username and password.

Remember in real-world (i.e. production) apps - this is a terrible idea to hard-code user details in the code. But since we are only doing a test on our own laptops/devices this is fine for testing purposes. In real-world apps, they should be stored securely in a database.



```
//username and password for testing purposes.  
// In production these should be stored in a database  
private const val usernameStatic: String = "testUser"  
private const val passwordStatic: String = "testPassword"
```

codesnap.dev

Have a think - where should we add these variables? See answer below after you have tried first yourself.

► Expand

## Step 1c: Create a function to handle LoginScreen feature on the app.

This function will do the following:

- Add a nice image to the top of the screen
- Create a username field, where the user can enter their username
- Create a password field
- Create a button which when clicked, tests the username/password against our hard-coded values (from above)
- Show a short on-screen notification (called a Toast)

Let's do this one step at a time.. First let's create the basic shell.

You'll see here there is a **Surface** element being used. We will create all our UI elements inside **Surface**.

- Surface is a simple container **for individual UI elements**.
- It helps apply **background color, elevation, and shape**.
- Think of it as a **Material Design wrapper** for UI components.

```
@Preview(showBackground = true)
@Composable
fun LoginScreen(modifier: Modifier = Modifier) {
    var username by remember { mutableStateOf("") }
    var password by remember { mutableStateOf("") }

    val context = LocalContext.current

    Surface(
        modifier = modifier.fillMaxSize(),
        color = MaterialTheme.colorScheme.background
    ) {
        Column(
        ) {

//            We will insert an Image here
//            userName
//            Password
//            Button

        }
    }
}
```

The function implements a login screen using Jetpack Compose. It uses the Surface composable

function to draw the background for the screen. It also uses the remember function to remember user input within the username and password variables.

The login process is triggered when the user interacts with the Login button. The application checks if the provided username and password match the static credentials (usernameStatic and passwordStatic). A Toast notification displays a success message if the credentials match. Otherwise, an error message indicates that the credentials are incorrect.

**Snapshot:** OK, so after doing these steps this is what your code should might look like so far. Note: in the following code, we've also added some spacers for aesthetic purposes.



```
/*
 * Composable function representing the login screen.
 * It includes fields for username and password, along with a login button.
 */
@Composable
fun LoginScreen(modifier: Modifier = Modifier) {
    // State to hold the username input. It's mutable and tied to the composable lifecycle.
    var username by remember { mutableStateOf("") }
    // State to hold the password input. It's mutable and tied to the composable lifecycle.
    var password by remember { mutableStateOf("") }

    //This context will be used to show a toast message
    val context = LocalContext.current

    // Surface acts as a container that fills the available space and applies a background color.
    Surface(
        modifier = Modifier.fillMaxSize(),
        color = MaterialTheme.colorScheme.background
    ) {
        Column(
            modifier = Modifier
                .fillMaxSize()
                .padding(16.dp),
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.Top
        ) {
            // Add image at the top
            androidx.compose.foundation.Image(
                painter = painterResource(id = R.drawable.fit2081_logo_yellow),
                contentDescription = "Fit Logo",
                modifier = Modifier.size(200.dp) // Adjust size as needed
            )
            // add space between logo and username
            Spacer(modifier = Modifier.height(24.dp))

            OutlinedTextField( //username input field
                value = username,
                onValueChange = { username = it },
                label = { Text(text = "Username") },
                modifier = Modifier.fillMaxWidth()
            )
            // add space between username and password
            Spacer(modifier = Modifier.height(16.dp))
            OutlinedTextField( //password input field
                value = password,
                onValueChange = { password = it },
                label = { Text(text = "Password") },
                modifier = Modifier.fillMaxWidth()
            )
        }
    }
}
```

```
// when the password is changed it will reflect in here

onValueChange = { password = it },
label = { Text("Password") },
visualTransformation = PasswordVisualTransformation(),
keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Password),
modifier = Modifier.fillMaxWidth()
)
Spacer(modifier = Modifier.height(24.dp))
Button(
    onClick = {
        //check if the username and password are correct
        if (username == usernameStatic && password == passwordStatic) {
            //if correct show a toast message
            Toast.makeText(context, "Login Successful", Toast.LENGTH_LONG).show()
        } else {
            //if incorrect show a toast message
            Toast.makeText(context, "Incorrect Credentials", Toast.LENGTH_LONG).show()
        }
    }
) {
    Text("Login")
}
}
}
```

codesnap.dev

## Step 1d: Call LoginScreen inside setContent{} of MainActivity class.

**i** Remember, that setContent{} is where anything you'd like to be displayed in that screen's UI i.e. using composable functions.

Update the main activity class to call a function called 'LoginScreen'.

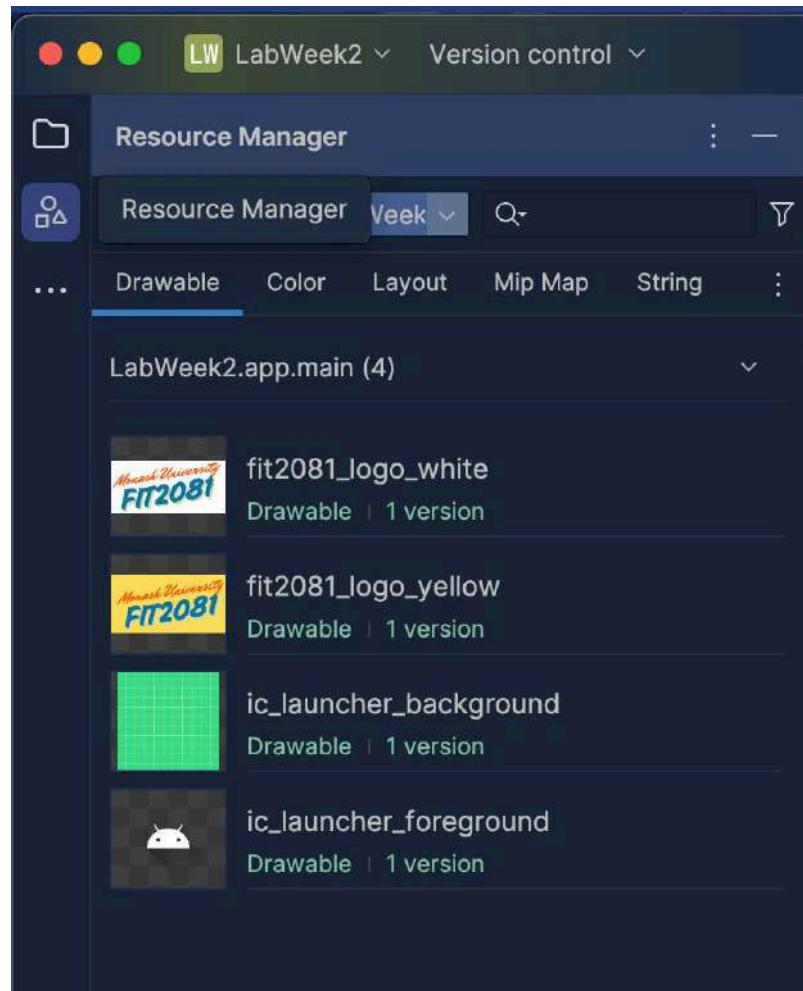
```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // The EdgeToEdge statement is used to make the app edge to edge
        enableEdgeToEdge()
        setContent {
            LabWeek2Theme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    LoginScreen(modifier = Modifier.padding(innerPadding))
                }
            }
        }
    }
}
```

codesnap.dev

As you can see in the code above, I have added an image to the top of the screen using 'Image'.

**i** If you want to use the same image as below, download the Resources zip file in the [Overview section](#).

The image is stored in the application resources, which can be reached from the left pane. Then, you only need to drag and drop your images and access them later in your code through 'R.drawable.image\_name'.



Run your app, and you should get the following:



Username

testUser

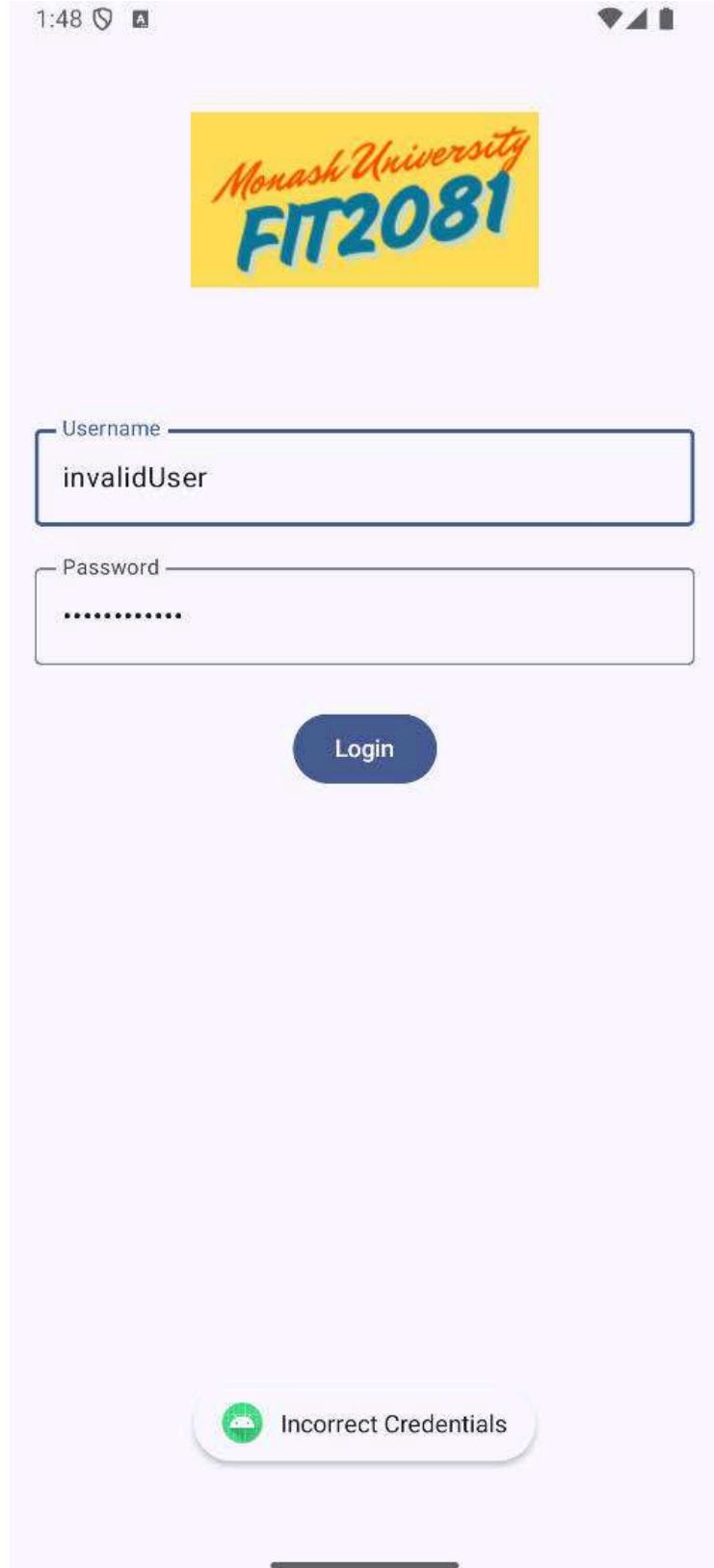
Password

.....|

Login



Login Successful



More details on how to Customize your images in Android:

<https://developer.android.com/develop/ui/compose/graphics/images/customize>

KeyboardOptions:

<https://developer.android.com/reference/kotlin/androidx/compose/foundation/text/KeyboardOption>

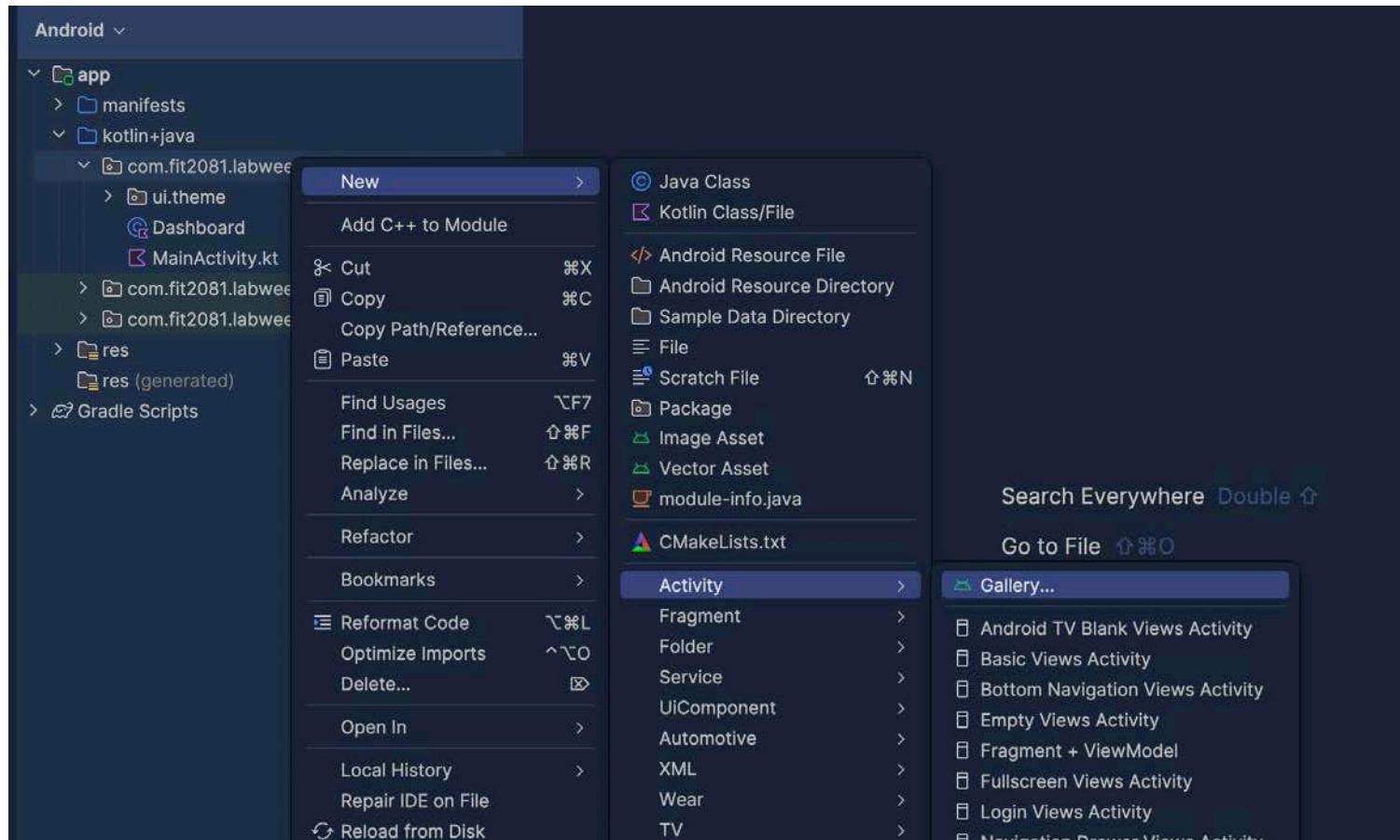
S

## 2. Basic Navigation

Now, let's learn how to navigate to another activity.

Firstly, add a second activity to your project named 'Dashboard'.

To add a new activity, right-click your package → **New** → **Activity** → **Gallery... Empty Activity**



Update the content of the dashboard activity to show a label.

```
class Dashboard : ComponentActivity() {
    /**
     * Called when the activity is starting.
     */
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            // Apply the theme to the whole composable content
            LabWeek2Theme {
                // Scaffold provides a basic layout structure for Material Design apps
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    // Column is used for vertical arrangement of items
                    Column(
                        modifier = Modifier.fillMaxSize().padding(innerPadding),
                        verticalArrangement = Arrangement.Center,
                        horizontalAlignment = Alignment.CenterHorizontally
                    ) {
                        // Displaying the text "Dashboard"
                        Text(
                            text = "Dashboard",
                            style = TextStyle(fontSize = 40.sp)
                        )
                    }
                }
            }
        }
    }
}
```

codesnap.dev

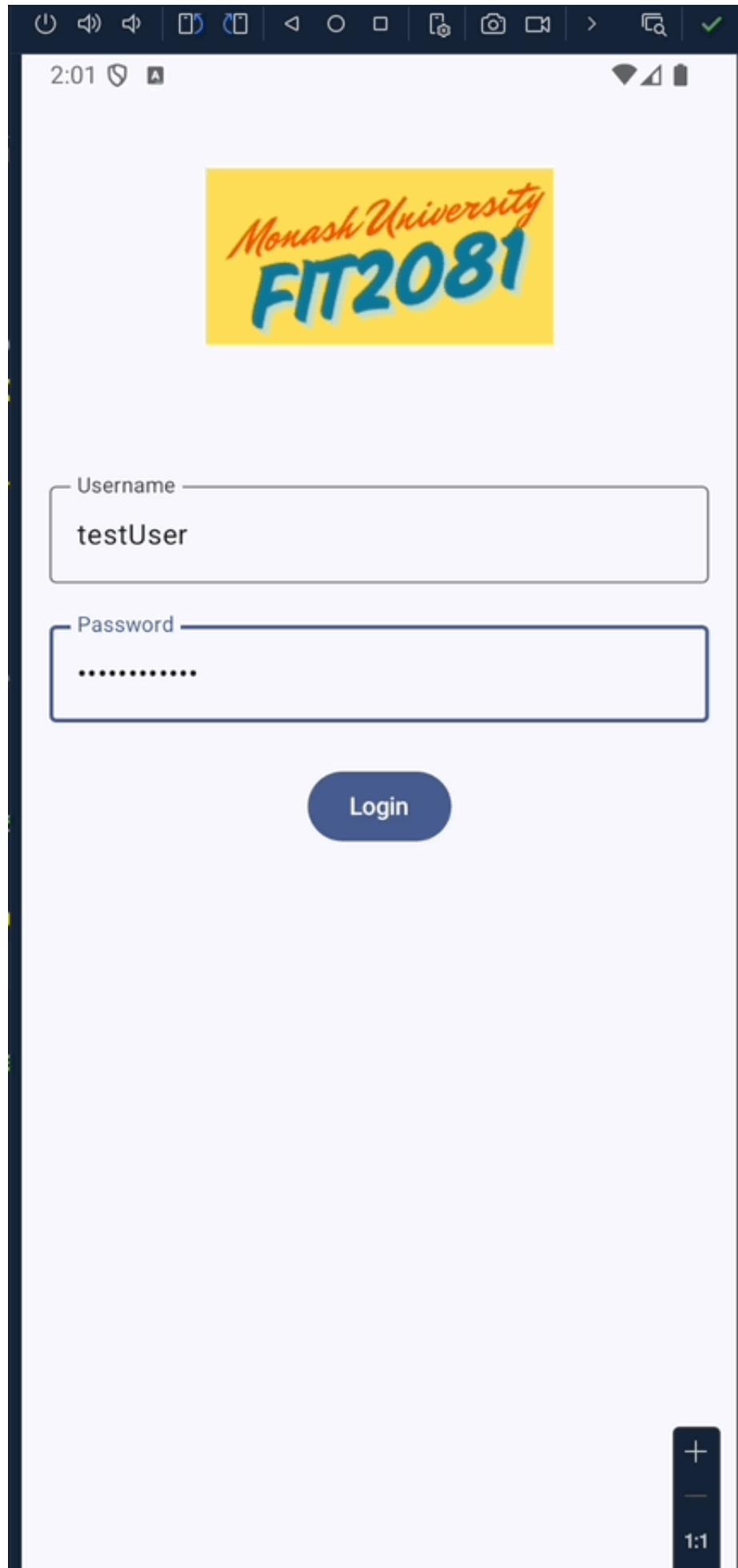
Now, update the if statement in the MainActivity.kt to start the Dashboard activity when the login operation is successful.

```
Button(  
    onClick = {  
        //check if the username and password are correct  
        if (username == usernameStatic && password == passwordStatic) {  
            //if correct show a toast message  
            Toast.makeText(context, "Login Successful", Toast.LENGTH_LONG).show()  
            //start the dashboard activity  
            context.startActivity(Intent(context, Dashboard::class.java))  
        } else {  
            //if incorrect show a toast message  
            Toast.makeText(context, "Incorrect Credentials", Toast.LENGTH_LONG).show()  
        }  
    }  
){  
    Text("Login")  
}
```

codesnap.dev

The statement `startActivity` takes as input an instance of class `Intent` that shows your intention to start a new activity.

Run your app and enjoy.



**Great! You have just created your first multi-screen app!**

► Expand

### 3. Bottom Menu Bar

#### Step 1: Understand What We Are Building

##### What is a Bottom Navigation Bar?

- A **Bottom Navigation Bar** is a menu that appears at the bottom of your app.
- It allows you to navigate to other activities or tasks. It could contain icons, buttons, or floating action buttons.
- It helps users **navigate** between different parts of the app.

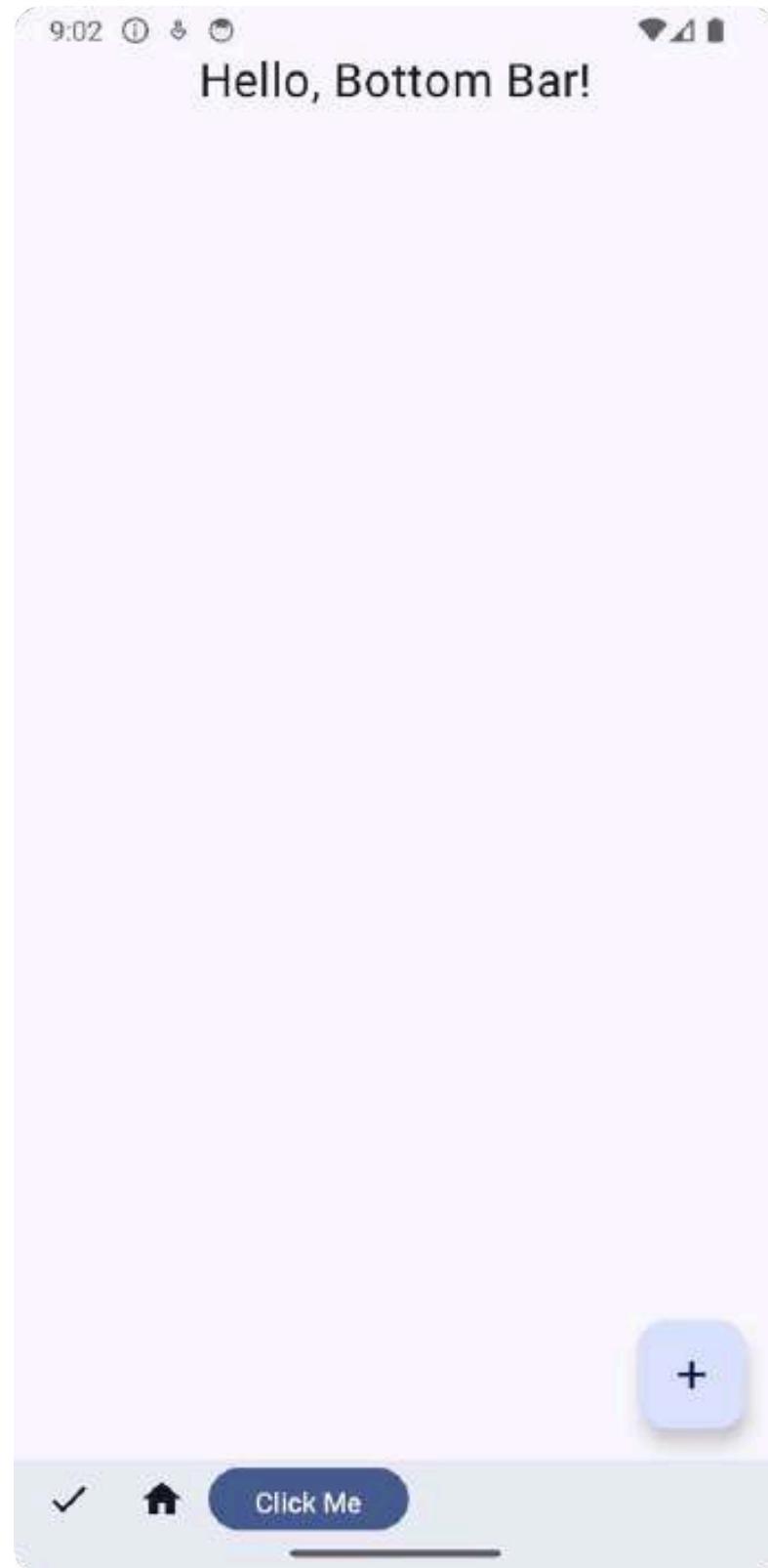
##### How will it work?

- We'll add a **bottom bar** with **buttons and icons**.
- One of the icons will **open a second screen** when clicked.
- There will also be a **Floating Action Button (FAB)** for a primary action.

##### Final Expected Output

At the end of this activity, your app will have:

- A **Bottom Navigation Bar** with icons and buttons.
- A **Floating Action Button (FAB)** on the bottom right.
- A **clickable icon** that opens a **new screen**.



====

## Step 2: Set Up Your Android Studio Project

1. Open Android Studio
2. Repeat steps from previous activities to create a new "Empty Compose Activity"
3. Run your app and make sure it works (it should show "Hello World").

====

# Step 3: Add the Bottom Navigation Bar

We'll use Jetpack Compose's `Scaffold` and `BottomAppBar` components.

## 3.1: Create a Composable Function for the Bottom Bar

- Open `MainActivity.kt`
- Add this function **below** the `onCreate` function.

```
@Composable
fun MyBottomBar() {
    BottomAppBar(
        modifier = Modifier.height(60.dp), // Set the height
        content = {
            IconButton(onClick = { /* TODO: Add action */ }) {
                Icon(Icons.Filled.Check, contentDescription = "Check icon")
            }
            IconButton(onClick = { /* TODO: Navigate to another screen */ }) {
                Icon(Icons.Filled.Home, contentDescription = "Go Home")
            }
            Button(onClick = { /* TODO: Add button action */ }) {
                Text("Click Me")
            }
        }
    )
}
```

## 3.2: Call this function from `setContent` in `MainActivity.kt`

Modify `MainActivity.kt` so that it calls our new function.

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

        setContent {
            MyBottomBar()
        }
    }
}
```

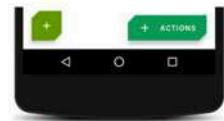
**Run your app** - you should see the Bottom Bar. Now let's also add a Floating Action Button (FAB).

*Refresh: This is what a FAB looks like on Android devices - see Figure below (Source: [geeksforgeeks.org](https://geeksforgeeks.org)).*

Examples of Theming Material Design  
Floating Action Button as well as Extended Floating Action Button



Example 1



Example 2



Example 3



Example 4

DG

### 3.3: Add the Floating Action Button (FAB)

Modify the `MyBottomBar` function to include a **Floating Action Button**. We need to make a few changes to make this possible.

- **Built-in Structure:** `Scaffold` gives you a high-level layout structure with a dedicated slot for a `bottomBar`, a slot for a `floatingActionButton`, and space for the main screen content.
- **Bottom Bar Integration:** The `BottomAppBar` is now passed to the `bottomBar` parameter of `Scaffold`. This means it's automatically positioned at the bottom, and Compose handles layout details like sizing and insets for you.
- **Floating Action Button:** You add a `floatingActionButton` parameter. This is how Compose knows to float a button above the bottom bar in a standard Material way.
- **Main Content:** There's now a `Column` (or any other layout) within the main content area of the `Scaffold`. The `innerPadding` passed into that `Column` ensures your content doesn't clash with the bar or the floating action button.
- **Single Entry Point:** Because `Scaffold` provides these slots (bottom bar, FAB, content), it acts as the "single container" for your screen. This approach is more scalable if you want to add things like a `topBar` in the future or a navigation drawer.

```
@Composable
fun MyBottomBar() {
    Scaffold(
        bottomBar = {
            BottomAppBar(
                modifier = Modifier.height(60.dp),
                content = {
                    IconButton(onClick = { /* TODO: Add action */ }) {
                        Icon(Icons.Filled.Check, contentDescription = "Check icon")
                    }
                    IconButton(onClick = { /* TODO: Navigate to another screen */ }) {
                        Icon(Icons.Filled.Home, contentDescription = "Go Home")
                    }
                    Button(onClick = { /* TODO: Add button action */ }) {
                        Text("Click Me")
                    }
                }
            )
        }
    )
}
```

```

        }
    )
},
floatingActionButton = {
    FloatingActionButton(onClick = { /* TODO: Add FAB action */ }) {
        Icon(Icons.Filled.Add, contentDescription = "Add something")
    }
}
) { innerPadding ->
Column(
    modifier = Modifier.padding(innerPadding).fillMaxSize(),
    horizontalAlignment = Alignment.CenterHorizontally
) {
    Text(text = "Hello, Bottom Bar!", fontSize = 24.sp)
}
}
}
}

```

**Run your app** – You should now see the Bottom Navigation Bar and the Floating Action Button.

====

## Step 4: Open a New Screen When Clicking an Icon

### 4.1: Create a New Screen (SecondActivity)

1. In **Android Studio**, right-click `com.example.yourappname`
2. Select **New → Activity → Empty Activity**
3. Name it **SecondActivity**
4. Replace the generated code in `SecondActivity.kt` with this:

```

class SecondActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContent {
            Column(
                modifier = Modifier.fillMaxSize(),
                horizontalAlignment = Alignment.CenterHorizontally,
                verticalArrangement = Arrangement.Center
            ) {
                Text(text = "Welcome to the Second Screen!", fontSize = 24.sp)
            }
        }
    }
}

```

## 4.2: Modify the Home Icon in `MyBottomBar` to Navigate

Now, go back to `MainActivity.kt` and **modify the home icon click action**:

```
val context = LocalContext.current // Get the current activity context

IconButton(onClick = {
    context.startActivity(Intent(context, SecondActivity::class.java))
}) {
    Icon(Icons.Filled.Home, contentDescription = "Go Home")
}
```

**Run your app** and **click the Home icon** – it should open the second screen!

====

## Summary of What We Did

Created a **Bottom Navigation Bar** using `BottomAppBar`.

Added **buttons and icons** inside the bar.

Added a **Floating Action Button (FAB)**.

Created a **Second Screen** (`SecondActivity`).

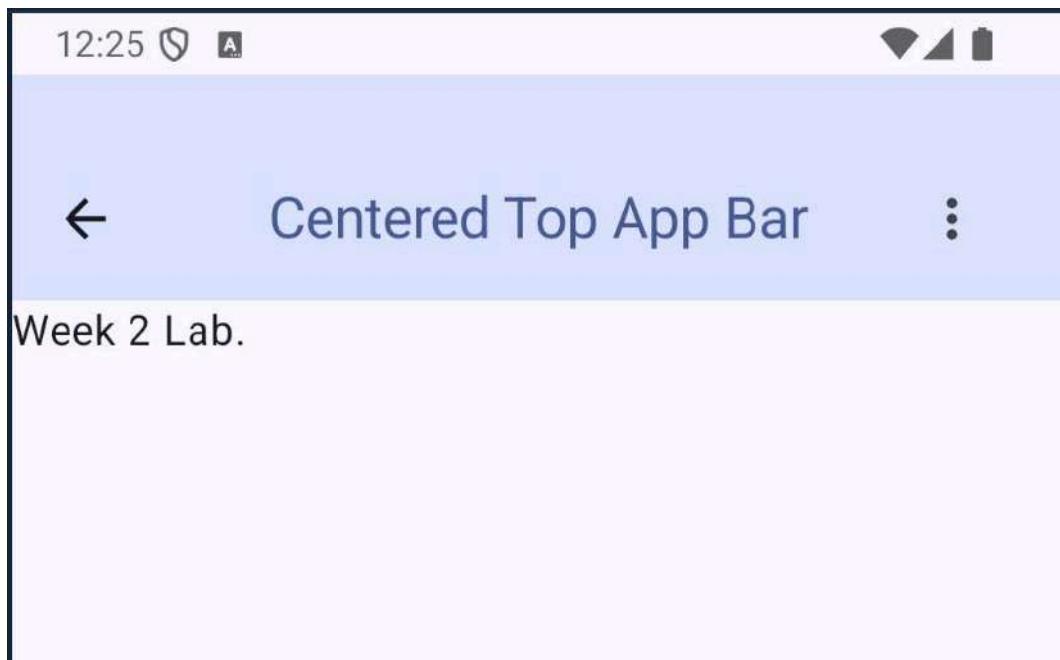
Made the **home icon open the Second Screen**.

## 4. Top Menu Bar

In the previous lesson, we implemented the bottom menu bar, adding several icons and buttons to navigate different tasks.

In this lesson, we will add a top menu bar to an activity with a back button to navigate to the previous activity and an options menu with three menu items.

Here is the expected output:



12:26 ⚡ A



Centered Top App Bar



Week 2 Lab.

Edit

Settings

Send Feedback F11

Let's start with the options menu.

```

1  @Composable
2  fun DropdownMenu() {
3      //The expanded property is used to control the visibility of the dropdown menu.
4      //if the value is true, the dropdown menu will be visible, otherwise it will be hidden.
5      var expanded by remember { mutableStateOf(false) }
6      //The Box composable is used to place a composable inside another composable.
7      Box(modifier = Modifier.padding(16.dp)) {
8          IconButton(onClick = { expanded = true }) {
9              Icon(Icons.Default.MoreVert, contentDescription = "More options")
10     }
11     DropdownMenu(
12         expanded = expanded,
13         //The onDismissRequest property is used to handle the dismissal of the dropdown menu.
14         //it changes the value of expanded to false to hide the menu.
15         onDismissRequest = { expanded = false }
16     ) {
17         //Here is the list of menu items.
18         //DropdownMenuItem is a composable that represents a single menu item in the dropdown menu.
19         DropdownMenuItem(
20             text = { Text("Edit") },
21             onClick = { /* Handle edit! */ },
22             leadingIcon = { Icon(Icons.Outlined.Edit, contentDescription = null) }
23         )
24         DropdownMenuItem(
25             text = { Text("Settings") },
26             onClick = { /* Handle settings! */ },
27             leadingIcon = { Icon(Icons.Outlined.Settings, contentDescription = null) }
28         )
29         HorizontalDivider()
30         DropdownMenuItem(
31             text = { Text("Send Feedback") },
32             onClick = { /* Handle send feedback! */ },
33             leadingIcon = { Icon(Icons.Outlined.Email, contentDescription = null) },
34             trailingIcon = { Text("F11", textAlign = TextAlign.Center) }
35         )
36     }
37 }

```

codesnap.dev

The `DropdownMenu` function in the provided code is a composable function that creates a dropdown menu UI element. It utilizes a mutable state variable `expanded` to control the visibility of the menu. When the `IconButton` (represented by the three vertical dots) is clicked, it sets `expanded` to true, making the menu visible. The `DropdownMenu` composable then renders the actual menu items. The `onDismissRequest` lambda function is triggered when the menu needs to be closed, such as when the user clicks outside the menu.

Inside the menu, `DropdownMenuItem` composables are used to define each item. These items include text labels and icons. The `onClick` lambda function within each `DropdownMenuItem` is where you would define the action to be performed when that item is selected. Additionally, there's a `HorizontalDivider` for visual separation and `trailingIcon` support to display elements like shortcuts. In short, `DropdownMenu` provides a flexible and customizable way to create dropdown menus in Jetpack Compose, handling both the display and the actions associated with each menu item.

Now, let's develop the function that builds the top menu bar.

```

1  @OptIn(ExperimentalMaterial3Api::class)
2  @Composable
3  fun TopAppBarExample() {
4      // Create a TopAppBarState object to control the behavior of the TopAppBar.
5      // RememberTopAppBarState() is a composable function that creates a TopAppBarState that is remembered across compositions.
6      val scrollBehavior = TopAppBarDefaults.enterAlwaysScrollBehavior(rememberTopAppBarState())
7      // The onBackPressedDispatcher is used to handle the back button press in the app.
8      val onBackPressedDispatcher = LocalOnBackPressedDispatcherOwner.current?.onBackPressedDispatcher
9
10     Scaffold(
11         topBar = {
12             CenterAlignedTopAppBar(
13                 // the colors property is used to customize the appearance of the TopAppBar.
14                 colors = TopAppBarDefaults.centerAlignedTopAppBarColors(
15                     containerColor = MaterialTheme.colorScheme.primaryContainer,
16                     titleContentColor = MaterialTheme.colorScheme.primary,
17                 ),
18                 // Title displayed in the center of the app bar
19                 title = {
20                     Text(
21                         "Centered Top App Bar",
22                         maxLines = 1,
23                         // the Ellipsis property is used to truncate the text if it exceeds the available space.
24                         overflow = TextOverflow.Ellipsis
25                     )
26                 },
27                 // Navigation icon (back button) with appropriate behavior
28                 navigationIcon = {
29                     IconButton(onClick = {
30                         // onBackPressedDispatcher is used to handle the back button press in the app.
31                         // it takes the current activity out of the back stack and shows the previous activity.
32                         onBackPressedDispatcher?.onBackPressed()
33                     }) {
34                         Icon(
35                             imageVector = Icons.AutoMirrored.Filled.ArrowBack,
36                             contentDescription = "Localized description"
37                         )
38                     }
39                 },
40                 // Action icons (currently a dropdown menu)
41                 actions = {
42                     DropdownMenu()
43                 },
44                 scrollBehavior = scrollBehavior,
45             )
46         }
47     ) { innerPadding ->
48         Text(
49             modifier = Modifier.padding(innerPadding),
50             text = "Week 2 Lab."
51         )
52     }
53 }

```

codesnap.dev

- The actions parameter of the CenterAlignedTopAppBar takes a Composable function that is used to display actions on the right side of the TopAppBar.
- In this example, the actions parameter is set to a lambda expression that returns a DropdownMenu composable function.
- The DropdownMenu function displays a dropdown menu with several menu items.
- Each menu item is represented by a DropdownMenuItem composable function.
- The DropdownMenuItem function takes a text parameter, an onClick parameter, and a leading icon parameter.
- The text parameter is used to display the text of the menu item.

- The onClick parameter is a lambda expression called when the menu item is clicked.
- The leadingIcon parameter is a composable function that is used to display an icon to the left of the text.
- The DropdownMenu function also takes an expanded parameter, which is a boolean value that determines whether the menu is expanded or not.
- The onDismissRequest parameter is a lambda expression called when the user clicks outside the menu.
- The DropdownMenu function is placed inside a Box composable function, which positions the menu relative to the IconButton.

Finally, we have to call the TopAppBarExample function from the onCreate function.

```
1 class MainActivity : ComponentActivity() {
2     override fun onCreate(savedInstanceState: Bundle?) {
3         super.onCreate(savedInstanceState)
4         enableEdgeToEdge()
5         setContent {
6             LabWeek3Theme {
7                 Scaffold(
8                     modifier = Modifier.fillMaxSize(),)
9                     { innerPadding ->
10                         Column(modifier = Modifier.padding(innerPadding).fillMaxSize(),
11                               horizontalAlignment = Alignment.CenterHorizontally) {
12                             TopAppBarExample()
13                         }
14                     }
15                 }
16             }
17         }
18     }
```

## 5. Simple Quiz Application

In this activity, we will develop an application that asks the user three questions. If the answers are correct, a second activity shows a green tick. Otherwise, it shows a red fail sign.

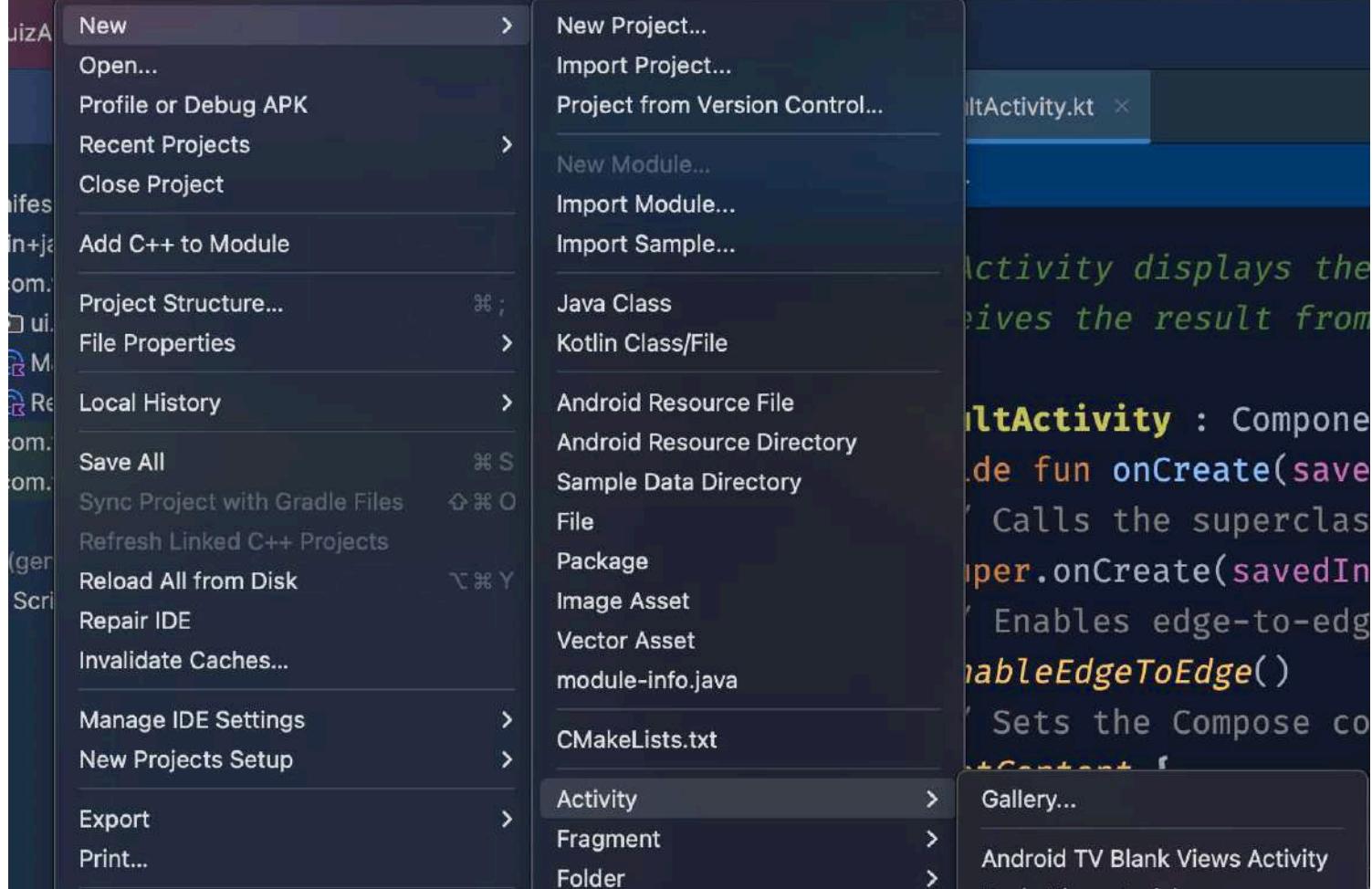
The main objective of this activity is to implement the following:

- Checkbox
- Buttons
- Multiple Activity
- Navigation
- ImageView
- Passing data between activities

 For more information on how to layout UI elements, read this article:  
<https://developer.android.com/develop/ui/compose/layouts/basics>

Create a project with two activities: MainActivity.kt and ResultActivity.kt.

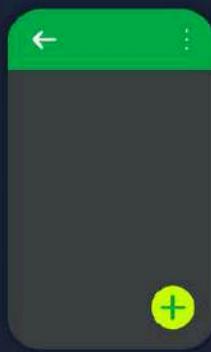
To add the second activity, select Gallary from the file menu as shown below:



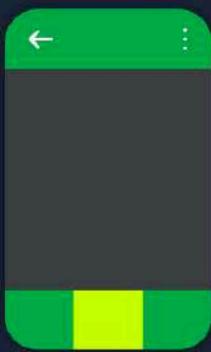
Then, choose Empty Activity.



Empty Activity



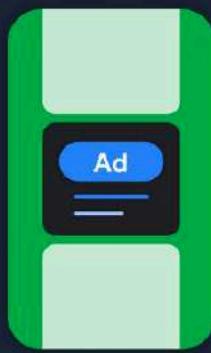
Basic Views Activity



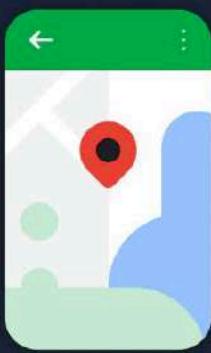
Bottom Navigation Views Activity



Fullscreen Views Activity



Google AdMob Ads Views Activity



Google Maps Views Activity

In the `MainActivity.kt`, let's do the following:

- Add a Scaffold to the theme to provide the basic layout theme.
- Because our UI elements must be positioned vertically, we add the 'Column' child inside the Scaffold (see the article above for more info).

```

1      setContent {
2          // Use the QuizAppTheme for styling
3          QuizAppTheme {
4              // Scaffold to provide basic Material Design layout structure
5              Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding →
6                  Column(modifier = Modifier.padding(innerPadding).fillMaxSize(),
7                         horizontalAlignment = Alignment.CenterHorizontally
8                  ) {...}

```

[codesnap.dev](https://codesnap.dev)

- The app logo is an image. So, to appear at the very top, it should be the first child of the Column element.

```
1 Column(modifier = Modifier.padding(innerPadding).fillMaxSize(),
2         horizontalAlignment = Alignment.CenterHorizontally) {
3     // Add the image at the top
4     Image(
5         painter = painterResource(id = R.drawable.fit2081_logo_yellow),
6         contentDescription = "Quiz Image",
7         modifier = Modifier
8             .fillMaxWidth()
9             .height(200.dp) // Adjust height as needed
10            .padding(16.dp)
11    )
}
```

codesnap.dev

- Now, we must add three checkboxes and listen to their changes. Checkboxes are data input elements with two values: true and false. Therefore, we should declare three boolean variables, one for each checkbox.

```
1 // Mutable state variables to track checkbox states
2 var checked1 by remember { mutableStateOf(false) }
3 var checked2 by remember { mutableStateOf(false) }
4 var checked3 by remember { mutableStateOf(false) }
```

codesnap.dev

**i** **mutableStateOf**: is a function in Jetpack Compose that creates an observable `MutableState` object.

**i** **MutableState**: An interface that holds a single value and is designed to be mutable. It allows you to read and write to a variable, but it also signals changes to the Compose runtime.

**i** **Observable**: When a value within a `MutableState` object changes, the Compose runtime automatically knows about it.

**i** `remember`: is your tool for storing and preserving values across recompositions within Jetpack Compose. It allows you to maintain the state of an object, ensuring that the data isn't reset or lost during UI updates.

If you want to change the vertical alignment from the parent `Column`, add a child one and set its local

alignment property to the desired value.

Now, let's add the three checkboxes and their captions. We will use the Row UI element to position the checkbox and its label horizontally.

```
1 // Mutable state variables to track checkbox states
2 var checked1 by remember { mutableStateOf(false) }
3 var checked2 by remember { mutableStateOf(false) }
4 var checked3 by remember { mutableStateOf(false) }
5
6 // Column to hold checkboxes and labels and align them to left (start)
7 Column(modifier = Modifier.fillMaxWidth().padding(16.dp), horizontalAlignment = Alignment.Start) {
8     // Checkboxes with labels
9     // Row for the first checkbox and label
10    Row(verticalAlignment = Alignment.CenterVertically) {
11        Checkbox(checked = checked1, onCheckedChange = { checked1 = it })
12        Text("Q1: The Earth is Flat.")
13    }
14    // Row for the second checkbox and label
15    Row(verticalAlignment = Alignment.CenterVertically) {
16        Checkbox(checked = checked2, onCheckedChange = { checked2 = it })
17        Text("Q2: Canberra is the capital of Australia. ")
18    }
19    // Row for the third checkbox and label
20    Row(verticalAlignment = Alignment.CenterVertically) {
21        Checkbox(checked = checked3, onCheckedChange = { checked3 = it })
22        Text("Q3: The speed of sound is 1,236 km/h.")
23    }
24}
```

codesnap.dev

From the code above, the boolean variable created in line 2 is passed as a variable to the first Checkbox in line 11. In other words, the variable's value will be automatically synced with the checkbox's actual value.

The last thing in MainActivity is to add a button to show the results activity.

```
1 // Button to go to next activity
2 Button(onClick = {
3     // logic to check if the answer correct or not
4     var resultBoolean = !checked1 && checked2 && checked3;
5     val intent =
6         Intent(this@MainActivity, ResultActivity::class.java).apply {
7             putExtra("isPassed", resultBoolean)
8         }
9     startActivity(intent)
10 }) {
11     Text("Check Result")
12 }
```

codesnap.dev

In the code above, we calculated the total results using a logic expression. Then, we added the value and its key to the Intent object, which will be used to start the result activity.

**i** **Intents:** are the primary way that Android components communicate and interact. They are messaging objects used to start Activities, Services, and deliver broadcasts.

**i** For more info about Intent, see this link: <https://developer.android.com/guide/components/intents-filters>

So, the MainActivity in one piece should be:

```
1 class MainActivity : ComponentActivity() {
2     // Override the onCreate method to initialize the activity
3     override fun onCreate(savedInstanceState: Bundle?) {
4         // Call the superclass onCreate method
5         super.onCreate(savedInstanceState)
6         // Enable edge-to-edge display
7         enableEdgeToEdge()
8
9         // Set the content of the activity
10        setContent {
11            // Use the QuizAppTheme for styling
12            QuizAppTheme {
13                // Scaffold to provide basic Material Design layout structure
14                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding -
15                    Column(
16                        modifier = Modifier
17                            .padding(innerPadding)
18                            .fillMaxSize(),
19                            horizontalAlignment = Alignment.CenterHorizontally
20                    ) {
21                        // Add the image at the top
22                        Image(
23                            painter = painterResource(id = R.drawable.fit2081_logo_yellow),
24                            contentDescription = "Quiz Image",
25                            modifier = Modifier
26                                .fillMaxWidth()
27                                .height(200.dp) // Adjust height as needed
28                                .padding(16.dp)
29                        )
30                        // Mutable state variables to track checkbox states
31                        var checked1 by remember { mutableStateOf(false) }
32                        var checked2 by remember { mutableStateOf(false) }
33                        var checked3 by remember { mutableStateOf(false) }
34
35                        // Column to hold checkboxes and labels and align them to left (start)
36                        Column(
37                            modifier = Modifier
38                                .fillMaxWidth()
39                                .padding(16.dp),
40                                horizontalAlignment = Alignment.Start
41                    ) {
42                        // Checkboxes with labels
43                        // Row for the first checkbox and label
44                        Row(verticalAlignment = Alignment.CenterVertically) {
45                            Checkbox(checked = checked1, onCheckedChange = { checked1 = it })
```

```

46             Text("Q1: The Earth is Flat.")
47         }
48         // Row for the second checkbox and label
49         Row(verticalAlignment = Alignment.CenterVertically) {
50             Checkbox(checked = checked2, onCheckedChange = { checked2 = it })
51             Text("Q2:Canberra is the capital of Australia. ")
52         }
53         // Row for the third checkbox and label
54         Row(verticalAlignment = Alignment.CenterVertically) {
55             Checkbox(checked = checked3, onCheckedChange = { checked3 = it })
56             Text("Q3: The speed of sound is 1,236 km/h.")
57         }
58     }
59     // Button to go to next activity
60     Button(onClick = {
61         //logic to check if the answer correct or not
62         var resultBoolean = !checked1 && checked2 && checked3;
63         val intent =
64             Intent(this@MainActivity, ResultActivity::class.java).apply {
65                 putExtra("isPassed", resultBoolean)
66             }
67             startActivity(intent)
68         }) {
69             Text("Check Result")
70         }
71     }
72 }
73 }
74 }
75 }
76 }
77

```

codesnap.dev

Now, let's develop the second activity, "ResultActivity.kt"

- The first step is to retrieve the boolean value the MainActivity has sent through the Intent.

```
val isPassed = intent.getBooleanExtra("isPassed", false)
```

codesnap.dev

The first parameter is the key of the value, which is the same key the MainAcitivty has used. The second parameter is the default value used if the key does not exist in the intent.

- We will use the same approach in MainActivity by adding a Scaffold as the root element, followed by a Column to position the results element vertically.
- After getting the boolean value, an if statement should do the job of showing the required image.

```
    if (isPassed) {
        // Display a green tick image if the quiz was passed.
        Image(
            painter = painterResource(R.drawable.green_tick),
            contentDescription = "Passed",
            modifier = Modifier.size(100.dp)
        )
    } else {
        // Display a red cross image if the quiz was failed.
        Image(
            painter = painterResource(R.drawable.red_cross),
            contentDescription = "Failed",
            modifier = Modifier.size(100.dp)
        )
}
```

codesnap.dev

Finally, let's add a button that terminates the current activity that leads to bringing back the previous one.

So, the second activity in one piece should be:

```
1 class ResultActivity : ComponentActivity() {
2     override fun onCreate(savedInstanceState: Bundle?) {
3         // Calls the superclass implementation.
4         super.onCreate(savedInstanceState)
5         // Enables edge-to-edge display for a more immersive UI.
6         enableEdgeToEdge()
7         // Sets the Compose content for this activity.
8         setContent {
9             // Retrieves the quiz result (true for pass, false for fail) from the intent.
10            val isPassed = intent.getBooleanExtra("isPassed", false)
11            QuizAppTheme {
12                // Creates a scaffold with a fill max size to manage app layout.
13                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
14                    // main Layout
15                    Column(
16                        modifier = Modifier
17                            .padding(innerPadding)
18                            .fillMaxSize(),
19                        verticalArrangement = Arrangement.Center,
20                        horizontalAlignment = Alignment.CenterHorizontally
21                    ) {
22                        if (isPassed) {
23                            // Display a green tick image if the quiz was passed.
24                            Image(
25                                painter = painterResource(R.drawable.green_tick),
26                                contentDescription = "Passed",
27                                modifier = Modifier.size(100.dp)
28                            )
29                        } else {
30                            // Display a red cross image if the quiz was failed.
31                            Image(
32                                painter = painterResource(R.drawable.red_cross),
33                                contentDescription = "Failed",
34                                modifier = Modifier.size(100.dp)
35                            )
36                        }
37                        // finish the activity when click Done button
38                        Button(onClick = { finish() }) {
39                            Text(text = "Done")
40                        }
41
42                    }
43                }
44            }
45        }
46    }
47 }
48 }
```

Here is the output:



4:39

Monash University  
**FIT2081**

- Q1: The Earth is Flat.
- Q2: Canberra is the capital of Australia.
- Q3: The speed of sound is 1,236 km/h.

Check Result

+  
-  
1:1

## 6. Challenge Exercise

If you have completed Activities 1-5, try these challenge exercises to stretch your Android skills!

1. Add a signup screen to the LoginScreen app. The signup screen should ask the user for their username and password, confirm the password, and forward them to the login screen to be used instead of the static strings.
2. Develop an application with a Main Activity, a toolbar (Top menu bar), and a bottom menu bar. The toolbar should have an options menu with two items to navigate to two activities. The bottom menu bar has three icons to help you navigate to three activities. Add a button to the bottom menu bar to exit the app.
3. Add more questions to the quiz app. Each question could be implemented as a separate activity and use different UI elements, such as Radio Buttons, progress bars, or normal buttons.

# Conclusion and Skills Learnt

As you work through this week's lab, you'll be developing foundational skills that will directly support your success in **Assignment 1A1\_ Assessment (1)**. This assignment focuses on creating a **mobile application with key screens like a login system, questionnaire, and data visualization**. The skills you will gain in this lab will help you complete major parts of your assignment, including:

**User Input Handling:** You'll practice building text fields for username and password entry, which will help you implement the **Login Screen** in your assignment.

**State Management in Jetpack Compose:** Using `remember` and `mutableStateOf` to store and update user inputs dynamically—essential for handling **form submissions** and user interactions in Assignment 1.

**UI Layouts & Composables:** You'll work with **Surface, Column, and Row** to structure UI elements effectively—skills that will be essential for designing the **Questionnaire and Home screens** in your project.

**Navigation Between Screens:** You'll implement navigation between activities, which is a crucial component of **switching between screens (Login → Questionnaire → Home → Insights)** in your assignment.

**Displaying Feedback with Toasts:** Learning how to show quick error messages or success notifications, useful for **validating form inputs and guiding users through the app workflow**.

**Understanding Jetpack Compose's Scaffold & UI Components:** This will set the foundation for implementing **menus, buttons, and structured layouts**—key features needed for designing an interactive, user-friendly app.

If you completed this lab, you'll be equipped with the practical skills needed to start **Assignment 1 with confidence**, ensuring a smoother development process when you build your **NutriTrack** app!

====

Well done you!

▶ Expand

# FIT2081 Week 3

## Android Project Structure & UI Layouts

# Lecture - Android Lifecycle & Persistence

---

## Learning Objectives

- Life Cycles, Persistence & Activities
- Examine and understand the structure of the Android Life cycle
- Look at varying degrees of data persistence
- Work with Intents & multiple activities

# Mobile OSs

## Mobile OSs are Different

- “Regardless of the fanfare about how much memory and computing power resides in the mobile devices of today compared to the desktop systems of yesterday, it is important to keep in mind that these devices are still considered to be “resource constrained” by the standards of modern desktop and laptop-based systems, particularly in terms of memory.
- As such, a key responsibility of the Android system is to ensure that these limited resources are managed effectively and that both the operating system and the applications running on it remain responsive to the user at all times.
- To achieve this, Android is given full control over the lifecycle and state of both the processes in which the applications run, and the individual components that comprise those applications.
  - [the Android app developer is not!]
  - See [here](#) for a heated exchange as developers struggle with aspects of this new OS paradigm and [here](#) for a calm analysis.
- An important factor in developing Android applications, therefore, is to gain an understanding of both the application and activity lifecycle management models of Android and how an application [i.e. the application’s developer] can react to the state changes that are likely to be imposed upon it during its execution lifetime.”

## Android is a Killer!

- “Each running Android application is viewed by the operating system as a separate process.
- If the system identifies that resources on the device are reaching capacity it will take steps to terminate processes to free up memory.
- When making a determination as to which process to terminate in order to free up memory, the system takes into consideration both the priority and state of all currently running processes, combining these factors to create what is referred to by Google as an importance hierarchy.
- Processes are then terminated starting with the lowest priority and working up the hierarchy until sufficient resources have been liberated for the system to function.
- Processes host applications and applications are made up of components. Within an Android system, the current state of a process is defined by the highest-ranking active component within the application that it hosts.”

## Who Gets Killed?

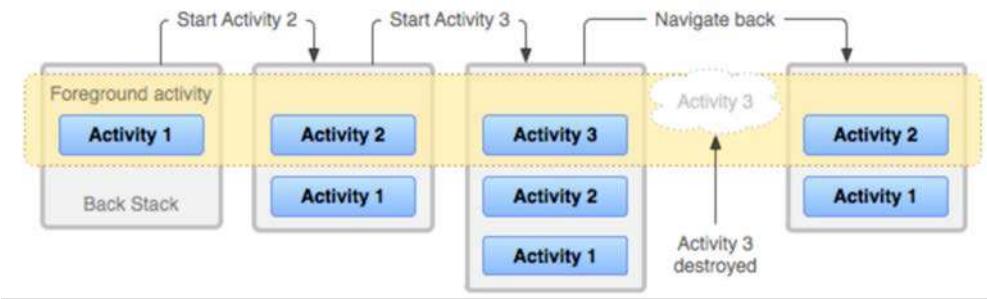
- So Android will terminate/kill a process if it needs resources
  - Remember each process contains an ART VM running an app which consists of components (e.g. activities, services etc.)
- It will rank processes by their highest ranking active component
  - Processes will be terminated/killed if necessary, lowest ranking first
  - Complication: Processes can depend on each other \*
    - A process can never be ranked lower than the process it's serving
  - To keep things simple\*\* we will now concentrate on the visible components of an app i.e. Activities
    - How are Activities ranked? By 3 of their 4 possible Lifecycle states!
      - 1. Activity is in the foreground (user is interacting/can interact with it)
      - 2. Activity's UI is partially hidden\*\*\*
      - 3. Activity's UI is fully hidden
    - Only 1 process contains a foreground Activity and it cannot be killed
    - All other processes can be killed beginning with those that only have Activities with fully hidden UIs
      - Killing a process destroys all of its app's components of course

# Activity Lifecycles

- So the state of a process's Activities determines its vulnerability to an OS kill
- Activities have 4 possible Lifecycle states
  1. Foreground
  2. Partially hidden
  3. Fully hidden
  4. Destroyed
- They transit between these states as a result of:
  - User interaction
    - This involves the concept of Tasks and Activity Back Stacks
      - See next slide
    - OS process kills
      - When a process is killed, all its app's components are destroyed including its Activities

# Stacks

- “A task is a collection of activities that users interact with when performing a certain job. The activities are arranged in a stack (the back stack), in the order in which each activity is opened.
- “When the current activity starts another, the new activity is pushed on the top of the stack and takes focus. The previous activity remains in the stack, but is stopped. When an activity stops, the system retains the current state of its user interface.
- When the user presses the Back button, the current activity is popped from the top of the stack (the activity is destroyed) and the previous activity resumes (the previous state of its UI is restored). Activities in the stack are never rearranged, only pushed and popped from the stack —pushed onto the stack when started by the current activity and popped off when the user leaves it using the Back button.”
- If the user continues to press Back, then each activity in the stack is popped off to reveal the previous one, until the user returns to the Home screen (or to whichever activity was running when the task began). When all activities are removed from the stack, the task no longer exists.
- A task is a cohesive unit that can move to the “background” when users begin a new task or go to the Home screen, via the Home button. While in the background, all the activities in the task are stopped, but the back stack for the task remains intact—the task has simply lost focus while another task takes [its] place [...]. A task can then return to the “foreground” so users can pick up where they left off.”



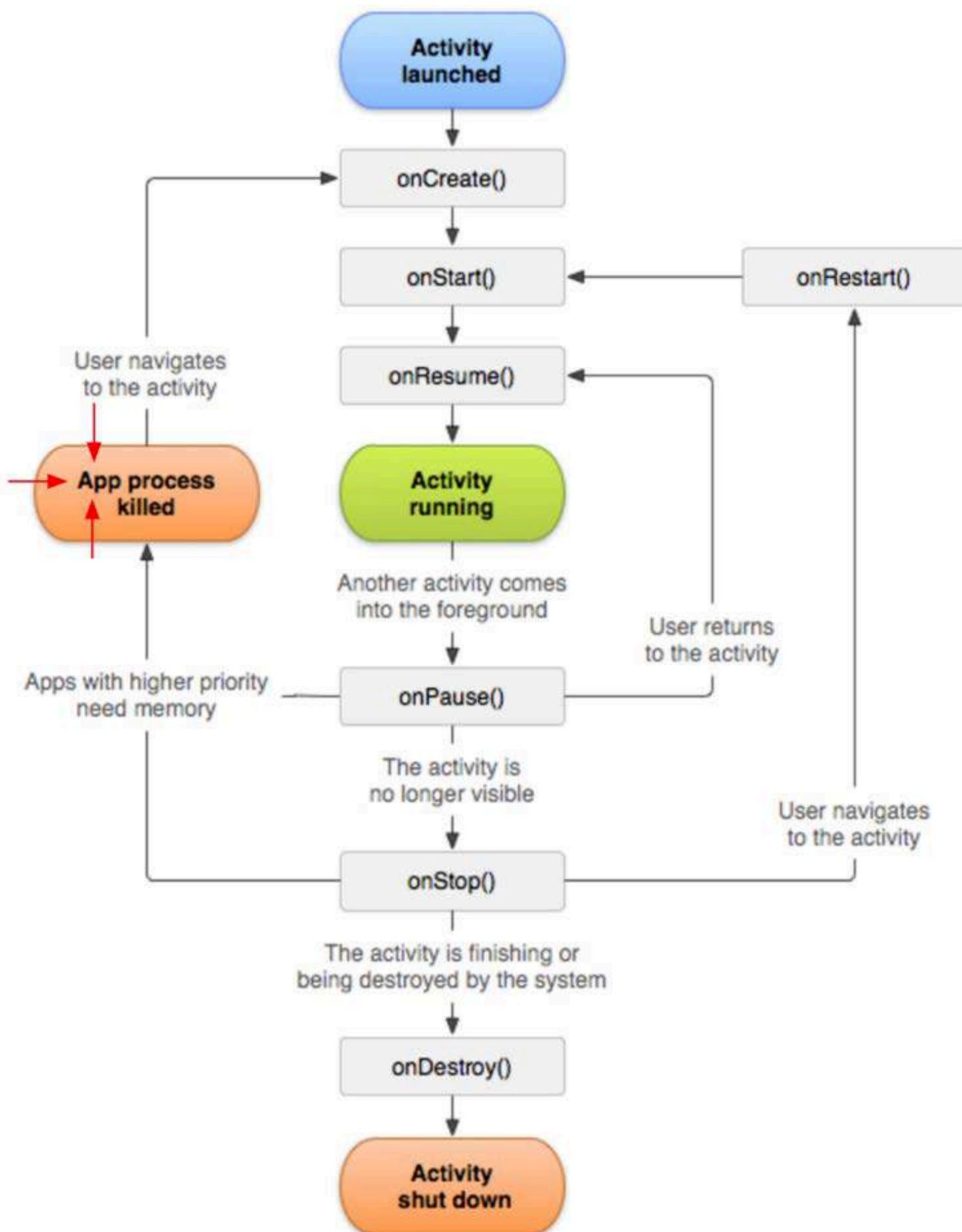
## There's More to It

- Complications
  - A task back stack can include Activities from several different apps
    - Remember an Activity can intent an Activity in another app
  - If the user performs an action that causes code to execute an Activity's finish() method this is entirely equivalent to the user hitting the back button (i.e. its Activity is destroyed)
  - Putting your phone to sleep has the same effect as pushing a blank activity onto the current task back stack
    - Waking your phone pops the blank Activity
  - There can be multiple background back stacks one for each in-progress task interrupted by a new foreground task
    - If one of these in-progress tasks is returned it becomes the foreground task and picks up where it left off
    - Multiple independent instances of the same Activity may appear in different back stacks if multiple tasks include it
  - Fragments (more on these later)
    - These UI "sub-Activities" have their own lifecycle albeit slaved to their parent Activity's lifecycle
    - Fragment transactions (such as displaying a fragment or swap fragments) can be pushed onto the back stack of a task making them reversible

## Activity Lifecycle

- Activity States
  - foreground, partially hidden, fully hidden, destroyed
- Lifecycle Callbacks
  - onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy()
- Lifecycle Loops
  - Partially hidden
    - Navigate back before destroyed (overview, home)
  - Fully hidden
    - Navigate back before destroyed (overview, home)

- Process Destroyed
  - App relaunched
- “Back” destroys an Activity!



# Lifecycle Callbacks

- Out of Control?
  - User interaction and OS process kills drag our Activities around their lifecycle without our say so
    - As coders are we doomed to watch as helpless bystanders
  - Not really!
    - “As a user navigates through, out of, and back to your app, the Activity instances in your app transition through different states in their lifecycle. The Activity class provides a number of callbacks that allow the activity to know that a state has changed: that the system is creating, stopping, resuming, or destroying an activity.”
      - Our custom Activities (Activity subclasses) inherit these callbacks which can be overridden to contain our custom responses to their state transitions
- “Within the lifecycle callback methods, you can declare how your activity behaves when the user leaves and re-enters the activity.”
  - For example, if you're building a streaming video player, you might pause the video and terminate the network connection when the user switches to another app. When the user returns, you can reconnect to the network and allow the user to resume the video from the same spot.
  - In other words, each callback allows you to perform specific work that's appropriate to a given change of state. Doing the right work at the right time and handling transitions properly make your app more robust and performant.
  - For example, good implementation of the lifecycle callbacks can help ensure that your app avoids:
    - Crashing if the user receives a phone call or switches to another app while using your app.
    - Consuming valuable system resources when the user is not actively using it.
    - Losing the user's progress if they leave your app and return to it at a later time.
    - Crashing or losing the user's progress when the screen rotates between landscape and portrait orientation.”
- Call super in all Lifecycle callbacks
  - Always begin lifecycle callback overrides with a call to their super
    - e.g. onCreate(...) override should begin with super.onCreate(...)
  - The supers perform a lot of standard processing required by any Activity during state transitions.
    - Our code just adds customised processing specific to our particular Activity in lifecycle callback overrides

## More Activity Callbacks

- `onSaveInstanceState(...), onRestoreInstanceState(...)`
  - Lifecycle callbacks are guaranteed to execute at the appropriate transitions between Activity states
  - These are 2 other important Activity callbacks that play a role in saving the state of an Activity (we will deal with this next)
    - These 2 callbacks will execute during specific transitions but only if certain conditions are met
    - As with the Lifecycle callbacks it's important to call the super of these callbacks
      - It turns out Android auto-saves and restores a lot (but not all) of an Activity's state in these methods

# Activity Data

- **What data?**

- Activity Instance Data
  - (i.e. data associated with a single use (instance) of the Activity)
    - Activity view data
      - i.e. the state of all Views in the Activity's layout
    - Activity non-view data
      - e.g. Activity instance variables
  - This involves writing to and reading from the device's volatile memory (this is a simplification)
- Persistent data
  - i.e. data associated with multiple uses of the Activity separated by any amount of time i.e. it spans an Activity's instances
    - Persistent across app uses implies across app destroys
      - Because uses are delineated by users indicating they are finished with an Activity which will involve a destroy
      - e.g. via back button or user interaction causes Activity's finish() method to execute
    - This involves writing to and reading from the device's non-volatile memory
    - Such data may be accessible by more than one Activity in an app so may be considered a part of the App's state rather than any Activity's state
    - It may also be loaded into an Activity's instance variable. Whose state now?

- **What events?**

- Activity is partially or fully hidden (+ No OS kill, + OS kill)
  - Activity is still in memory, no state data (view or non-view) is lost
  - Remember putting your phone to sleep is the same as completely hiding it with a blank Activity
  - Based on just this Activity its app's process is eligible for an OS kill. Why?  
What if this happens?
    - Surprisingly Android automatically saves the Activity's view-state away in a Bundle object (an object capable of storing key/value pairs) that is stored in an Activity record in the OS's own process since the Activity's process has been destroyed (how long for?)
    - Developers must code to save and restore non-view state in the same Bundle object (Android provides and calls callbacks to code to do this)
    - Persistent data remains in its last updated state in non-volatile memory
- Activity is destroyed using the back button
  - Or user interaction causes Activity code containing the Activity's finish() method to

execute or user swipes Activity from overview/recents

- Android interprets such a user action to mean the Activity's view-state is no longer required so it destroys it along with the Activity
  - Developers must follow Android's lead and not save non-view state (Android does not provide or call any callbacks to do this)
- Persistent data remains in its last updated state in non-volatile memory

# Saving/Restoring Activity Instance State

- **onSaveInstanceState(Bundle outState)**
  - Not called if user indicates they are done with the Activity
  - Must call super to let Android save view-state
  - Must code to manually save non-view state in Bundle
- **onCreate(Bundle savedInstanceState)**
  - Must call super to let Android restore view-state
  - Must code to manually restore previously saved non-view state from Bundle
    - Check if Bundle is non-null to see if there is an existing state that needs restoring else initialise the state with sensible default values
- **onRestoreInstanceState(Bundle savedInstanceState)**
  - Only called if the Bundle is non-null
    - So no need to check for this condition
    - Unlike onCreate you cannot set default values here
      - Because cannot test for null bundle
  - Must call super to let Android restore view-state it previously saved
  - Must code to manually restore previously saved non-view state from Bundle
- **A sub-class instance of Activity**
  - Will execute its parent's (i.e. Activity's) lifecycle callbacks at the appropriate times if you don't override them
  - If you do override however you must manually call their super if you want them to execute in addition to their override's code
- **The auto-saving/restoring of the view-state of an Activity is done in Activity's onCreate(...), onSaveInstanceState(...) and onRestoreInstanceState(...)**
  - The parameter of these three callbacks is a shared Bundle object that Android uses to save/restore an Activity's view-state
  - Programmers can also use this bundle to manually save and restore an Activity's non-view state
- **To auto-save/restore their state each View must have:**
  - A unique ID (to uniquely tag their saved state in the bundle)
  - Their saveEnabled property set to true (the default)
- **Android will auto-save the view-state of an Activity**
  - If it destroys it and the user can reasonably expect its state will be maintained
    - i.e. the destruction was not instigated by the user (e.g. System kill for resources) or not the user's fault (e.g. device reorientation)
    - The view-state will be auto-saved/restored using the Bundle object that is a shared parameter to an Activity sub class's inherited onSaveInstanceState, onCreate and

onRestoreInstanceState methods

- A programmer can override these methods and use the same Bundle to manually save/restore an Activity's non-view state
- If these methods are overridden in an Activity subclass and these overrides do not call their super Android's auto-save/restore of view-state will not occur

# Persistent Data

## Saving/Restoring Persistent Activity Data

- Options for saving data persistently
  - i.e. across all of an app's uses, i.e. Activity destroys
    - Saving Key-Value Sets
      - Using a shared preferences file for storing small amounts of information in key-value pairs. We will cover this next.
    - Saving Files
      - Using a basic file, such as storing long sequences of data that are generally read in order. Not covered in this unit.
    - Saving Data in SQL Databases
      - Learn to use an SQLite database to read and write structured data.
      - We will cover this in later weeks
  - All these options
    - Write to and read from a device's non-volatile memory

## Shared Preferences

If you have a relatively small collection of key-values that you'd like to save, you should use the SharedPreferences APIs. A SharedPreferences object points to a file containing key-value pairs and provides simple methods to read and write them. Each SharedPreferences file is managed by the framework and can be private or shared.  
~Android Documentation

### Create a shared preferences handler

```
SharedPreferences sharedPref=getSharedPreferences("file-name", Context.MODE_PRIVATE);
```

if you need just one shared preference file for your activity, you can use the getPreferences() method:

```
SharedPreferences sharedPref = getPreferences(Context.MODE_PRIVATE);
```

### Write to Shared Preferences

```
// initialise shared preference class variable to access Android's persistent storage
SharedPreferences sharedPref = getPreferences(Context.MODE_PRIVATE);

// use .edit function to access file using Editor variable
```

```
SharedPreferences.Editor editor = sharedPref.edit();

// some sample data one variable to represent key and another one value
int value = 23;
String key = "key";

// save key-value pairs to the shared preference file
editor.putInt(key, value);

// use editor.apply() to save data to the file asynchronously (in background without freezing the U
// // doing in background is very common practice for any File Input/Output operations
editor.apply();

// or
// editor.commit()
// commit try to save data in the same thread/process as of our user interface
```

## Read from Shared Preferences

```
SharedPreferences sharedPref = getPreferences(Context.MODE_PRIVATE);
int defaultValue = 0;
String key = "key";
int valueRestored = sharedPref.getInt(key,defaultValue);
```



**FIT2081**

**Mobile application  
development (MAD)**



# **Activity Lifecycle in Android & Data Persistence**

**Week 3**

Delvin Varghese

# Checklist

- My Android Studio is running well
- I can create **Buttons** and **Toasts**
- I can launch a new activity using **Intents**



# Learning objectives for today

## 1. The Android Lifecycle

- What is the lifecycle?
- Why it is crucial for Android coding..

## 2. Data persistence

- How to keep data in memory..
- Shared Preferences (for accessing data later)



# Recap: Mobile OSs

## Android is optimized for limited resources

- “Resource constrained” by the standards of modern desktop/laptop computers
- ensure that these limited resources are managed effectively

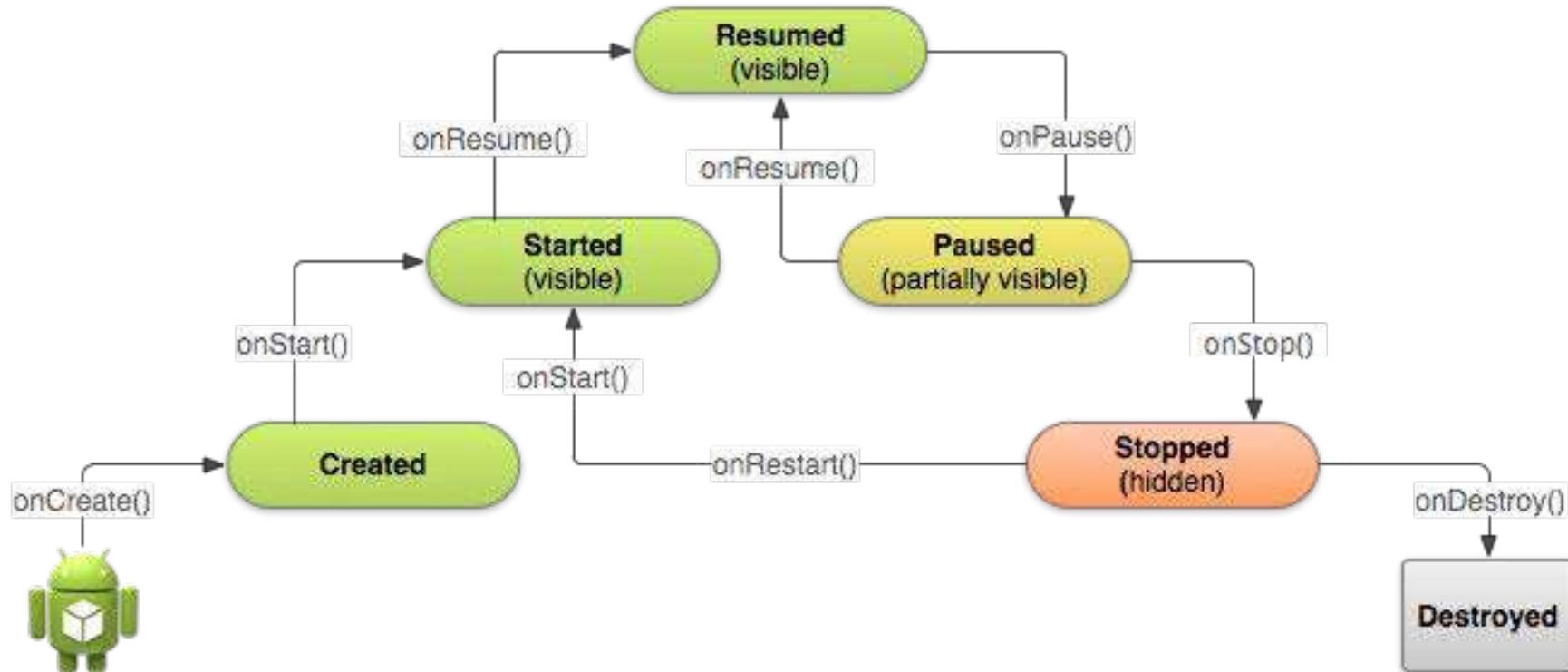
## Android OS as a terminator (supervisor)..

- full control over the lifecycle and state of both the processes in which the applications run
- Uses an algorithm to prioritise which application to terminate



# 1. The Android Lifecycle

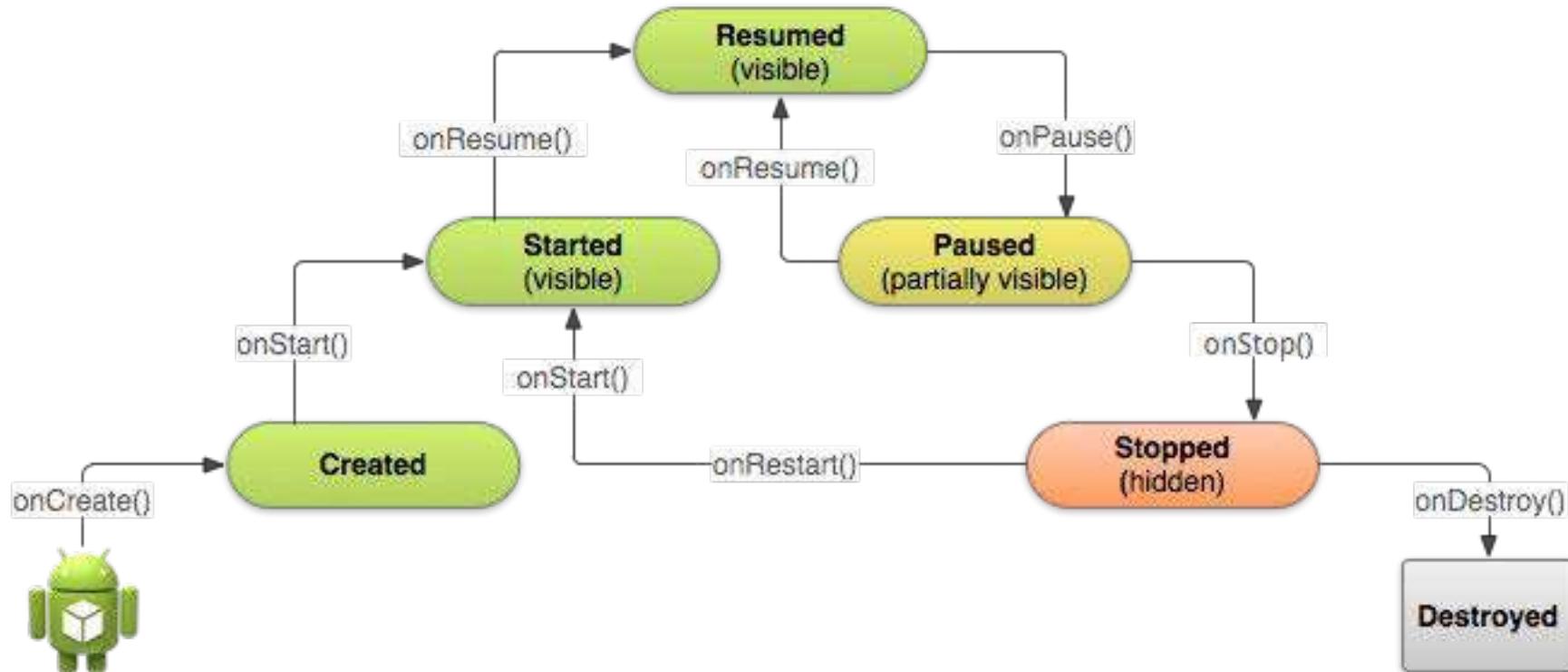
# Lets meet the Lifecycle....



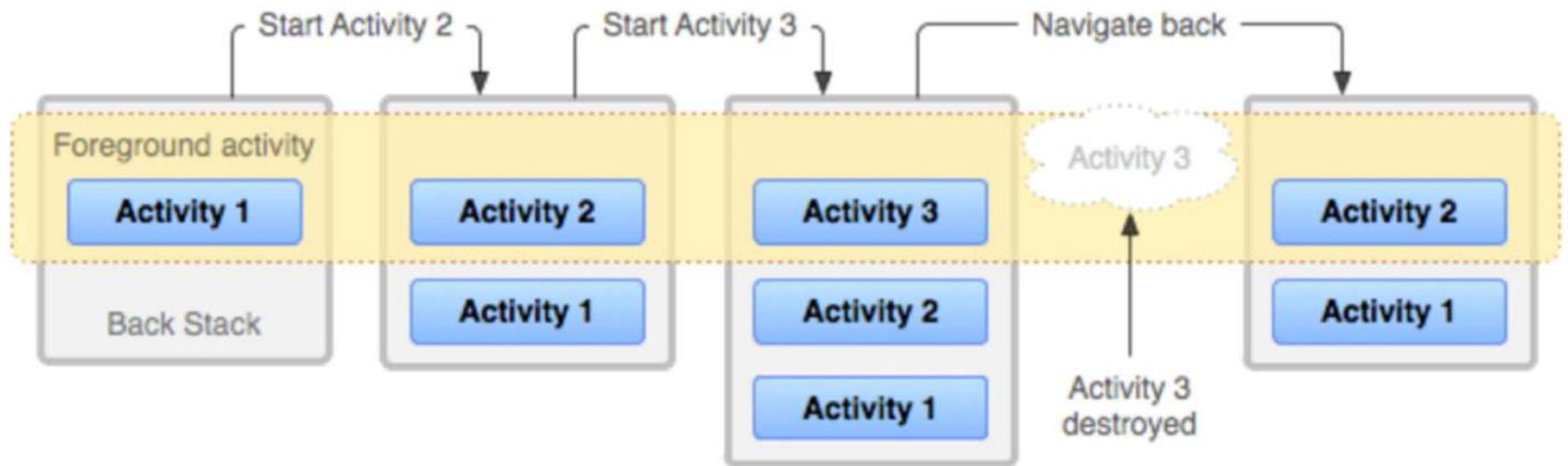


**LIVE CODING\_**

# Lets meet the Lifecycle....



# An App's journey through the lifecycle...





# QUESTION

Despite Activity being destroyed,  
how is the view data still being  
preserved?

## 2. Data Persistence

# Saving/Restoring Persistent Activity Data

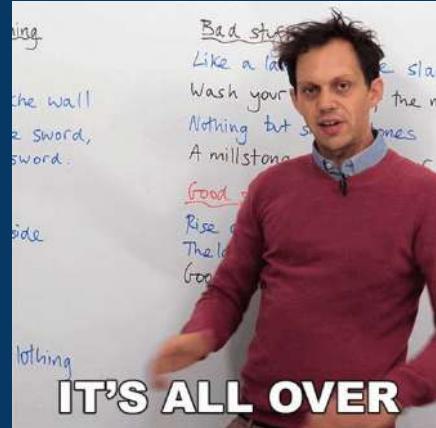
## Many options for saving data

- Persistence = accessing data across all of an app's uses i.e. even after Activity is destroyed
- Shared Preferences file (key-value pairs) - ideal for storing small amount of information
- SQL Databases to read/write structured data (covered later in the unit)





**LIVE CODING\_**



THE END\_

## Week 3 Lecture Notes: Android Lifecycle & Data Persistence in Shared Preferences

---

### Introduction & Overview

- **Welcome & Progress Update:**
  - Acknowledge students for reaching the 25% mark of the unit.
  - Encourage continued effort, as nine more weeks remain.
- **Focus of Week 3:**
  - Understanding **Android Activity Lifecycle**.
  - **Data Persistence** using **Shared Preferences**.
  - Preparing for **Assignment 1** by implementing persistent data storage.
- **Context of Data Persistence:**
  - Advanced applications use **databases** (covered in Week 6 & 7).
  - This week focuses on **storing user data beyond a session** using **Shared Preferences**.

---

## 1. Android Lifecycle

### Why is the Lifecycle Important?

- Unlike other operating systems, **Android actively manages memory**.
- **Applications don't control their lifecycle; Android does**.
- Helps keep devices **efficient, cool, and responsive**.
- Developers must handle lifecycle events to **avoid data loss or app crashes**.

### Lifecycle Phases & Callbacks

Lifecycle Method	Description
<code>onCreate()</code>	Called when the activity is first created. Initialize UI and variables.
<code>onStart()</code>	Activity is visible but not interactive yet.
<code>onResume()</code>	Activity is fully interactive.
<code>onPause()</code>	Another activity comes in front. Save temporary states.
<code>onStop()</code>	Activity is no longer visible. Can be killed if memory is needed.

<b>onDestroy()</b>	Final cleanup before activity is removed from memory.
--------------------	---

## User Actions & Lifecycle Effects

- **Pressing Home Button:** Moves app to **onPause()**, then **onStop()**.
- **Switching Apps:** Current app moves to **onPause()**.
- **Pressing Back Button:** Calls **onDestroy()**, completely **removing the activity from memory**.
- **Screen Rotation:** Triggers **onDestroy()** and recreates the activity.

## Demo: Tracking Lifecycle

- Use **Logcat** to print logs in each lifecycle method.
  - Show students how rotation affects activity states.
- 

## 2. Persisting Data Beyond App Session

### What is Data Persistence?

- Allows **saving user data** beyond the session.
- Important when users:
  - Close and reopen the app.
  - Rotate the device.
  - Navigate between activities.

### Options for Data Storage

1. **Shared Preferences** (for simple key-value storage).
2. **Internal Storage** (saving files within the app's storage).
3. **SQLite/Room Database** (for complex relational data – covered in Week 6 & 7).
4. **Cloud Storage** (Firebase, APIs – covered in later weeks).

### Shared Preferences in Android

- Stores **small amounts of data persistently** (e.g., user settings, app preferences).
  - Uses a **key-value pair** system.
  - Data is **private** to the app but persists after closing.
- 

## 3. Implementing Shared Preferences

### Step 1: Saving Data

```
val sharedpreferences = getsharedpreferences("MyAppPrefs", Context.MODE_PRIVATE)
val editor = sharedpreferences.edit()
editor.putString("USERNAME", "JohnDoe")
editor.putInt("AGE", 25)
editor.apply() // Saves asynchronously
```

- **getSharedPreferences**: Retrieves the file.
- **edit()**: Opens the editor for modification.
- **apply() vs. commit()**:
  - **apply()**: Saves asynchronously (background thread).
  - **commit()**: Saves immediately (blocks execution).

## Step 2: Retrieving Data

```
val sharedpreferences = getsharedpreferences("MyAppPrefs", Context.MODE_PRIVATE)
val username = sharedpreferences.getstring("USERNAME", "DefaultUser")
val age = sharedpreferences.getint("AGE", 0)
```

- **Provide a default value** in case the key does not exist.

## Step 3: Clearing Data

```
editor.clear().apply() // Clears all preferences
editor.remove("USERNAME").apply() // Removes a specific key
```

---

## 4. Live Coding Example: Formula 1 Fan App

**Objective:** Create an app where users:

- Enter **favorite F1 team**.
- Select **event date** (using a **DatePicker**).
- Adjust **hype level** (using a **Slider**).
- Save preferences using **Shared Preferences**.
- View saved data in a **Dialog Box**.

### UI Components

Component	Purpose
TextField	Input favorite F1 team
Button	Opens <b>DatePicker</b> to select event date
Slider	Sets excitement level (0-100)

<b>Save Button</b>	Saves inputs to Shared Preferences
<b>Show Summary Button</b>	Displays stored values in a <b>Dialog</b>

## Step 1: UI Layout

```
Column(
    modifier = Modifier.fillMaxSize().padding(16.dp),
    horizontalAlignment = Alignment.CenterHorizontally
) {
    var teamName by remember { mutableStateOf("") }
    var eventDate by remember { mutableStateOf("Select Date") }
    var hypeLevel by remember { mutableStateOf(50f) }

    TextField(value = teamName, onValueChange = { teamName = it }, label = { Text("Enter
F1 Team") })

    Button(onClick = { /* Show DatePicker */ }) {
        Text(text = eventDate)
    }

    Slider(value = hypeLevel, onValueChange = { hypeLevel = it }, valueRange = 0f..100f)

    Button(onClick = { /* Save values to Shared Preferences */ }) {
        Text("Save Preferences")
    }

    Button(onClick = { /* Show Summary Dialog */ }) {
        Text("Show Summary")
    }
}
```

## Step 2: Implementing Shared Preferences

```
val sharedPreferences = context.getSharedPreferences("Formula1Prefs",
Context.MODE_PRIVATE)
val editor = sharedPreferences.edit()

editor.putString("TEAM_NAME", teamName)
editor.putString("EVENT_DATE", eventDate)
editor.putFloat("HYPE_LEVEL", hypeLevel)
editor.apply()
```

## Step 3: Retrieving Data in Dialog

```
val savedTeam = sharedPreferences.getString("TEAM_NAME", "Not Selected")
```

```

val savedDate = sharedPreferences.getString("EVENT_DATE", "No Date")
val savedHype = sharedPreferences.getFloat("HYPE_LEVEL", 50f)

AlertDialog(
    onDismissRequest = { /* Close Dialog */ },
    title = { Text("F1 Team Details") },
    text = {
        Column {
            Text("Team: $savedTeam")
            Text("Event Date: $savedDate")
            Text("Hype Level: ${savedHype.toInt()}")  

        }
    },
    confirmButton = {
        Button(onClick = { /* Close Dialog */ }) { Text("OK") }
    }
)

```

---

## 5. Testing & Debugging

### Test Cases

Scenario	Expected Outcome
Enter and save a team name	Should persist after app restart
Select a date and reopen app	Should display selected date
Adjust hype level and reopen	Should retain the selected level
Press "Show Summary"	Should display saved values in Dialog

### Common Bugs & Fixes

Issue	Fix
Values reset after restart	Ensure <code>apply()</code> is used
Incorrect date displayed	Ensure correct format conversion
Slider value not persisting	Convert <code>Float</code> to <code>Int</code> if needed

---

## 6. Summary & Next Steps

### Key Takeaways

- **Android manages activity lifecycle automatically.**
- **Shared Preferences** allows storing **small, simple** data persistently.
- Lifecycle awareness is **critical for efficient app design**.

### Next Week (Week 4 Preview)

- **Kotlin OOP concepts.**
  - **Handling lists dynamically in Jetpack Compose.**
  - **LazyColumn vs RecyclerView.**
- 

## Assignment & Lab Preparation

- **Lab Exercises:**
    - Implement Shared Preferences in a small app.
    - Store and retrieve user preferences.
    - Handle configuration changes.
  - **Assignment 1:**
    - Begin planning UI elements and data persistence.
    - Choose an **app concept** that requires **saving user preferences**.
- 

## End of Lecture

- Encourage **questions on Ed Forum**.
- Upload recorded lecture & sample code.

# Lab: Input Validation | Modals | Shared Preferences | CSV Files

---

## Overview

### Introduction

In today's mobile app ecosystem, user interaction is paramount. This lab introduces essential UI interaction components that form the foundation of engaging mobile applications. We'll build practical interfaces that handle user input, validate data, present information through modals, persist preferences, and interact with data files—all skills crucial for your Assignment 1 NutriTrack application.

### Learning Objectives

By the end of this lab, you'll be able to:

- Implement comprehensive input validation for login screens
- Create and manage modal dialogs to collect user information
- Save and retrieve user preferences using Android's SharedPreferences
- Implement date and time pickers for user data collection
- Display data using interactive components (sliders, progress bars)
- Read and process data from CSV files in Android applications



As usual, concepts relating to these topics will be explained further in the weekly lecture, so make sure to watch it!

### Why These Skills Matter

These components form the building blocks of sophisticated mobile applications:

- **Input validation** ensures data integrity and enhances user experience
- **Modal dialogs** provide focused interfaces for essential user interactions
- **Data persistence** allows your app to maintain state across sessions
- **Data reading capabilities** enable your app to work with external information

### Connection to Assignment 1

Today's lab directly supports the NutriTrack app you'll develop for Assignment 1. You'll practice:

- Building login screens with validation (similar to NutriTrack's Login Screen)
- Implementing user input forms (like NutriTrack's Food Intake Questionnaire)
- Saving user preferences (required for maintaining user state in NutriTrack)
- Reading from CSV files (essential for loading user IDs and food quality scores)

## Lab Structure

1. **Input Validation:** Create a secure login screen with email and password validation
2. **Modal Dialogs:** Implement popup interfaces for focused user interactions
3. **User Preferences & UI Components:** Save data with SharedPreferences and implement date/time pickers
4. **CSV Data Processing:** Read and parse structured data from CSV files
5. **Challenge Exercises:** Extend your learning with practical enhancements



**Topics we have covered in previous weeks will only be covered in passing. If you are struggling with any topic from previous weeks, please go through W1 and W2 material and if you are still having issues, reach out to your peers/tutors!**

## Getting Started

Ensure Android Studio is open and create a new Compose project. We'll be building multiple independent components that you can later adapt for your assignment work.

Let's strengthen your mobile development toolkit with these essential skills!

---

## 1. Input Validation

### Learning Objectives

By the end of this section, you will be able to:

- Implement real-time input validation for text fields
- Create validation functions for common data patterns
- Display contextual error messages to guide users
- Implement a secure login workflow

### Expected Output

Here is the expected output:

10:59



## Login

Email

Password

Login

# What is Data Validation?

Before coding, what is data validation?

Data validation ensures that the data entered by the user or received from external sources meets specific criteria and is in the expected format before it is processed or stored. This helps maintain the integrity and accuracy of the app's data and prevents errors or crashes.

Data validation is the process of ensuring that user input meets specific criteria before it's processed by your application. This serves several critical purposes:

- **Security:** Prevents malicious input that could compromise your app
- **Data Integrity:** Ensures the information stored is accurate and usable
- **User Experience:** Provides immediate feedback to help users enter correct information

## Types of Validation

- **Format Validation:** Checks if data follows a specific pattern (email, phone number)
- **Range Validation:** Verifies values fall within acceptable limits
- **Presence Validation:** Ensures required fields are not empty
- **Consistency Validation:** Confirms data is logical (e.g., start date before end date)

To discuss this topic practically, let's develop a login screen and validate its input fields.

## Building Your Login Screen

**Step A:** Let's implement a login screen with email and password validation. We'll use Jetpack Compose to create the UI and add validation logic.

1. Create a new Compose project in Android Studio
2. Comment out the Greeting/GreetingPreview code and create a new composable for the LoginScreen like we did last week



**Step B:** Next, let's add four state variables for email, password and error flags. Remember: We need state variables to track the user's input and validation status:

```
// Email and password state
var email by remember { mutableStateOf("") }
var password by remember { mutableStateOf("") }

// Error flags for validation
var emailError by remember { mutableStateOf(false) }
var passwordError by remember { mutableStateOf(false) }
```

Before you proceed, check if you understand: what is the purpose of 2 variables each for email and password? Once you've had a guess, see the answer below:

► Expand

### Step C: Create the Email Input Field

Let's add an email field with validation:

First, let's add a column to list the UI elements vertically. Inside the column, let's add an OutlinedTextField for the email address.

```
1     OutlinedTextField(
2         value = email,
3         onValueChange = {
4             email = it
5             //check if the email is valid every time the value changes
6             emailError = !isValidEmail(it)
7         },
8         label = { Text("Email") },
9         keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Email),
10        //if isError is true, the UI provides a visual
11        // indicator of a validation error state.
12        isError = emailError,
13        modifier = Modifier.fillMaxWidth(),
14        singleLine = true
15    )
16    // if email error , display error message
17    if (emailError) {
18        Text(
19            text = "Invalid email",
20            color = MaterialTheme.colorScheme.error,
21            modifier = Modifier.padding(start = 16.dp, top = 4.dp)
22        )
23    }
```

In the code above, note the following **key points**:

- The boolean variable 'emailError' is re-evaluated every time the email address changes (see line 6).
- The current value of the email (it) is passed to another boolean function (isValidEmail) to validate its value (see the function below).
- The isError attribute is set to emailError to provide a visual indicator if there is an error (if the value is true)
- In addition to the visual indicator, the screen shows an error message "Invalid email" in a Text element (see lines 17-23)

#### Step D: Email Validation Function

This function checks if the email follows a valid format.



Note: this is a new function declared using the 'fun' Kotlin keyword. So you need to declare this in the same scope (but outside of) as LoginScreen() function.

```
1 /**
2  * Validates if the provided email is in a valid format.
3  *
4  * @param email The email string to validate.
5  * @return True if the email is valid, false otherwise.
6  */
7 fun isValidEmail(email: String): Boolean {
8     return android.util.Patterns.EMAIL_ADDRESS.matcher(email).matches()
9 }
```

codesnap.dev

You may have already encountered Regex in previous programming. But if you haven't the following explanation will help.

▶ Expand

#### Step E: Add Password Field with Validation

Just like we implemented the username field, we will do the same for the password.

```
1     OutlinedTextField(
2         value = password,
3         onValueChange = {
4             //this code gets executed when the
5             // user types in the password field
6             password = it
7             //check if the password is less than 6 characters
8             //you can add more constraints here
9             passwordError = it.length < 6
10        },
11        label = { Text("Password") },
12        visualTransformation = PasswordVisualTransformation(),
13        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Password),
14        //if isError is true, the UI provides a visual
15        // indicator of a validation error state.
16        isError = passwordError,
17        modifier = Modifier.fillMaxWidth(),
18        singleLine = true
19    )
20    //if password error, display error message
21    if (passwordError) {
22        Text(
23            text = "Password must be at least 6 characters",
24            color = MaterialTheme.colorScheme.error,
25            modifier = Modifier.padding(start = 16.dp, top = 4.dp)
26        )
27    }

```

codesnap.dev

Do you want to implement password validation? How would you do it?

▶ Expand

#### Step F: Add Login Button with Validation

Lastly, the login button should ensure both values are valid before proceeding to the next screen.

```
1  Button(  
2      onClick = {  
3          //Before proceeding with the login,  
4          // you need to ensure that both email and password are valid.  
5          emailError = !isValidEmail(email)  
6          passwordError = password.length < 6  
7          if (!emailError && !passwordError) {  
8              // For example, navigate to the next screen or call an API  
9          }  
10         },  
11         modifier = Modifier.fillMaxWidth()  
12     ) {  
13         Text("Login")  
14     }
```

[codesnap.dev](https://codesnap.dev)

Therefore, the activity in one piece:

```
1 package com.fit2081.week3validation
2
3 import android.os.Bundle
4 import androidx.activity.ComponentActivity
5 import androidx.activity.compose.setContent
6 import androidx.activity.enableEdgeToEdge
7 import androidx.compose.foundation.layout.*
8 import androidx.compose.foundation.text.KeyboardOptions
9 import androidx.compose.material3.*
10 import androidx.compose.runtime.*
11 import androidx.compose.ui.Alignment
12 import androidx.compose.ui.Modifier
13 import androidx.compose.ui.text.input.KeyboardType
14 import androidx.compose.ui.text.input.PasswordVisualTransformation
15 import androidx.compose.ui.unit.dp
16 import androidx.compose.ui.unit.sp
17 import com.fit2081.week3validation.ui.theme.Week3ValidationTheme
18
19 // Main activity that hosts the login screen
20 class MainActivity : ComponentActivity() {
21     override fun onCreate(savedInstanceState: Bundle?) {
22         super.onCreate(savedInstanceState)
23         enableEdgeToEdge()
24         setContent {
25             Week3ValidationTheme {
26                 Surface(
27                     modifier = Modifier.fillMaxSize(),
28                     color = MaterialTheme.colorScheme.background
29                 ) {
30                     LoginScreen()
31                 }
32             }
33         }
34     }
35 }
```

```
32         }
33     }
34 }
35 */
36 /**
37 * Composable function for the Login Screen UI.
38 */
39 @Composable
40 fun LoginScreen() {
41
42     // State variables for email, password, and error flags
43     var email by remember { mutableStateOf("") }
44     var password by remember { mutableStateOf("") }
45     var emailError by remember { mutableStateOf(false) }
46     var passwordError by remember { mutableStateOf(false) }
47     // Column layout for the entire screen
48     Column(
49         modifier = Modifier
50             .fillMaxSize()
51             .padding(16.dp),
52         verticalArrangement = Arrangement.Center,
53         horizontalAlignment = Alignment.CenterHorizontally
54     ) {
55         // Login title
56         Text(
57             text = "Login",
58             fontSize = 24.sp,
59             modifier = Modifier.padding(bottom = 16.dp)
60         )
61         // Email TextField area
62         OutlinedTextField(
63             value = email,
64             onValueChange = {
65                 email = it
66                 //check if the email is valid every time the value changes
67                 emailError = !isValidEmail(it)
68             },
69             label = { Text("Email") },
70             keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Email),
71             //if isError is true, the UI provides a visual
72             // indicator of a validation error state.
73             isError = emailError,
74             modifier = Modifier.fillMaxWidth(),
75             singleLine = true
76         )
77         // if email error , display error message
78         if (emailError) {
79             Text(
80                 text = "Invalid email",
81                 color = MaterialTheme.colorScheme.error,
82                 modifier = Modifier.padding(start = 16.dp, top = 4.dp)
83             )
84         }
85     }
86
87     // Vertical spacing
88     Spacer(modifier = Modifier.height(16.dp))
89     // Password TextField
90     OutlinedTextField(
91         value = password,
```

```
92     onChange = {
93         //this code gets executed when the
94         // user types in the password field
95         password = it
96         //check if the password is less than 6 characters
97         //you can add more constraints here
98         passwordError = it.length < 6
99     },
100    label = { Text("Password") },
101    visualTransformation = PasswordVisualTransformation(),
102    keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Password),
103    //if isError is true, the UI provides a visual
104    // indicator of a validation error state.
105    isError = passwordError,
106    modifier = Modifier.fillMaxWidth(),
107    singleLine = true
108 )
109 //if password error, display error message
110 if (passwordError) {
111     Text(
112         text = "Password must be at least 6 characters",
113         color = MaterialTheme.colorScheme.error,
114         modifier = Modifier.padding(start = 16.dp, top = 4.dp)
115     )
116 }
117
118 //spacing
119 Spacer(modifier = Modifier.height(24.dp))
120 // Login Button area
121 Button(
122     onClick = {
123         //Before proceeding with the login,
124         // you need to ensure that both email and password are valid.
125         emailError = !isValidEmail(email)
126         passwordError = password.length < 6
127         if (!emailError && !passwordError) {
128             // For example, navigate to the next screen or call an API
129         }
130     },
131     modifier = Modifier.fillMaxWidth()
132 ) {
133     Text("Login")
134 }
135 }
136 }
137 /**
138 * Validates if the provided email is in a valid format.
139 *
140 * @param email The email string to validate.
141 * @return True if the email is valid, false otherwise.
142 */
143 fun isValidEmail(email: String): Boolean {
144     return android.util.Patterns.EMAIL_ADDRESS.matcher(email).matches()
145 }
```

## 2. Dealing with Modals: Creating Focused User Interactions

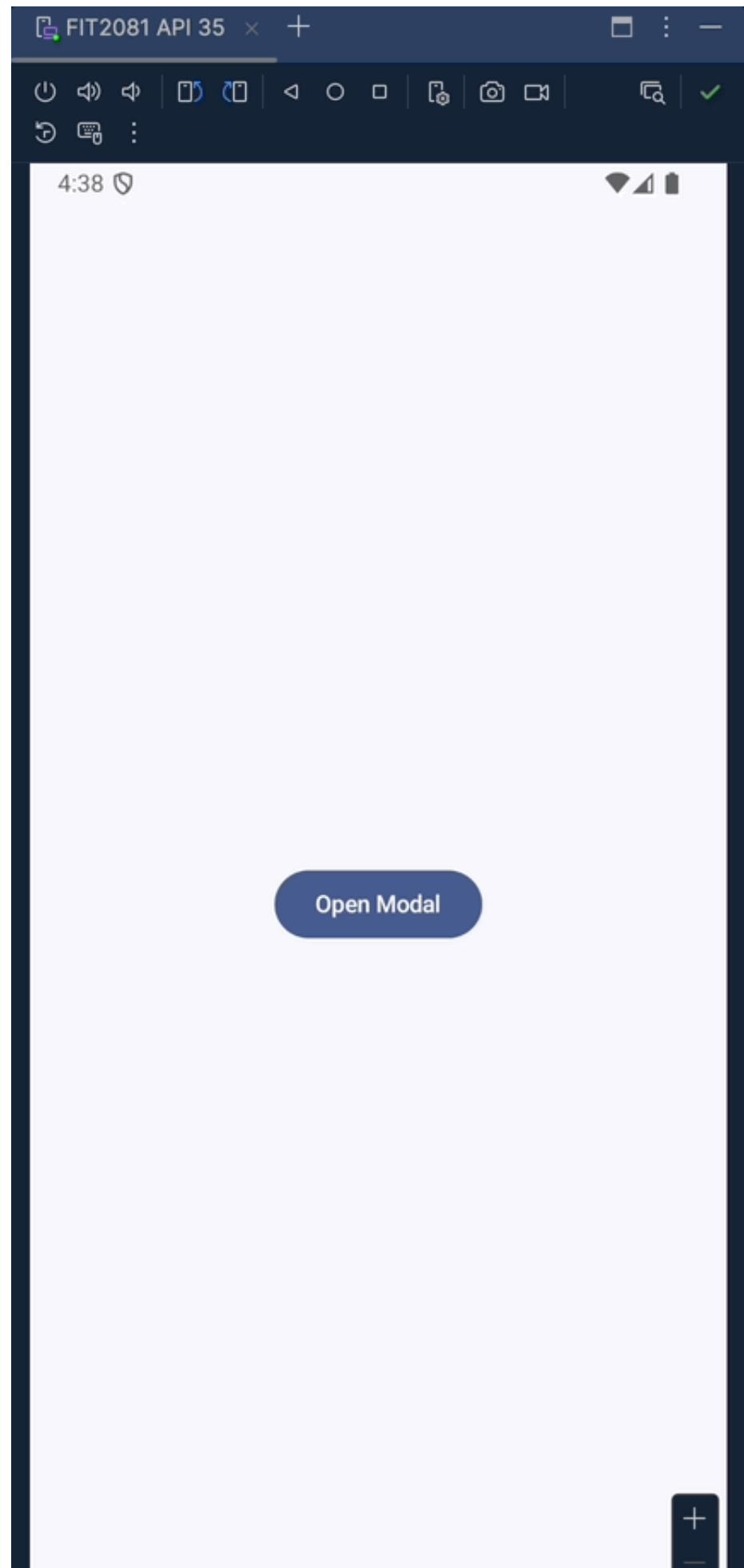
### Learning Objectives

By the end of this section, you will be able to:

- Create and display modal dialogs in Jetpack Compose
- Implement callback functions to return data from modals
- Use higher-order functions in Kotlin
- Manage modal visibility with state variables
- Cancel and dismiss a modal

### Expected Output

See the expected output below (clicking a button that opens a Modal, and we pass back the user entry when we return to the screen with the button).



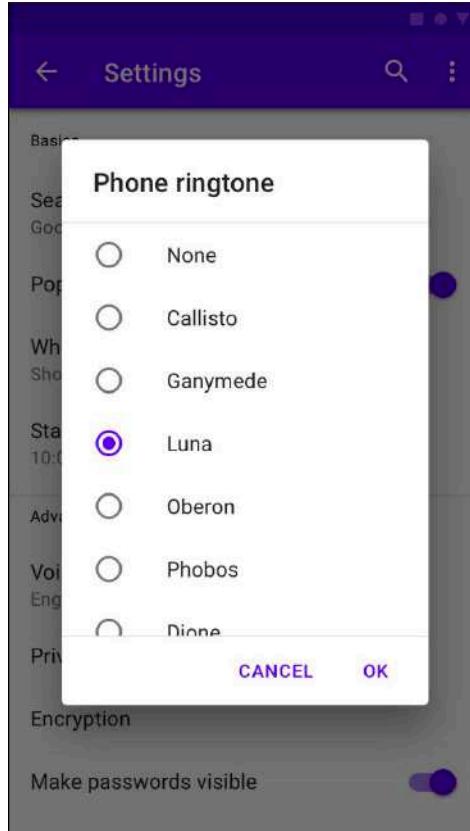
# What is a Modal in mobile applications?

Modals in mobile applications are user interface (UI) components that display content in a temporary, focused window or dialogue box, overlaying the main screen. They are often used to capture the user's attention, request input, or present vital information without requiring navigation away from the current screen. Modals typically darken or blur the background to highlight the modal content and prevent interaction with the underlying interface until the modal is closed.

They're essential for:

- **Focused Input:** Collecting specific information without navigation
- **Confirmations:** Asking users to confirm important actions
- **Information Display:** Showing critical messages that require attention
- **Contextual Actions:** Providing options relevant to the current context

Here is an example of a modal in the Settings screen helping the user pick a ringtone.



## How to add and use a modal?

### Step 1: Create a Button that Shows the Modal

First, let's create a function that displays a button and manages the modal dialog by showing the modal (AlertDialog).

```
/*
 * ShowButtonAndModal composable function to
 * display a button that opens an AlertDialog.
 * @param onConfirm Function1<String, Unit>: A callback
 * function to be invoked when the user confirms the input in the dialog.
 */
@Composable
fun ShowButtonAndModal(onConfirm: (String) → Unit) {
    // Your code goes here
}
```

codesnap.dev

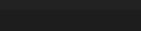
As you can see in the code above, the function ShowButtonAndModal accepts one parameter: a callback function (onConfirm). The caller of the ShowButtonAndModal function implements this callback function, which is invoked when the user confirms the modal.



We'll be storing ALL the logic for the Button + Modal inside this Composable function (ShowButtonAndModal)  
- so any button related logic in Steps 2,3,4

## Step 2: Define State Variables

Add a boolean variable to control the model's visibility, like the options menu we covered last week. You also need some variables to hold the values of your input fields. In our case, we need an extra string variable.

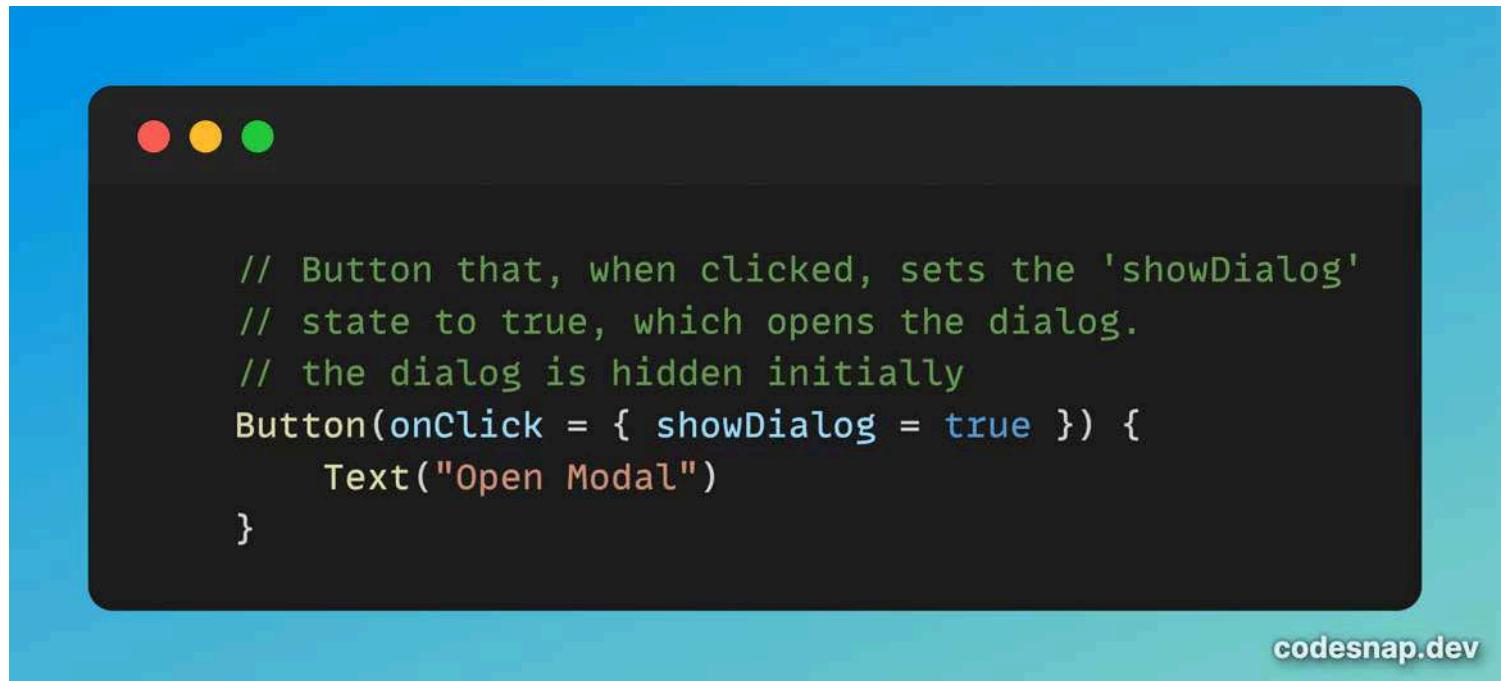


```
// State to control the visibility of the AlertDialog.
var showDialog by remember { mutableStateOf(false) }
// State to hold the value entered in the text field.
var textFieldValue by remember { mutableStateOf("") }
```

codesnap.dev

## Step 3: Add the Button to Show the Modal

A button that sets the boolean variable to true is added later.



If the showDialog boolean variable is true, the user wants to open the modal (the dialog).

## Step 4: Implement the AlertDialog Modal

We will use the UI element **AlertDialog** ([Resource: Android Docs](#)) as a base for our modal. add the following code from the code screenshot for AlertDialog below the Button() element i.e. in the following style:

```
Button(onClick = { showDialog = true }) {  
    Text("Open Modal")  
}  
if (showDialog) {  
    AlertDialog(  
        // Code  
    )  
}
```

```

1   if (showDialog) {
2     AlertDialog(
3       //switch the visibility of the dialog to false when the user dismisses it
4       onDismissRequest = { showDialog = false },
5       title = { Text("Enter Your Name") },
6       text = {
7         Column {
8           OutlinedTextField(
9             value = textFieldValue,
10            onValueChange = { textFieldValue = it },
11            label = { Text("Your name") },
12            modifier = Modifier.fillMaxWidth()
13          )
14          Spacer(modifier = Modifier.height(16.dp))
15          if (textFieldValue.isNotEmpty()) {
16            //This text will be shown inside the dialog (Modal)
17            Text("Your name is: $textFieldValue")
18          }
19        }
20      },
21      confirmButton = {
22        Button(onClick = {
23          showDialog = false
24          //if the user clicks on the confirm button,
25          //call the callback function 'onConfirm' with the entered text
26          //the implementation of the callback function is in the onCreate function
27          onConfirm(textFieldValue)
28        }) {
29          Text("Confirm")
30        }
31      },
32      dismissButton = {
33        //if the user clicks on the dismiss button,
34        ////
35        Button(onClick = { showDialog = false }) {
36          Text("Cancel")
37        }
38      }
39    )
40  }
41 }

```

In the code above, note the following:

- If the user dismisses the dialog, we need only switch the boolean flag back to false (see line 4).
- The Text field (line 17) shows the input text inside the dialog.
- The confirmButton is a button that has two tasks:
  - Dismiss the dialog by setting false to the boolean variable showDialog (see line 24)
  - Call the callback function, which is implemented by the caller, to pass the value from the modal back to the caller (see line 28)
- The dismissButton is simply used to dismiss the dialog by setting false to the variable 'showDialog' (see line 36)

## Step 5: Complete the UI for displaying name from Modal

Inside onCreate function, we should add a column to place the modal's button and the text item vertically. The Text field is used to show the data retrieved from the modal.

```
1  override fun onCreate(savedInstanceState: Bundle?) {
2      super.onCreate(savedInstanceState)
3      enableEdgeToEdge()
4
5      // Set the content view to the composable layout.
6      setContent {
7          Week3modalsTheme {
8              Scaffold { innerPadding -
9                  Column(
10                      modifier = Modifier
11                          .fillMaxSize()
12                          .padding(innerPadding),
13                      horizontalAlignment = Alignment.CenterHorizontally,
14                      verticalArrangement = Arrangement.Center,
15                  ) {
16                      //this variable is used to store
17                      // the value entered in the modal (AlertDialog)
18                      var textFieldValue by remember { mutableStateOf("") }
19                      //We are passing an anonymous function to the ShowButtonAndModal composable
20                      //This anonymous function will be called
21                      // when the user confirms the input in the dialog
22                      ShowButtonAndModal(onConfirm = {
23                          textFieldValue = it
24                      })
25                      Spacer(modifier = Modifier.height(40.dp))
26                      //if the textFieldValue is not empty,
27                      // we display the text "Hello $textFieldValue"
28                      if (textFieldValue.isNotEmpty())
29                          Text("Hello $textFieldValue", fontSize = 40.sp)
30
31                  }
32              }
33          }
34      }
35  }
```

codesnap.dev

As you can see in the code above (lines 22-24), we are calling the function ShowButtonAndModal and passing the function 'onConfirm' as a parameter to be invoked when the user taps the confirm button. If this method is invoked, the state of textFieldValue will be changed; hence, the 'if' statement (lines 28-30) will show the text.

## Check Your Understanding

1. What is the purpose of a modal dialog in mobile applications?
2. How do [higher-order functions](#) enable data to be passed back from modals?
3. What function handles the situation when a user taps outside a dialog?
4. How would you modify the modal to return multiple values?

---

### 3. Shared Preferences, Date and Time pickers, Slider and Progress Bar

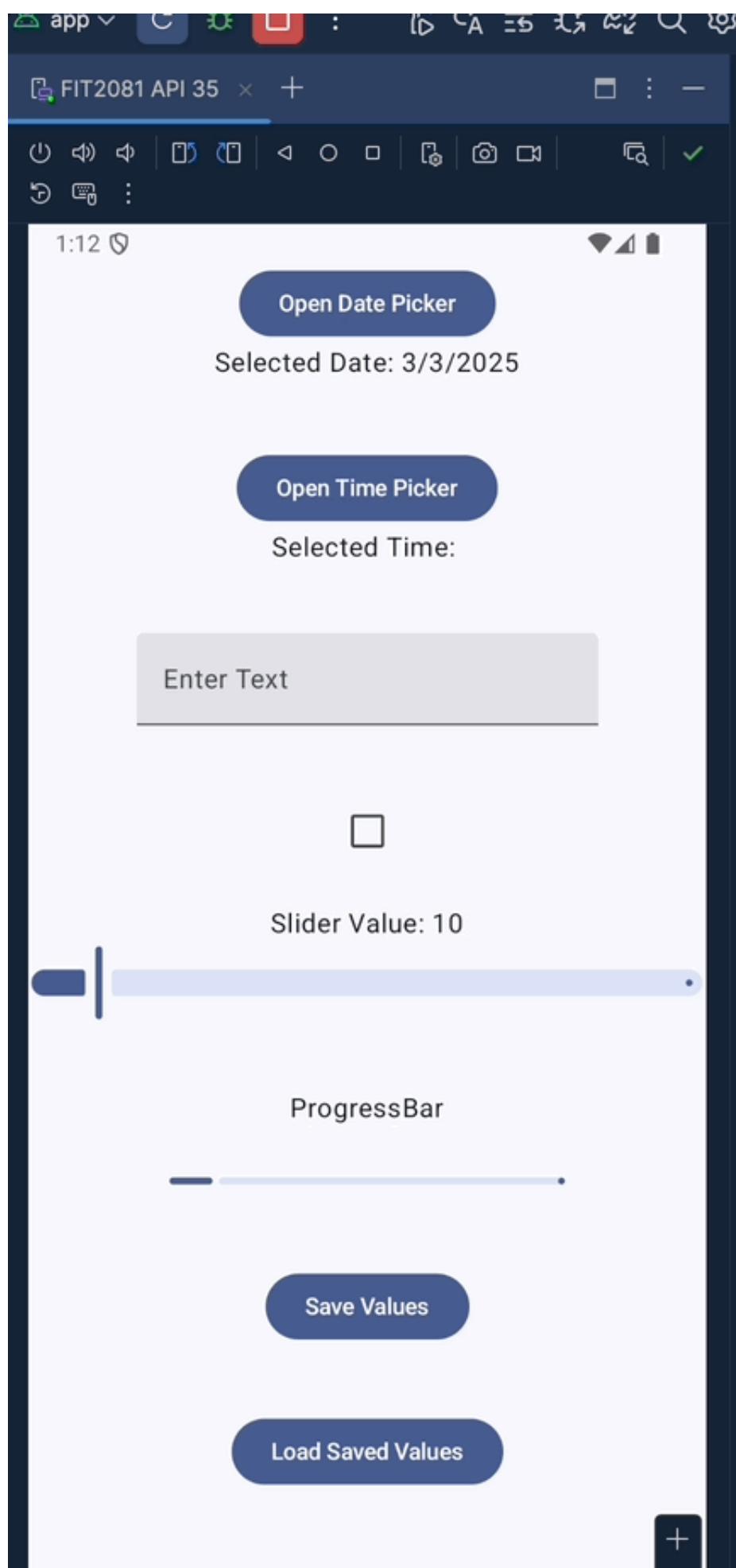
#### Learning Objectives

By the end of this section, you will be able to:

- Save and retrieve data using Android's SharedPreferences
- Implement date and time pickers for user data collection
- Create interactive UI components like sliders and progress bars
- Build a complete settings screen with persistent user preferences

#### Expected Output

Here is the expected output:



# What is SharedPreferences?

SharedPreferences is Android's lightweight key-value storage system designed for saving small amounts of data that need to persist across app sessions. It's perfect for:

- User preferences and settings
- Login credentials (with appropriate security considerations)
- UI state and configuration
- Small amounts of application data

Think of SharedPreferences as a persistent dictionary that survives app restarts and device reboots.

## Key Characteristics

- **Persistent Storage:** Data remains available even after app closure
- **Key-Value Pairs:** Data is stored as simple key-value associations
- **Simple API:** Easy to use with straightforward get/put methods
- **Limited Scope:** Best for small data, not large or complex structures
- **Type Safety:** Supports various primitive data types (strings, integers, booleans, etc.)

**i** Key-Value Pairs: Data is stored as key-value pairs, much like a dictionary or map. This allows for easy retrieval of specific values using their corresponding keys.

=====

## Building a Preferences Screen

Let's create a screen that collects user preferences and saves them:

### Step 1: Define State Variables

First, we'll define state variables for our different UI components inside `setContent{}` like we did in previous weeks:

```
1     setContent {
2         val mContext = LocalContext.current
3
4         val mDate = remember { mutableStateOf("") }
5         val mTime = remember { mutableStateOf("") }
6
7         val mTextFieldValue = remember { mutableStateOf("") }
8         val mCheckBoxState = remember { mutableStateOf(false) }
9
10        // Call the function to return a TimePickerDialog
11        var mTimePickerDialog = TimePickerFun(mTime)
12        var mDatePickerDialog = DatePickerFun(mDate)
13
14        var sliderValue by remember { mutableStateOf(10f) }
```

codesnap.dev

In the code above, the first two variables (lines 4 and 5) are for Date and Time pickers, respectively. Then, we added two more variables for the text field and the checkbox, which we will save in the shared preferences.

The statements in lines 11 and 12 call two custom methods (we will create them ourselves below) to create the required date and time pickers.

## Step 2: Create Date Picker

Now let's implement a date picker that updates our selectedDate state:

In the code below,

- The function takes as input the state variable 'mDate' we declared earlier
- The function returns the DatePickerDialog (see lines 2 and 20)
- Line 27 sets a new value to 'mDate' variable if the user selects and confirms the new date

```
1 @Composable
2 fun DatePickerFun(mDate: MutableState<String>): DatePickerDialog {
3     // Get the current context
4     val mContext = LocalContext.current
5
6     // Initialize variables to hold the current year, month, and day
7     val mYear: Int
8     val mMonth: Int
9     val mDay: Int
10
11    // Get a calendar instance
12    val mCalendar = Calendar.getInstance()
13
14    // Extract the current year, month, and day from the calendar
15    mYear = mCalendar.get(Calendar.YEAR)
16    mMonth = mCalendar.get(Calendar.MONTH)
17    mDay = mCalendar.get(Calendar.DAY_OF_MONTH)
18
19    // Create and return a DatePickerDialog
20    return DatePickerDialog(
21        // Context
22        // Listener to be invoked when the date is set
23        // Initial year, month, and day
24
25        mContext,
26        { _: DatePicker, mYear: Int, mMonth: Int, mDayOfMonth: Int →
27            mDate.value = "$mDayOfMonth/${mMonth + 1}/$mYear"
28        }, mYear, mMonth, mDay
29    )
30 }
```

codesnap.dev

## Step 3: Create Time Picker

Similar to the date picker, let's implement a time picker:

Like the Date Picker, this function:

- It takes as the input state of 'mTime' we declared earlier.
- Updates the value of the 'mTime' if the user selects and confirms.

```
1 @Composable
2 fun TimePickerFun(mTime: MutableState<String>): TimePickerDialog {
3     // Get the current context
4     val mContext = LocalContext.current
5     // Get a calendar instance
6     val mCalendar = Calendar.getInstance()
7
8     // Get the current hour and minute
9     val mHour = mCalendar.get(Calendar.HOUR_OF_DAY)
10    val mMinute = mCalendar.get(Calendar.MINUTE)
11
12    // Set the calendar's time to the current time
13    mCalendar.time = Calendar.getInstance().time
14    // Return a TimePickerDialog
15    return TimePickerDialog(
16        // Context
17        // Listener to be invoked when the time is set
18        // Initial hour and minute
19        // Whether to use 24-hour format
20
21        mContext,
22        { _, mHour: Int, mMinute: Int →
23            mTime.value = "$mHour:$mMinute"
24        }, mHour, mMinute, false
25    )
26 }
```

codesnap.dev

## Step 4: Build the UI with Interactive Components

Let's combine all the components into a complete preferences screen:

We decided to position all the elements vertically, so let's use the **Column** UI element.



Remember, this column of elements needs to be added in `setContent` within the `Theme` container that `AndroidStudio` already created for you when you created this project. If you accidentally deleted this at the start, don't worry, just use the following code snippet to recreate your container:

▶ Expand

In pseudo-code terms, we need to do the following inside the column:

```
Column(){  
    Button <For opening DatePicker>  
  
    Text  
  
    Button <For opening TimePicker>  
  
    TextField  
  
    Checkbox  
  
    Text <Progress Bar>  
  
    ProgressBar  
}
```

Let's now implement this step by step..

We'll add spacers as we go along and add some vertical padding between each element in the column..

- Add a button for each picker

```
1     Button(onClick = { mDatePickerDialog.show() }) {  
2         Text(text = "Open Date Picker")  
3     }  
4     Text(text = "Selected Date: ${mDate.value}")  
5     Spacer(Modifier.height(40.dp))  
6     Button(onClick = { mTimePickerDialog.show() }) {  
7         Text(text = "Open Time Picker")  
8     }  
9     Text(text = "Selected Time: ${mTime.value}")  
10
```

codesnap.dev

- Add a text field to read a text from the user. We will save this text later in the shared preferences.

```
TextField(  
    value = mTextFieldValue.value,  
    onValueChange = { mTextFieldValue.value = it },  
    label = { Text("Enter Text") }  
)  
Spacer(Modifier.height(40.dp))
```

codesnap.dev

- Add a checkbox and link it to the mCheckBoxState variable.

```
Checkbox(  
    checked = mCheckBoxState.value,  
    onCheckedChange = { mCheckBoxState.value = it }  
)  
Spacer(Modifier.height(20.dp))
```

codesnap.dev

- Add a slider and set its range to 0..100

```
Text(text = "Slider Value: ${sliderValue.toInt()}")
Slider(
    value = sliderValue,
    onValueChange = { sliderValue = it },
    valueRange = 0f..100f
)
Spacer(Modifier.height(40.dp))
```

codesnap.dev

- Add a progress bar and link its value to the slider's value.

```
Text("ProgressBar")
Spacer(Modifier.height(20.dp))

LinearProgressIndicator(
    progress = { sliderValue / 100f },
    modifier = Modifier
        .padding(10.dp)
)
```

codesnap.dev

## Step 5: Implement SharedPreferences Storage Functions

Now, let's implement functions to save and load preferences:

We must get an instance from the shared preferences object to save and restore data to/from shared

preferences storage.

```
val sharedPref = mContext.getSharedPreferences("week3_sp", Context.MODE_PRIVATE).edit()
```

codesnap.dev

In the code above:

- The first parameter is the file name. It allows us to create multiple storage if needed.
- The MODE\_PRIVATE mode means only the calling application will access the created file.
- The edit() opens the file for updates. You don't need it during the data retrieval.

Now, let's push some data.

```
1     Button(onClick = {
2         val sharedPref =
3             mContext.getSharedPreferences("week3_sp", Context.MODE_PRIVATE)
4                 .edit()
5
6         sharedPref.putString("date", mDate.value)
7         sharedPref.putString("time", mTime.value)
8         sharedPref.putString("text", mTextFieldValue.value)
9         sharedPref.putBoolean("checkbox", mCheckBoxState.value)
10        sharedPref.putInt("slider", sliderValue.toInt())
11        sharedPref.apply()
12    }) {
13        Text(text = "Save Values")
14    }
```

codesnap.dev

As you can see in the code above, in lines 6-10, we added all the values using the key-value pairs format. We must save all the keys in a safe place so that we can use them again during the data restoration. The statement in line 11 applies the changes to the physical storage.

To restore, add another button that gets an instance of the shared preferences (without the .edit()), get all the values and update the UI elements.

```
1     Button(onClick = {
2         val sharedPref =
3             mContext.getSharedPreferences("week3_sp", Context.MODE_PRIVATE)
4
5         val loadedDate = sharedPref.getString("date", "01/03/2025")
6         val loadedTime = sharedPref.getString("time", "12:00")
7         val loadedText = sharedPref.getString("text", "")
8         val loadedCheckbox = sharedPref.getBoolean("checkbox", false)
9         val loadedSlider = sharedPref.getInt("slider", 0).toFloat()
10
11         mDate.value = loadedDate.toString()
12         mTime.value = loadedTime.toString()
13         mTextFieldValue.value = loadedText.toString()
14         mCheckBoxState.value = loadedCheckbox
15         sliderValue = loadedSlider
16     }) {
17         Text(text = "Load Saved Values")
18     }

```

codesnap.dev

## How SharedPreferences Works

When you save data with SharedPreferences:

1. Android creates or opens an XML file in your app's private storage area
2. Data is written to this file in key-value format
3. The file persists across app restarts and device reboots
4. When you read from SharedPreferences, Android retrieves values from this file

For example, your preferences might be stored like this:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="user_notes">Remember to drink water</string>
    <boolean name="notifications_enabled" value="true" />
    <float name="preference_level" value="75.0" />
    <long name="selected_date" value="1677456000000" />
    <long name="selected_time" value="1677412800000" />
</map>
```

## Best Practices for SharedPreferences

1. **Use Consistent Keys:** Store key names as constants to avoid typos
2. **Apply vs Commit:** Use `apply()` for asynchronous saves (better performance) and `commit()`

only when you need to know if the save succeeded

3. **Handle Default Values:** Always provide default values when retrieving preferences
4. **Security Considerations:** Don't store sensitive information like passwords without encryption
5. **Preference Cleanup:** Consider clearing outdated preferences when your app updates

## Check Your Understanding

1. What is the main purpose of SharedPreferences in Android?
2. How would you retrieve a string value from SharedPreferences with a default value?
3. What's the difference between `apply()` and `commit()` when saving preferences?
4. How can you implement a time picker that allows selection of precise minutes?
5. How do sliders and progress bars work together to provide visual feedback?

## Resources:

[SharedPreferences on Android Developers](#)

## 4. Dealing with Assets: Reading CSV Files

### Learning Objectives

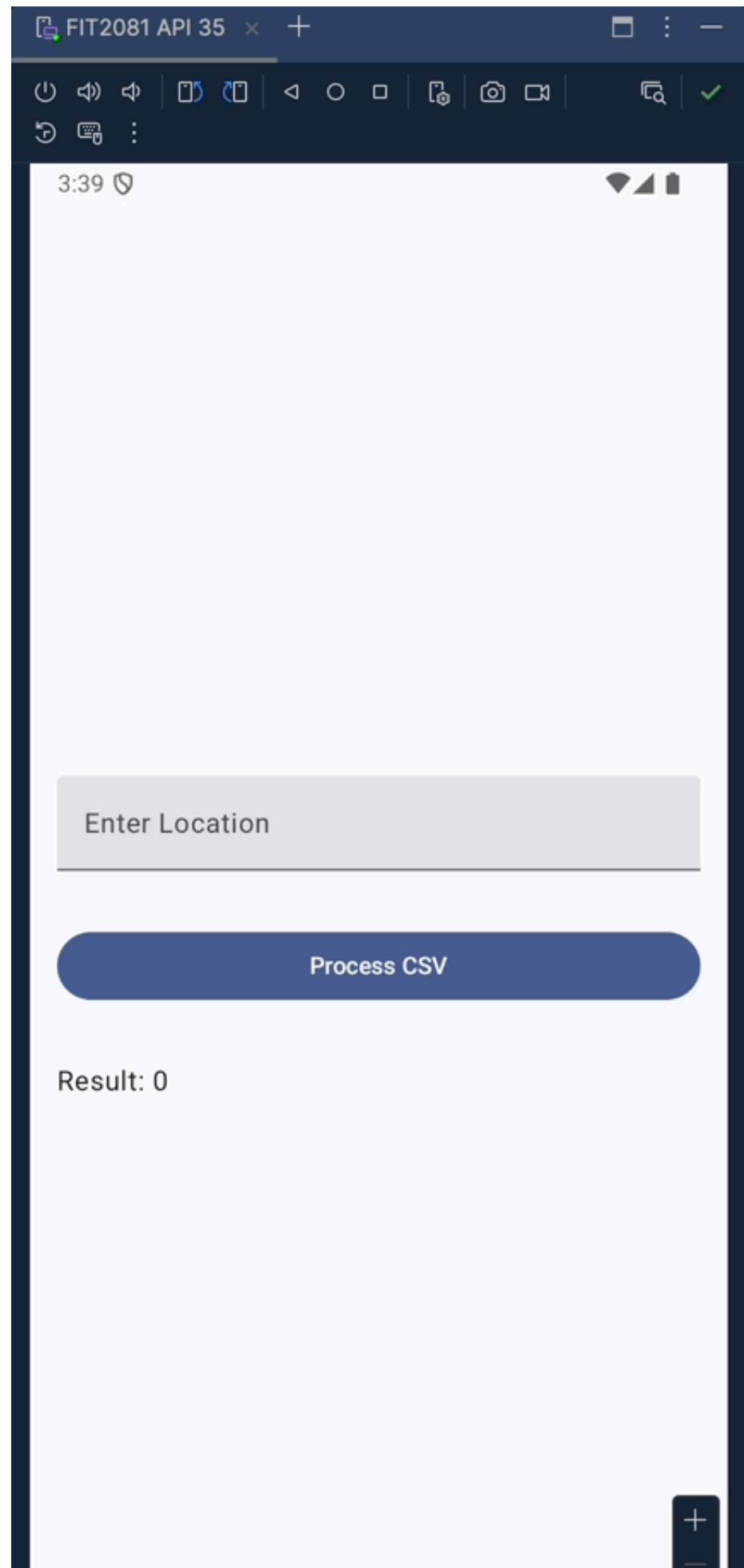
By the end of this section, you will be able to:

- Create and configure an assets folder in Android Studio
- Read raw files from Android assets
- Parse CSV data in Kotlin
- Filter and process structured data from files
- Display results from CSV data in your app's UI

### Expected Output

We'll build an app that reads a CSV file and based on user-input calculates the number of relevant entries in the CSV file.

Here is the expected output.



# What is the Assets folder, and what are its roles in Android Applications

The assets folder in Android applications is a directory within your Project where you can store raw files that your application might need at runtime. These files are bundled directly into your APK (Android Package) without any processing by the build system. This differs from resources in the res/ directory, which are processed and assigned resource IDs.

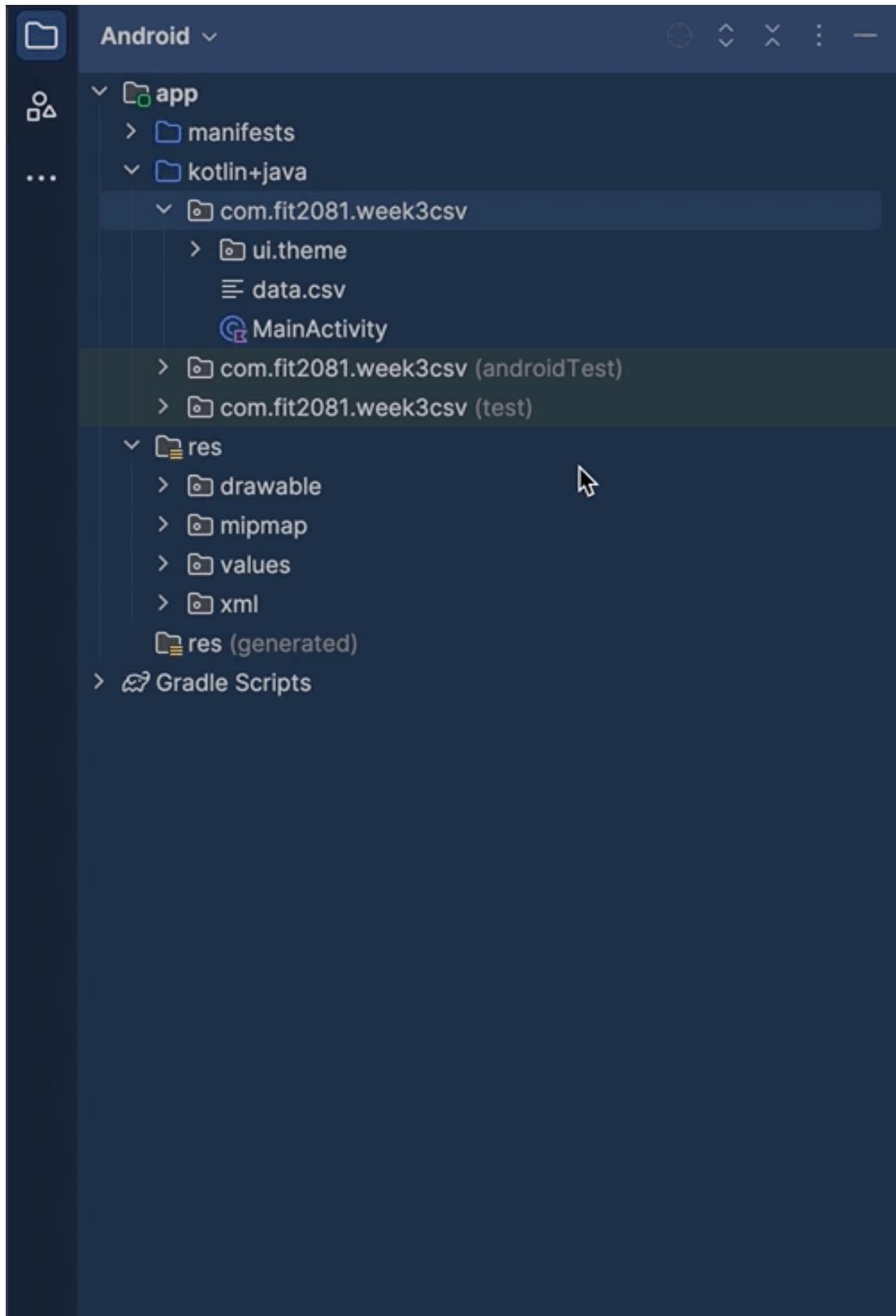
## Purpose of the Assets Folder

- Storing Raw Files: The primary purpose is to store arbitrary files in their original format. This can include text files, JSON files, HTML files, audio files, video files, and more.
- Accessing Files Directly: Files in the assets folder are not assigned resource IDs. Instead, you can access them directly using their file path using the AssetManager class.
- Data Bundling: The assets folder is useful when you want to bundle data or content with your app that doesn't neatly fit into the standard resource types provided by the res/ directory.
- Read-Only at Runtime: These files become read-only once the app is built. You cannot write or modify them directly at runtime.

## How to create the assets folder

Follow these steps to create an assets folder in your Android project:

1. In Android Studio, change the Project Explorer view from **Android** to **Project**
2. Navigate to the `main` folder in your project
3. Right-click on the `main` folder and select **New → Directory**
4. Name the directory `assets` and click **OK**



Drag and drop the required files (s) into the assets folder. I will add a CSV file; find it below.

 [data.csv](#)

And here is its content.

Username	Identifier	One-time password	Recovery code	First name	Last name	Department	Location
booker12	9012	12se74	rb9012	Rachel	Booker	Sales	Manchester
grey07	2070	04ap67	lg2070	Laura	Grey	Depot	London
johnson81	4081	30no86	cj4081	Craig	Johnson	Depot	London
jenkins46	9346	14ju73	mj9346	Mary	Jenkins	Engineering	Manchester
smith79	5079	09ja61	js5079	Jamie	Smith	Engineering	Manchester

## Project



week3csv2 [week3csv] ~/projects/week3csv2

.gradle

.idea

.kotlin

app

build

src

androidTest

main

assets

data.csv

java

com.fit2081.week3csv

ui.theme

MainActivity

res

AndroidManifest.xml

test [unitTest]

.gitignore

build.gradle.kts

proguard-rules.pro

gradle

.gitignore

build.gradle.kts

gradle.properties

Switch back to Android View in the Project Explorer.

Now, let's develop an Android application that counts the number of users who live in the provided location.

## CSV Counter Function

In this function:

- We receive the context as a parameter to access the AssetsManager
- The second and third parameters are the asset filename and location.
- In line 7, we open the file and then access its content for reading using the BufferedReader class (line 9).
- Lines 10-19 iterates through all the lines after skipping the first row( the header)

```
1 // Function to count the number of rows in a CSV file that match a given location.
2 fun countRowsByLocation(context: Context, fileName: String, location: String): Int {
3     var count = 0 // Initialize the count to 0.
4     var assets = context.assets // Get the asset manager.
5     // Try to open the CSV file and read it line by line.
6     try {
7         val inputStream = assets.open(fileName) // Open the file from assets.
8         // Create a buffered reader for efficient reading.
9         val reader = BufferedReader(InputStreamReader(inputStream))
10        reader.useLines { lines →
11            lines.drop(1).forEach { line → // Skip the header row.
12                val values = line.split(",") // Split each line into values.
13                // Check if the row has enough columns and
14                // if the 7th column matches the location.
15                if (values.size > 6 && values[7].trim() == location.trim()) {
16                    count++ // Increment the count if the location matches.
17                }
18            }
19        }
20    } catch (e: Exception) {
21        // Handle any exceptions that might occur during file reading.
22    }
23    // Return the total count of rows matching the location.
24    return count
25 }
```

codesnap.dev

## CSV Screen

We start the function by declaring the required variables: location, count, and result.

```
// State to hold the location input.  
var location by remember { mutableStateOf("") }  
// State to hold the count of rows matching the location.  
var count by remember { mutableStateOf(0) }  
// State to hold the text to display the result.  
var resultText by remember { mutableStateOf("Result: 0") }  
// Column to arrange UI elements vertically.
```

codesnap.dev

Then, add a column with the following children:

A text field to enter the location. As you can see in line 5, we updated the value of the variable location when a change happens to the text field.

```
1 // Text field for entering the location.  
2 TextField(  
3     value = location,  
4     // Update location state when the text changes.  
5     onValueChange = { location = it },  
6     label = { Text("Enter Location") },  
7     modifier = Modifier  
8         .fillMaxWidth()  
9         .padding(bottom = 16.dp)  
10    )  
11    // Spacer for adding vertical space.  
12    Spacer(modifier = Modifier.height(16.dp))
```

codesnap.dev

After that, add a button that invokes the CSV processing function. Line 7 invokes the function we developed earlier to count the number of rows with a location equal to the one we got from the user and passed as the third parameter.

```
1      // Button to process the CSV.
2      Button(
3          onClick = {
4              // Call the function to count rows matching the location.
5              // the second parameter is the file name that should be saved in
6              // app/src/main/assets/data.csv
7              count = CountRowsByLocation(context, "data.csv", location)
8              // Update the result text to show the count.
9              resultText = "Result: $count"
10         },
11         modifier = Modifier
12             .fillMaxWidth()
13             .padding(bottom = 16.dp)
14     ) {
15         Text("Process CSV")
16     }
17     // Spacer for adding vertical space.
18     Spacer(modifier = Modifier.height(16.dp))
19     // Text to display the result.
20     Text(text = resultText)
21 }
```

Finally, the onCreate function must call the CSVProcessorScreen function to process the UI of the activity.

```
1 class MainActivity : ComponentActivity() {
2     // Called when the activity is starting.
3     override fun onCreate(savedInstanceState: Bundle?) {
4         super.onCreate(savedInstanceState)
5         // Enable edge-to-edge display.
6         enableEdgeToEdge()
7         // Set the content view using Jetpack Compose.
8         setContent {
9             Week3csvTheme { // Apply the application's theme.
10                 Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
11                     //the this@MainActivity is the context of the activity
12                     //we pass it down to access the assets later
13                     CSVProcessorScreen(
14                         this@MainActivity,
15                         modifier = Modifier.padding(innerPadding)
16                     )
17                 }
18             }
19         }
20     }
21 }
```

## 5. Challenge Exercises

1. Modify the input validation app to:
  1. Accept only Monash email addresses.
  2. Require passwords with a minimum length of 8 characters, consisting of only letters and digits.
  3. (Advanced) Only enable the Button if the email/password fields are not-empty and do not have errors. \*
2. Add a button to the shared preferences app to clear all the storage items.
3. Modify the CSV app to:
  1. Accept a user ID as input.
  2. Display the corresponding first name, last name, and department.
  3. Return an error message if the ID is not found.
4. Modify the modals app to:
  1. Accept the final mark as an integer from 0 to 100.
  2. Accept the unit code as a string.
  3. The caller function (the function that invokes the AlertDialog) retrieves the three values (name, mark and unit code) and displays them on the activity using three different Text elements.

\* Spoiler: Answer for 1.3

► Expand

# Assessment Skills!

## Assessment Skills: Connecting This Lab to Assignment 1

### How Today's Lab Prepares You for Assignment 1

This lab has been specifically designed to develop the skills you'll need for the successful implementation of the NutriTrack app in Assignment 1. Let's examine the direct connections between today's exercises and your upcoming assignment work:

#### Login Screen Implementation (Lab Section 1)

##### **NutriTrack Requirement:**

- A login screen that validates user IDs and phone numbers against CSV data
- Display of error messages for invalid credentials

##### **Skills You'll Practice Today:**

- Creating text input fields with real-time validation
- Implementing conditional error messages
- Building a complete login workflow

**Application in Assignment:** You'll need to adapt today's login validation approach to verify user credentials against your CSV file in NutriTrack, displaying appropriate error messages for invalid inputs.

#### Modal Dialogs (Lab Section 2)

##### **NutriTrack Requirement:**

- Persona Information Modal that displays details when users click on a persona button

##### **Skills You'll Practice Today:**

- Creating overlay dialogs that focus user attention
- Passing data between the main screen and modals
- Implementing confirm and dismiss functionality

**Application in Assignment:** You'll implement similar modals for the Persona Information in NutriTrack's Food Intake Questionnaire, adapting the techniques to show persona details and characteristics.

## SharedPreferences & UI Components (Lab Section 3)

### NutriTrack Requirement:

- Storage of food preferences and questionnaire data
- Implementation of time pickers for meal timing information

### Skills You'll Practice Today:

- Saving and retrieving data with SharedPreferences
- Implementing date and time pickers
- Creating interactive UI components like sliders and checkboxes

**Application in Assignment:** You'll use SharedPreferences to store user questionnaire responses and implement time pickers for the three meal timing questions in the Food Intake Questionnaire.

## CSV Data Processing (Lab Section 4)

### NutriTrack Requirement:

- Reading user data and food quality scores from CSV files
- Displaying this data in the Home and Insights screens

### Skills You'll Practice Today:

- Setting up the assets folder
- Reading and parsing CSV files
- Extracting specific data based on search criteria

**Application in Assignment:** You'll adapt these techniques to load user IDs, validate phone numbers, and retrieve food quality scores from your assignment's CSV data file.

## Assessment Checklist

Use this checklist to ensure you're mastering the skills needed for Assignment 1:

- I can implement input validation for text fields
- I can display appropriate error messages for invalid inputs
- I can create and manage modal dialogs
- I can pass data between screens and components
- I can save and retrieve data using SharedPreferences
- I can implement time pickers for user input
- I can read and parse CSV files
- I can extract specific data from CSV files based on search criteria
- I can display data from CSV files in my UI

By thoroughly understanding and practicing these concepts in today's lab, you'll develop the foundation needed to tackle the NutriTrack implementation with confidence.

# FIT2081 Week 4

## Kotlin OOP & UI

### Components

**FIT2081**  
**Mobile application**  
**development (MAD)**



## Navigation | LazyColumn

**Week 4**

Delvin Varghese

## Checklist

- I'm more than 50% done with Assignment 1
- I can load a CSV file
- I know how to save values using SharedPreferences

## **Announcement:**

**In-semester quiz (20%) will now be conducted as  
an online (Moodle-based) in-semester assessment  
on:**

**W12, Monday 13:00**

# Learning objectives for today

## 1. Navigation

- How to navigate in a multi-screen app
- Modern navigation practices

## 2. LazyColumn

- Elegantly displaying data to users



"Today we're going to learn how to implement navigation in a modern Android app using Jetpack Compose.

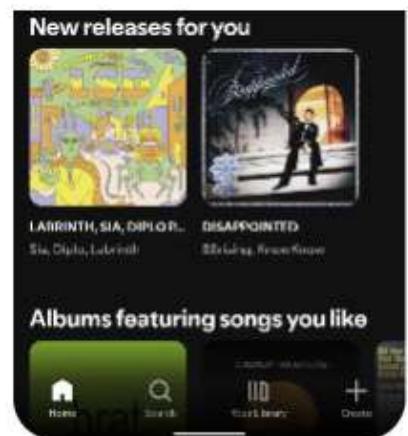
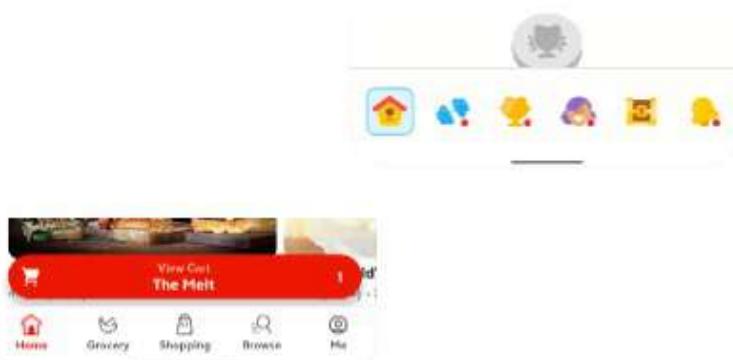
Navigation is one of the most fundamental aspects of mobile app development—it's how users move between different screens and access various features.

We'll build an F1 Stats app with multiple screens connected through both traditional Intent-based navigation and modern Compose Navigation with a bottom navigation bar."

# Navigation

## 1. Navigation

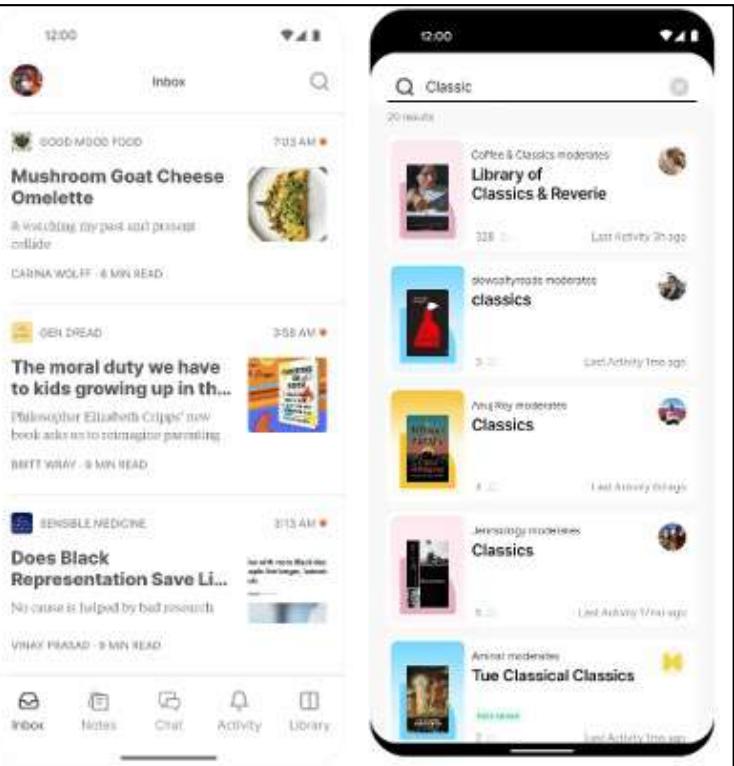
- How to navigate in a multi-screen app
- Modern navigation practices



# LazyColumn

## 2. LazyColumn

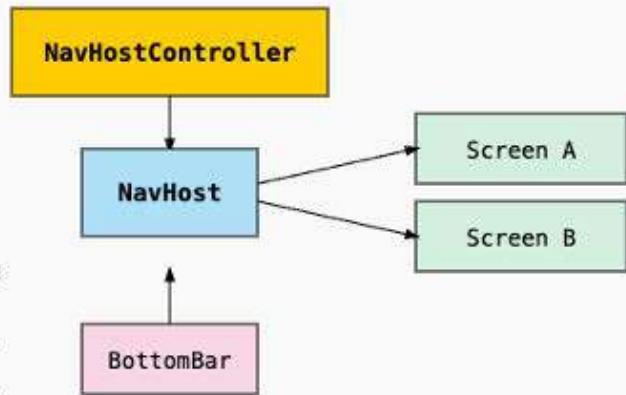
- Elegantly displaying data to users



# 1. Navigation

## Behind the Scenes

- NavHostController controls navigation and is shared between NavHost and BottomBar.
- NavHost manages which composable to show based on route.
- Each destination (Screen A, Screen B) is a composable registered with the NavHost.
- BottomBar uses the NavHostController to trigger navigation when a tab is clicked.



- **Remember**
- Introduce rememberNavController(): "This creates and remembers a NavHostController that will manage our navigation. The 'remember' part ensures it survives recomposition events."
- Explain Scaffold with bottomBar: "We're adding a parameter to our Scaffold to include a bottom navigation bar, which is a common UI pattern in mobile apps."
- Highlight connection between components: "Notice how we pass the navController to

both the BottomBar and NavHost. This is crucial—they need to share the same controller to coordinate navigation."

- **Step Aside:** "In modern apps, bottom navigation is one of the most common patterns for primary navigation. Apps like Instagram, Twitter, and Spotify all use this pattern. It's effective because it keeps navigation always accessible without taking up too much screen space."

```
@Composable
fun MyNavHost(innerPadding: PaddingValues, navController: NavHostController) {
    NavHost(
        navController = navController,
        startDestination = "home"
    ){
        composable("home") {
            HomeScreen(innerPadding)
        }
        composable("rankings") {
            F1CardDisplay(modifier = Modifier.padding(innerPadding))
        }
        composable("settings") {
            SettingsScreen(innerPadding)
        }
    }
}
```

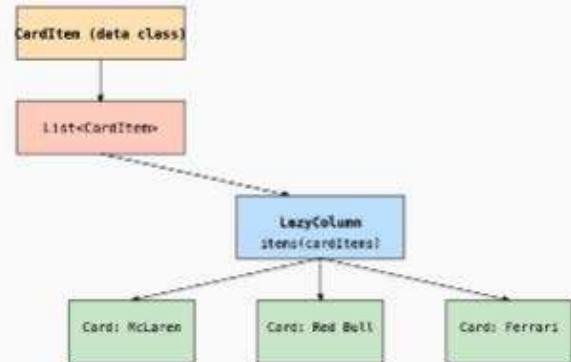
- Explain NavHost: "This is the container where our navigation destinations will be swapped in and out as the user navigates."
- Point out startDestination: "This defines which screen is shown first when this NavHost is created."
- Describe composable(): "Each composable function defines a destination in our navigation graph. The string parameter is the route name that identifies this destination."
- **Step Aside:** "Think of routes like

addresses. When we navigate, we're telling the NavController 'I want to go to this address' and it knows which screen to display based on the route name."

## **2. LazyColumns**

## Behind the Scenes

- 1. Data Class: CardItem:** You define a simple Kotlin data class with fields title, text, and image. Each instance represents one card's data (e.g., for Ferrari, Red Bull, McLaren).
- 2. List of Data:** You create a List<CardItem> inside your F1CardDisplay composable. This list acts as the **data source** for your UI.
- 3. LazyColumn:** You use LazyColumn to efficiently display scrollable content. It only composes visible items, making it memory-efficient.
- 4. items( ) Block:** You use the items() function from LazyColumn to iterate through your cardItems list. For each item, a **Card UI component** is created.
- 5. Card:** Each item in the list is rendered as a Card, which wraps a Column containing the title (Text), image (Image), and description (Text) from the CardItem data.

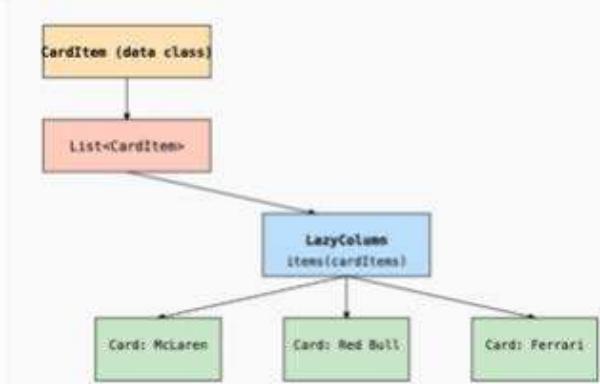


## Benefits

This setup represents a clean **data-driven**

**UI** pattern in Compose:

- **Separate data from UI logic** using CardItem.
- LazyColumn ensures **performance and efficiency**.
- Card components give a polished, structured layout for each entry.



## Data Class

```
data class CardItem(val title: String, val text: String, val image: Int)
```

Kotlin Data Classes are special classes primarily intended to hold data.

**Auto-generated methods:** Data classes automatically generate `equals()`, `hashCode()`, `toString()`, `copy()`, for the properties declared in the primary constructor.



**LIVE CODING\_**



**IT'S ALL OVER**

**THE END\_**

# Lab: Kotlin OOP, LazyColumn, and Navigation

---

## Overview

### Overview

This week's lab builds essential skills for displaying and managing collections of data in your Android apps. You'll learn how to implement a LazyColumn with Card components and organize your code with proper architecture patterns.

### Learning Objectives

By the end of this lab, you will be able to:

1. Create and use Kotlin data classes to model domain entities
2. Implement efficient list displays using LazyColumn and Card components
3. Manage list state using proper architecture patterns
4. Apply Material Design principles to create visually appealing list items
5. Implement common list interactions like adding, removing, and filtering items

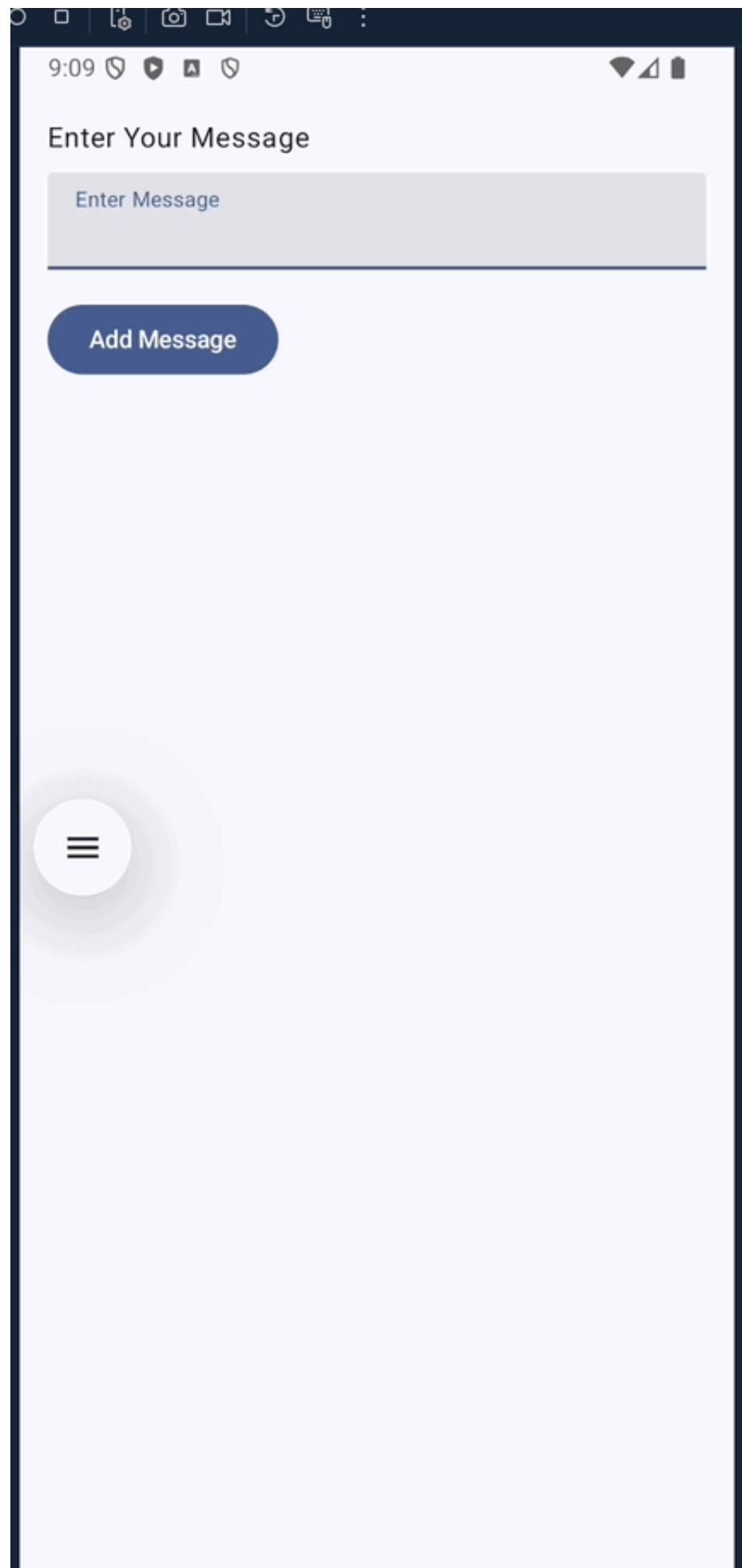
## Resources

You can download the image resources we use in this Lab here:

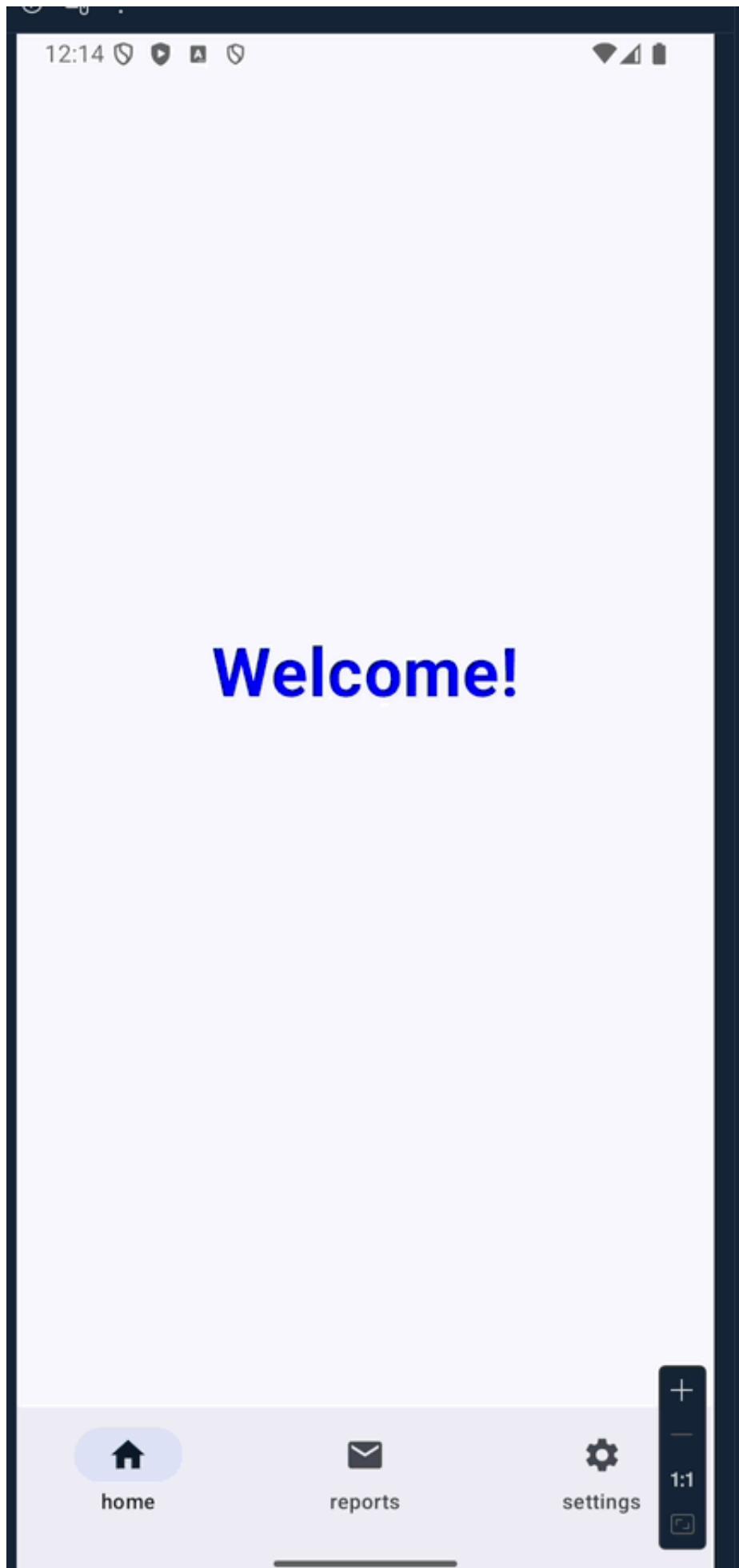


## Expected Outputs from W4 Labs:

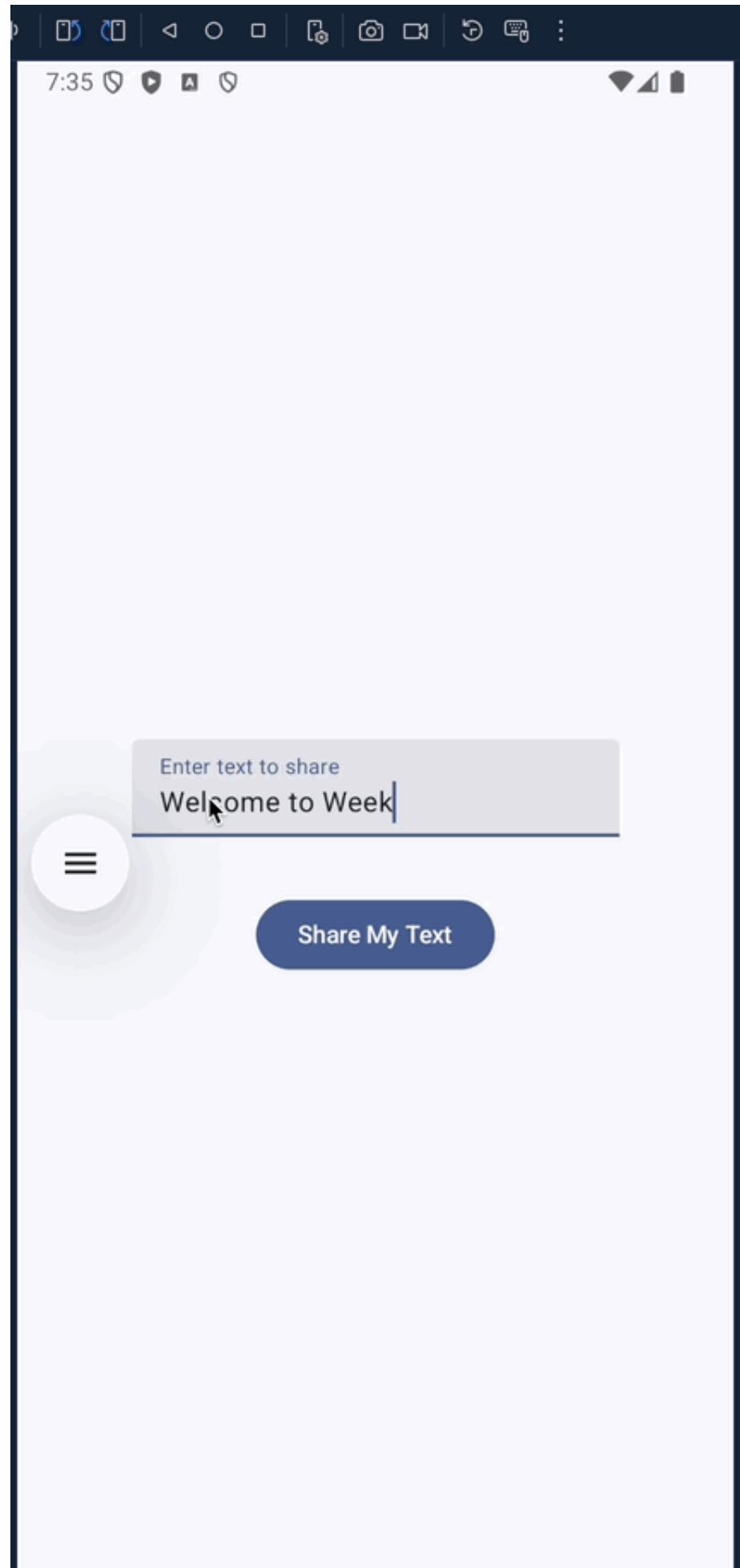
Output 1: LazyColumn Implementation



Output 2: Efficient Navigation in Jetpack Compose



Output 3: Sharing Data with other applications in Jetpack Compose



# A: Recap - OOP in Kotlin

**Objective:** Understand and practice Kotlin data classes and companion objects.

**Kotlin Data Classes** are special classes primarily intended to hold data. If you've used Java or Python before, you might recall how you sometimes create classes whose main purpose is to store a few values (fields) and possibly provide some basic functionality like equality checks (i.e., `equals()` / `__eq__()`) and string representation (`toString()` / `__str__()`).

Kotlin's Data Classes are designed to reduce the amount of "boilerplate" code you need to write for these basic tasks.

## 1. Recap: How Classes work..

Imagine you need a simple class to hold information about a **Person**, with fields like `name` and `age`.

In Java, you might write:



Hint: you can click the Run button below to run this code!

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    // Getter for name  
    public String getName() {  
        return name;  
    }  
  
    // Setter for name  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    // Getter for age  
    public int getAge() {  
        return age;  
    }  
  
    // Setter for age  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```

```
}

@Override
public String toString() {
    return "Details of your Person: {" +
        "name='" + name + '\'' +
        ", age=" + age +
        '}';
}

public static void main(String[] args) {
    Person myPerson = new Person("Gandalf", 130);
    System.out.println(myPerson);
}
}
```

In Python, you might write the same functionality like this:

 Hint: you can click the Run button below to run this code!

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"Details of your Person: {{name='{self.name}', age={self.age}}}""

if __name__ == "__main__":
    my_person = Person("Gandalf", 130)
    print(my_person)

# No explicit getters or setters: In Python, we typically access and set attributes (person.name, person.age)
# __str__ is used to provide a human-readable string of the object (like toString() in Java). For data classes, we often use a simple if __name__ == "__main__": block as an equivalent to Java's main method for running the class
```

**Take a look again at the code above at the coding language background you've come from (whether Java or Python).** The goal of Data classes is to emulate this behaviour but in Kotlin programming language.

How would we write this in Kotlin?

```
/**
 * In Kotlin, we generally don't write separate "getter" and "setter" methods
 * for simple properties—Kotlin automatically generates them behind the scenes
 * for 'var' properties.
 *
 * The 'main' function is written at the top level, which is typical in Kotlin.
 */
```

```

class Person(var name: String, var age: Int) {

    override fun toString(): String {
        return "Details of your Person: {name='$name', age=$age}"
    }
}

fun main() {
    val myPerson = Person("Gandalf", 130)
    println(myPerson)
}

```

**Takeaway: we've created a regular 'class' in Kotlin. Let's now look at Data classes.**

## 2. Data Classes (and why we use them rather than regular Classes)

In Kotlin, a **data class** is a specialized class primarily intended to store data. Here are the main differences compared to a regular class:

### 1. Auto-generated methods

- Data classes automatically generate `equals()`, `hashCode()`, `toString()`, `copy()`, and component functions (`component1()`, `component2()`, etc.) for the properties declared in the primary constructor.
- A regular class requires you to provide these methods yourself if you want custom or consistent behavior.

### 2. Primary constructor requirements

- Data classes must have at least one parameter in the primary constructor.
- Each of these primary constructor parameters needs to be marked as either `val` or `var`.

### 3. Intended usage

- Data classes are typically used to represent simple “models” or “entities” whose main purpose is to hold data.
- Regular classes are more general-purpose and may encapsulate complex logic, behavior, or state that doesn’t revolve solely around holding data.

### 4. Destructuring declarations

- Data classes enable destructuring declarations, which lets you do something like this without any extra code.

```
val (name, age) = Person("Gandalf", 130)
```

*Destructuring* is not possible with a regular class.

==

Now, let's compare what the Person class would look like as a data class..

```
// Regular class
class Person(var name: String, var age: Int)

// Data class
data class PersonData(var name: String, var age: Int)
```

## Differences

- With `PersonData`, you automatically get `equals()`, `hashCode()`, `toString()`, `copy()`, and the ability to destructure the object.
- With `Person`, you'd need to define any of those methods or behaviors yourself if you want them.

Finally, let's implement `PersonData` in Kotlin and see how the auto-generated `toString()` and `equals()` functionality works in practice:

 **Hint:** you can run this class by clicking 'Run'

```
// Data class automatically provides:
// - equals() / hashCode()
// - toString()
// - copy()

data class PersonData(var name: String, var age: Int)

fun main() {
    val person1 = PersonData("Gandalf", 130)
    val person2 = PersonData("Gandalf", 130)
    val person3 = PersonData("Boromir", 130)

    // toString() is used automatically when printing
    println("Person1: $person1")

    // equals() is automatically generated to compare field values
    println("Person1 == Person2? ${person1 == person2}") // true, since name and age match
    println("Person1 == Person3? ${person1 == person3}") // false, since name doesn't match
}
```

=====

### Reflection Prompt:

"How might Kotlin's data classes improve your app's maintainability and readability?"

=====

## Benefits for Android Development (e.g., LazyColumn/Cards in UI)

When developing Android apps (especially with **Jetpack Compose**), you often represent your screen state or the information displayed in UI elements as plain data objects. For instance:

- In a **LazyColumn**, you might have a list of items, each represented by some data class (e.g., `Article`, `UserProfile`, `MovieItem`).
- In **Cards**, you may display structured data (title, subtitle, image link) – a perfect job for a data class.

### a) Easy State Management

By using data classes for state, you can quickly create, copy, and modify states without worrying about writing manual copy constructors or equals methods. This is crucial in Jetpack Compose, where you often deal with **immutable states** and update them by creating copies.

```
data class Article(val title: String, val content: String, val author: String)
```

If you need a **slightly** modified version (e.g., edited content), you do:

```
val editedArticle = oldArticle.copy(content = "New content")
```

### b) Readability and Clarity

When you pass data objects to Composable functions (e.g., `ArticleCard(article: Article)`), it's super clear what is being passed around. You also get a well-formatted `toString()` in logs, making debugging easier.

### c) Performance and Correctness

Since Data Classes automatically implement `equals()` and `hashCode()`, data comparison (like checking if a list item changed) is consistent and less error-prone.

 This is helpful in certain optimizations or when working with **DiffUtil** in RecyclerView-like structures or Compose's recomposition logic.

# B: Implementing LazyColumn Pt. 1

## 1.1 The Problem that LazyColumn solves

Let's assume you are developing an application for managing an extensive list of items, such as a warehouse manager. Using traditional (older) approaches, updating a list dynamically (e.g., adding, removing, or modifying items) can be cumbersome and inefficient.

For example, RecyclerView requires notifying the adapter of changes, which can lead to unnecessary UI updates or inconsistencies. Also, Rendering all items in a list at once (e.g., using a `ScrollView`) can lead to excessive memory usage, especially for large datasets or complex item layouts.

```
// Old approach using ScrollView (inefficient for large lists)
ScrollView {
    Column {
        for (item in allItems) {
            ItemComponent(item) // All items are composed at once!
        }
    }
}

// Old RecyclerView approach required a lot of boilerplate
// - Adapter class
// - ViewHolder class
// - Multiple layout files
// - Manual diffing for updates
```

### Solution: Use LazyColumn.

#### LazyColumn solves these problems by:

- Only composing and laying out visible items
- Handling recycling automatically
- Providing a simple, declarative API
- Managing state and updates efficiently

## What is LazyColumn?

`LazyColumn` is a component in Jetpack Compose, a modern Android UI toolkit for building native user interfaces. It is designed to efficiently display a vertically scrolling list of items, particularly when the list is large or dynamic. Unlike traditional RecyclerView in Android, `LazyColumn` only composes and lays out the currently visible items on the screen, making it highly performant for large datasets.



Click [HERE](#) for more info about LazyColumn from Android official documentation.

**Reflection:** Why is LazyColumn more efficient than a regular Column inside a ScrollView for displaying large lists?

► Expand

So it is good to be lazy!

► Expand

## 1.2 What is a Card?

A card is a pre-defined composable that represents a Material Design card component. Cards are used to display content and actions about a single subject in a concise and visually appealing way. They are commonly used in UIs to group related information, such as in news feeds, product listings, or profile details.

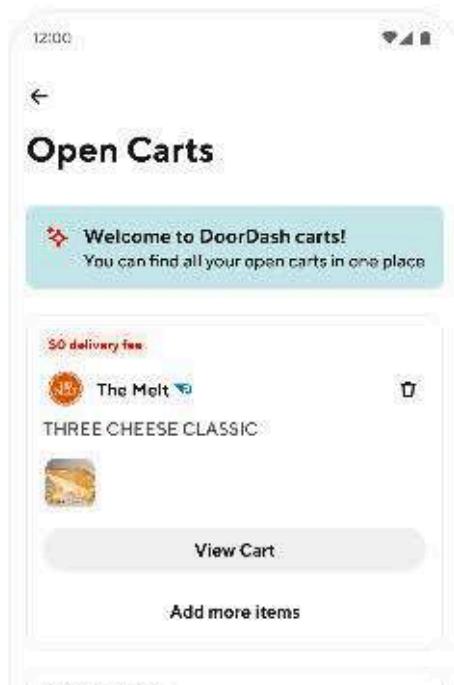
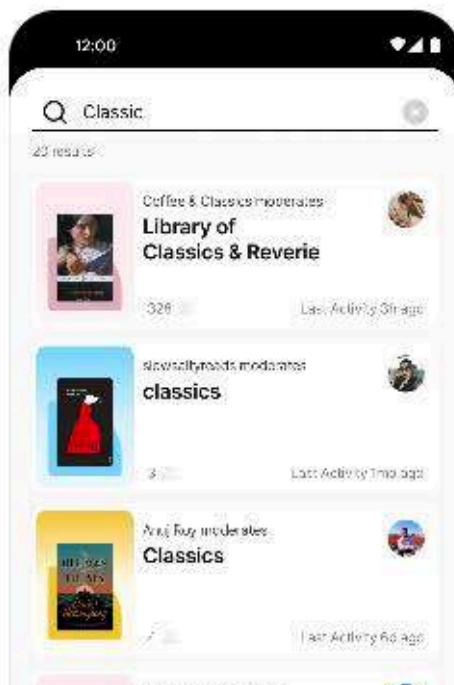
Key Features of Card:

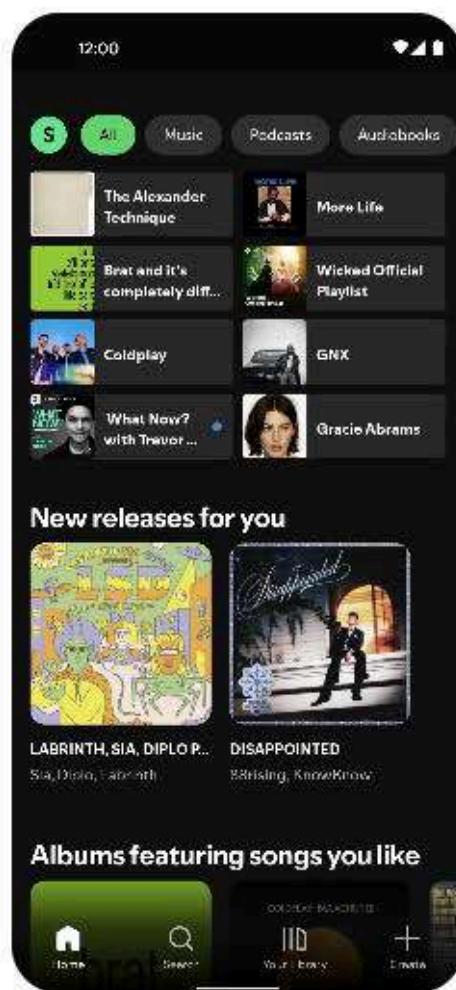
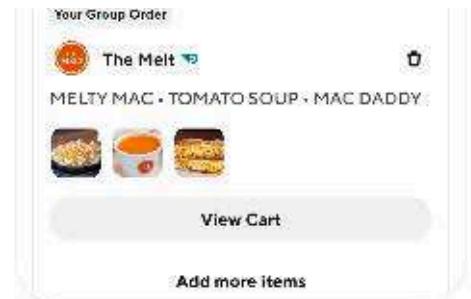
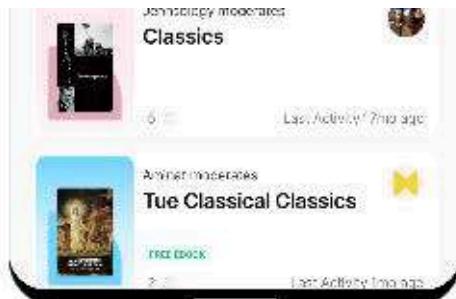
- Elevation: Cards have a default elevation, which gives them a shadow effect, making them appear raised above the background.
- Rounded Corners: Cards have rounded corners by default, following Material Design guidelines.
- Padding: Cards provide built-in padding to separate their content from the edges.
- Customizable: You can customize the shape, elevation, background colour, and other properties of a Card.



For more info about Cards, click [HERE](#)

Almost every single app uses Cards. Have a look at the following screenshots:



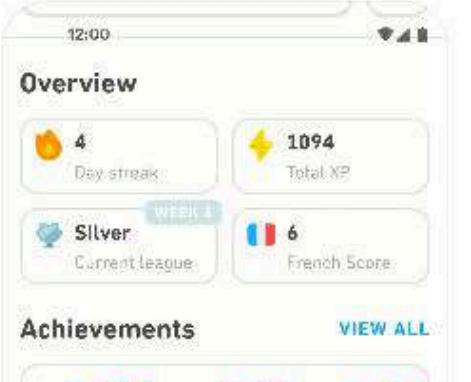
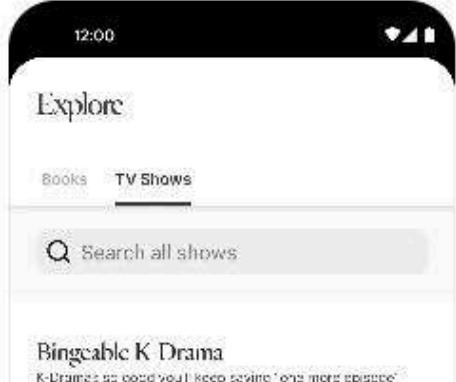


Spotify



Instagram

**i** As you can see, cards come in all sorts of styles and contents!





## Now streaming

The TV shows we're binging now.



88



Recent



Watchlist

**un garçon**  
a boy

**une fille**  
a girl

**un homme**  
a man

**une femme**  
a woman

**un chien**  
a dog



## Friend suggestions



Jazzy

You may know each other

FOLLOW



Sam

You may know each other

FOLLOW



John

One can



12:00

Search music

Saved

For you      Browse

Original audio

- Amélie x Soulmate - andreavanzo, composer • 0:41
- Gifts to your future self - adam dodson • 0:33
- Too Sweet (Short Cover) - bernicarpensproject • 0:38

Birthday

- Unhappy Birthday (2011 Remaster) - The Smiths • 2:45
- Ratchet Happy Birthday - Drake • 3:27
- Cake By The Ocean - DNCE • 3:35

Date Night

baggy jeans

Save

Explore      Shop

\$17.11 ADY70K15 Dudsella

\$47.99 ADY70K15 Dudsella

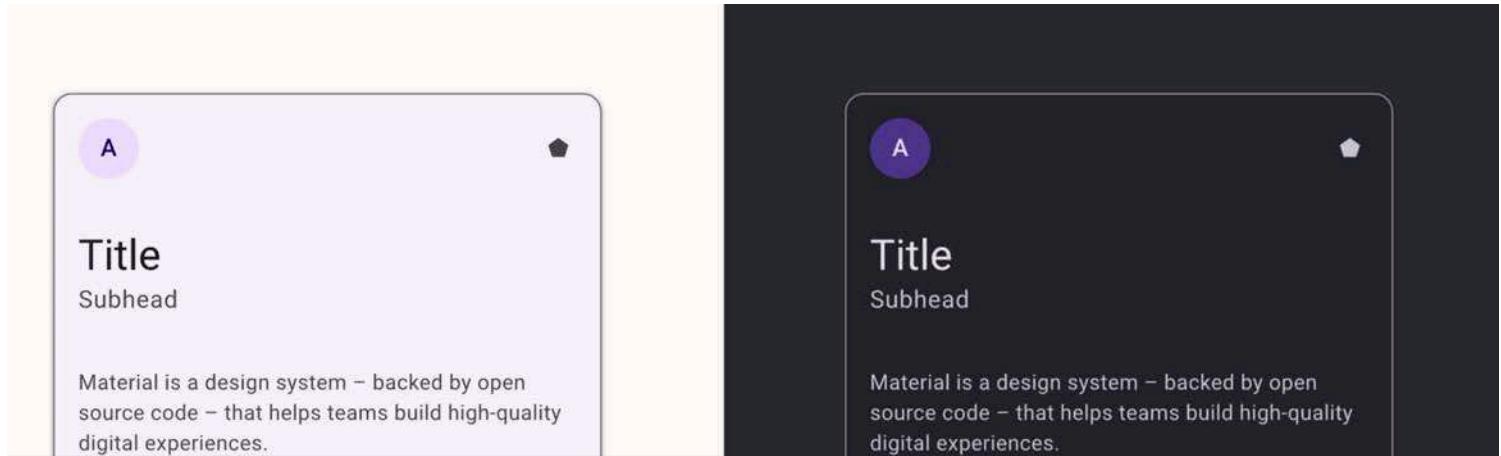
\$21.38 ADY70K15 Dudsella

\$17.11 ADY70K15 Dudsella

Search

Grid view





Let's take a look at the code needed for a very basic card..

```
Card(  
    modifier = Modifier  
        .padding(8.dp)  
        .fillMaxWidth(),  
    elevation = CardDefaults.cardElevation(defaultElevation = 4.dp),  
    shape = RoundedCornerShape(12.dp),  
    colors = CardDefaults.cardColors(  
        containerColor = MaterialTheme.colorScheme.surface,  
    ),  
    // Card content goes here  
)  
  
// Dont forget to import androidx.compose.material3.Card
```

## What are the different parameters?

- **modifier**
  - Customizes layout properties (e.g., padding, width, alignment).
  - Typical use: `.padding()`, `.fillMaxWidth()`.
- **elevation**
  - Controls shadow and depth effect.
  - Set using `CardDefaults.cardElevation()` (e.g., `defaultElevation = 4.dp`).
- **shape**
  - Defines the corner shape of the card.
  - Typically uses `RoundedCornerShape()` (e.g., `RoundedCornerShape(12.dp)`).
- **colors**
  - Manages card background and content colors.
  - Set via `CardDefaults.cardColors()` (e.g., `containerColor = MaterialTheme.colorScheme.surface`).
- **content**
  - A composable lambda containing UI elements displayed inside the card.
  - Common examples: `Text`, `Button`, `Image`, `Row`, `Column`.



Just like other UI elements, **you will have to import** CardDefaults, RoundedCornerShape, MaterialTheme, Modifer & unit.dp. Use Android Studio IDE to auto-import missing imports!

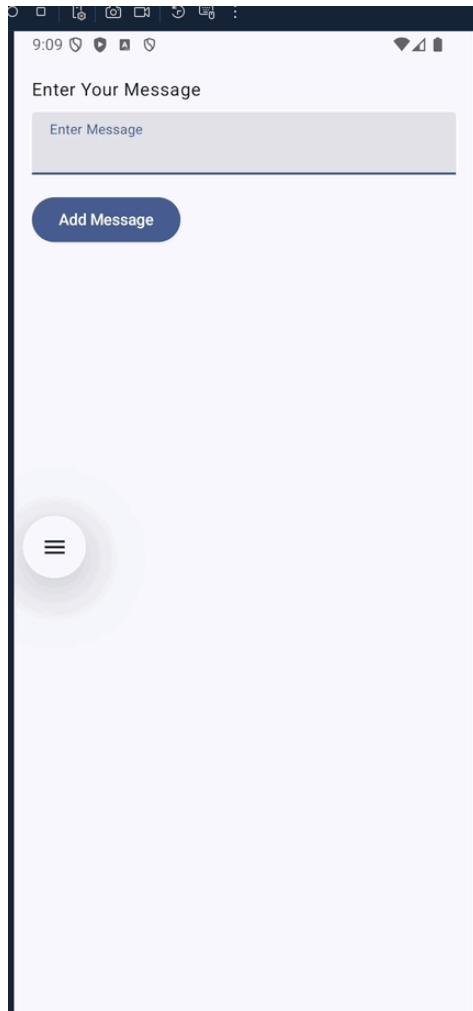
**Check your understanding:** When would you use a Card component instead of a simple Surface or Box?

▶ Expand

## 1.3 Time for Coding

Now, let's develop an application that uses LazyColumn to list a set of messages.

Here is the expected output:



## B. Implementing LazyColumn Pt. 2

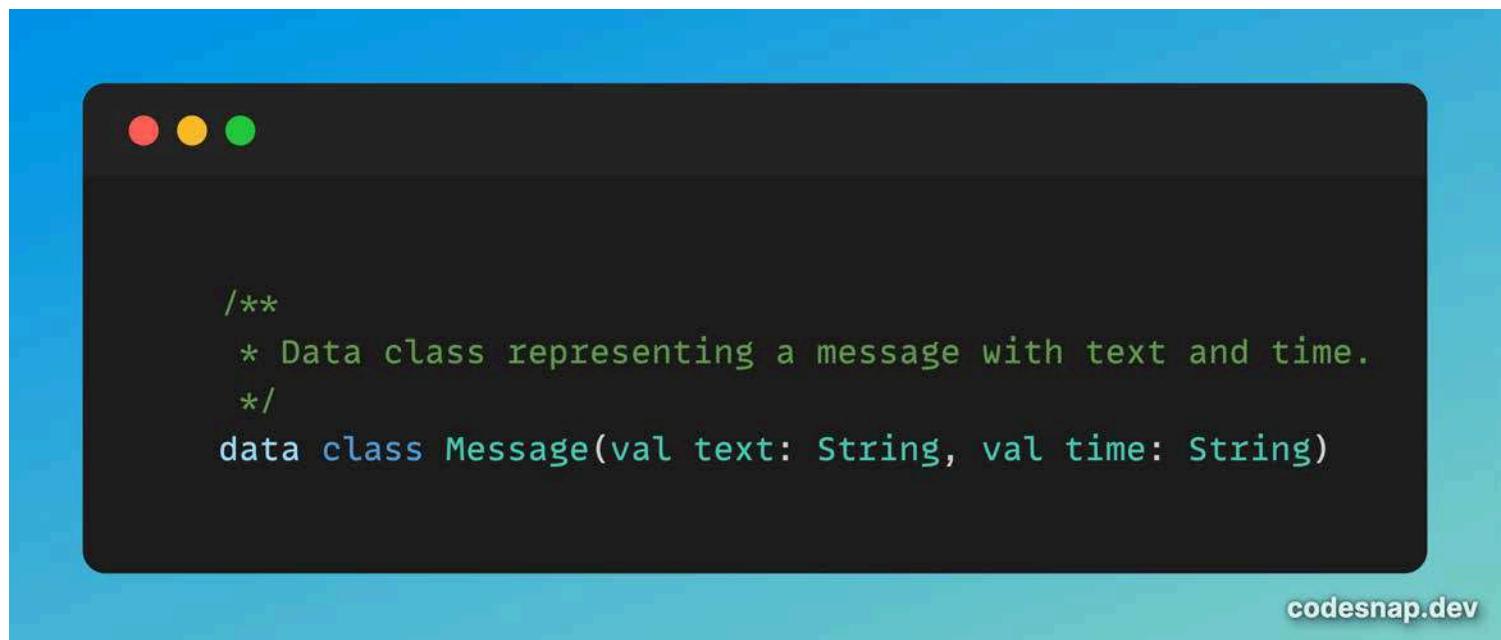
### 1.3a Creating a Message Data Class

In our app, we'll create a simple messaging app.

Create a new Kotlin app (Empty Activity) as you've done in previous weeks.

First, let's define a data class inside MainActivity.kt to represent a message:

*Make sure to read and understand Part A: Recap - OOP in Kotlin if you are not sure about data classes.*



Where should you place this class? Try yourself first before looking at the answer:

► Expand

### 1.3b Understand the UI Requirements

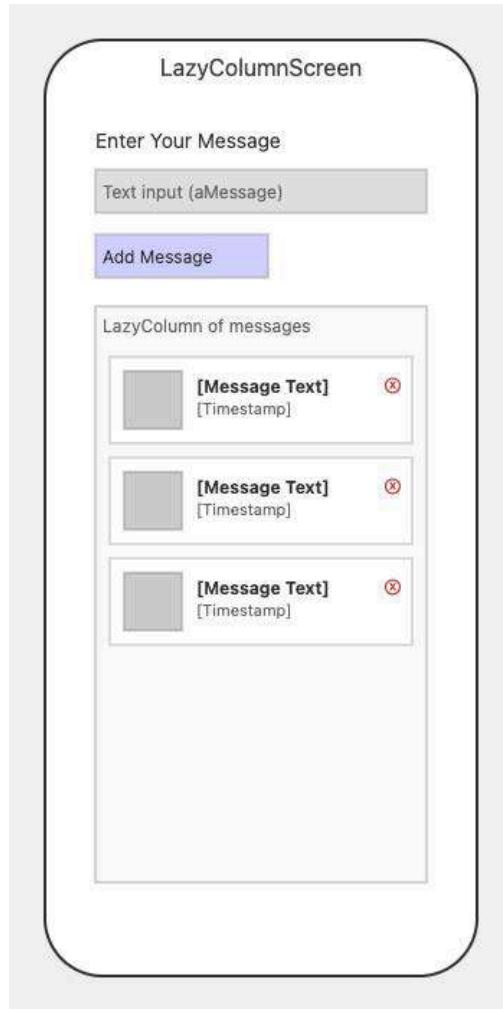
As you can see from the expected output, we need a column to vertically position the label, text field, button and list of cards. You should be able to do this yourself based on the previous week's code.

Pseudocode for UI & Logic

```
FUNCTION LazyColumnScreen (innerPadding):  
  
    INITIALIZE empty text input variable (aMessage)  
    INITIALIZE empty list of messages (messages)  
  
    DISPLAY column with padding and full-screen layout:  
  
        DISPLAY text label ("Enter Your Message")
```

```
DISPLAY text input field:  
    ON text input CHANGE:  
        UPDATE input variable (aMessage)  
  
DISPLAY button ("Add Message"):  
    WHEN CLICKED:  
        GET current timestamp  
        ADD new message (text + timestamp) to messages list  
        CLEAR input field (reset aMessage)  
  
DISPLAY scrollable list of messages (LazyColumn):  
    FOR EACH message in messages list:  
  
        DISPLAY card containing:  
            DISPLAY horizontally arranged layout (Row):  
  
                DISPLAY Image (fixed resource)  
  
                DISPLAY vertically arranged text layout (Column):  
                    DISPLAY message text (bold)  
                    DISPLAY timestamp  
  
                DISPLAY delete icon button:  
                    WHEN CLICKED:  
                        REMOVE selected message from messages list
```

**UI:** This is what our app needs to look visually (medium-fidelity mockup)



How to interpret this diagram:

1. **Text Label & Input Field** – Shows a label (“Enter Your Message”) and a gray rectangle representing the `TextField` where the user types `aMessage`.
2. **Add Message Button** – A rectangle labeled “Add Message,” indicating where the button would appear.
3. **LazyColumn of Messages** – The large rectangle is the scrollable list region. Each “card” (the white boxes) contains:
  - o A small square as an image placeholder.
  - o Bold text for the message content.
  - o Smaller text for the timestamp.
  - o A delete icon (represented by “⊗”).

====

**Logic:** Take a look at the pseudocode again. There are two new changes we need to do to the UI code we have created in previous weeks to make this work for this Lab:

1. Writing code to get the current time (in HH:mm:ss format) and displaying it in the card.
2. Storing card data (text, time) using the Message data class

We will do both requirements in the next few steps.

### 1.3c Create the Composable function for the Screen

Just like previous weeks, we need a composable function for the Screen content.

Inside of this function, declare two stateful variables: 1) for the text input from user, 2) for holding a list of Message(s).

```
@Composable
fun LazyColumnScreen(innerPadding: PaddingValues) {
    //create a variable to hold the text input
    var aMessage by remember { mutableStateOf("") }
    //create a variable to hold the list of messages
    var messages by remember { mutableStateOf(listOf<MainActivity.Message>()) }
}
```

Lets take a look at the **messages** variable declaration. What is it doing?

- You are creating a variable named `messages` whose type is inferred to be `List<MainActivity.Message>`. It holds a list of `Message` objects (coming from the `MainActivity.Message` data class).
- By using `remember { ... }`, Compose will keep track of this state **across recompositions**. Whenever a composable function reading `messages` is recomposed, it will retrieve the same state value rather than resetting it.
- `mutableStateOf(...)` creates an observable state holder. When `messages` is updated, Compose automatically re-triggers any UI that reads `messages`, causing it to refresh with the new data.

Next, let's create a column containing our UI elements (TextField, Buttons etc). Once you've finished, this is what your code should look like:



Don't forget to add imports (use Android Studio IDE's recommendations) as you go along. Don't just copy-paste lots of code and then start fixing errors, fix errors as you go along if possible!

```

1 @Composable
2 fun LazyColumnScreen(innerPadding: PaddingValues) {
3     //create a variable to hold the text input
4     var aMessage by remember { mutableStateOf("") }
5     //create a variable to hold the list of messages
6     var messages by remember { mutableStateOf(listOf<MainActivity.Message>()) }
7
8     Column(
9         modifier = Modifier
10            .fillMaxSize()
11            .padding(innerPadding)
12            .padding(16.dp)//add padding of 16 dp
13    ) {
14        // Text label for the input field
15        Text("Enter Your Message")
16        Spacer(modifier = Modifier.height(8.dp))
17        //input text field
18        TextField(
19            value = aMessage,
20            //when value change , update text variable
21            onValueChange = { aMessage = it },
22            label = { Text("Enter Message") },
23            modifier = Modifier.fillMaxWidth()
24        )
25        Spacer(modifier = Modifier.height(16.dp))
26        Button(onClick = {
27            val currentTime = SimpleDateFormat("HH:mm:ss", Locale.getDefault()).format(Date())
28            //add message to list , then clear text field
29            messages = messages + MainActivity.Message(aMessage, currentTime)
30            aMessage = ""
31        }) {
32            //add caption to button
33            Text("Add Message")
34        }
35
36        Spacer(modifier = Modifier.height(16.dp))
37

```

codesnap.dev

**Remember:** In the code above, the function starts by creating the required variables. The variable `messages` (see line 6) is an array of classes, where each cell is an instance of data class **Message**. In line 29, we create a new instance of class `Message` and add it to the list of messages. The `Message` class takes two parameters: the message payload and the current time, which is calculated in line 27.

Let's unpack this a bit more!

### 1. Formatting the timestamp

```
val currentTime = SimpleDateFormat("HH:mm:ss", Locale.getDefault()).format(Date())
```

- This grabs the current system time (`Date()`).
- Formats it into a string with hours:minutes:seconds ("HH:mm:ss") according to the device's default locale.

### 1. Adding a new message

```
messages = messages + MainActivity.Message(aMessage, currentTime)
```

- You take the existing list of messages and use `+` to append a new `Message` object onto it.
- Because you're using a `mutableStateOf` for `messages`, assigning a new list triggers a UI recomposition—updating any composables that read `messages`.
- The `Message` data class presumably holds a `text` and a `timestamp`.

### 1. Clearing the text field

```
aMessage = ""
```

- After adding the new message to the list, you set `aMessage` to an empty string.
- This will clear the text in the text field (which is also backed by a `mutableStateOf`).

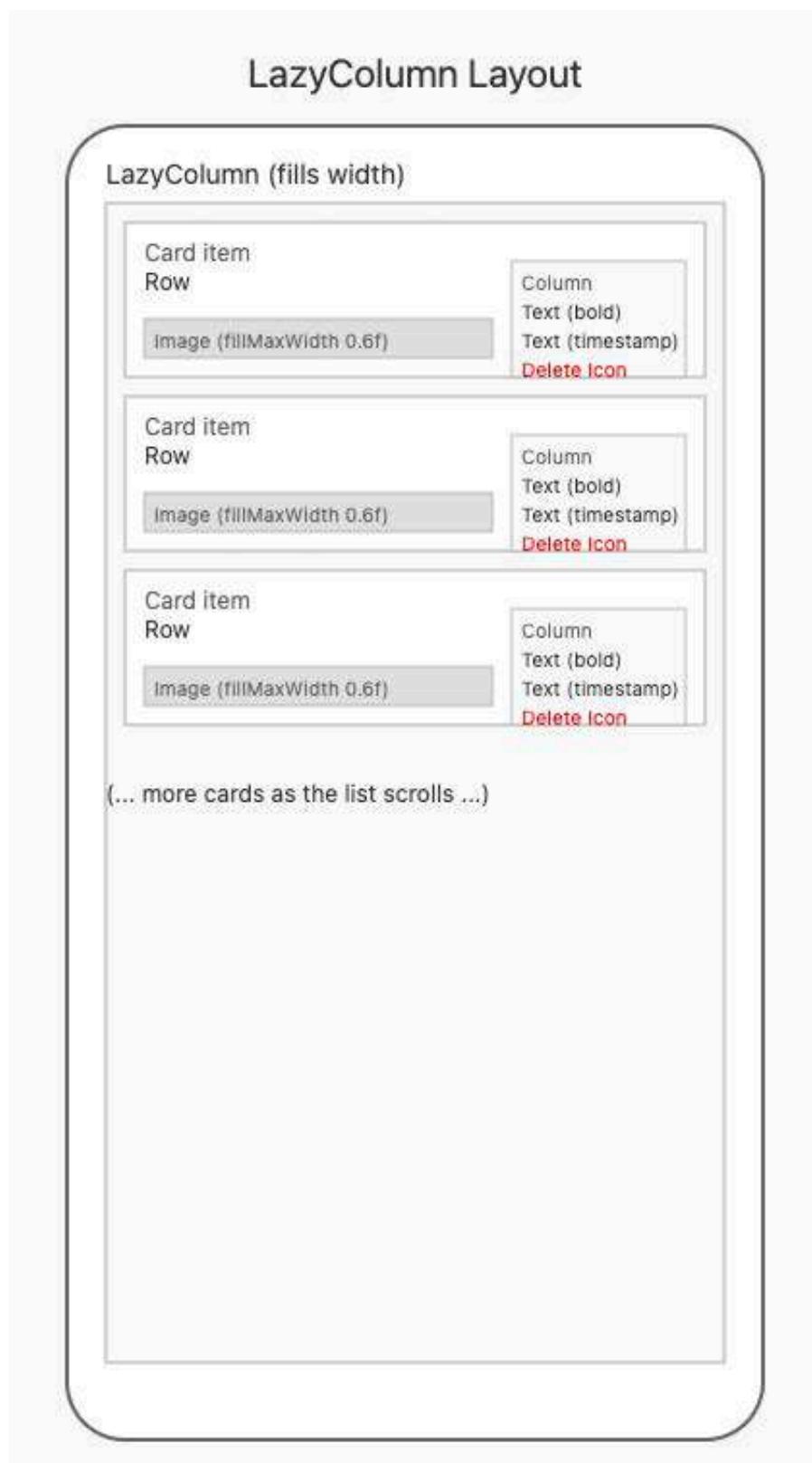
Putting it all together: when the user clicks “**Add Message**”, the code gets the current time, creates a new message with the user’s input (`aMessage`) and the timestamp, appends it to the `messages` list, and finally clears the input field. Because `messages` and `aMessage` are state variables, Compose will automatically re-render any UI displaying them.

Well done on finishing Part 2. In the next Part, we will extend our code to show the actual LazyColumn..

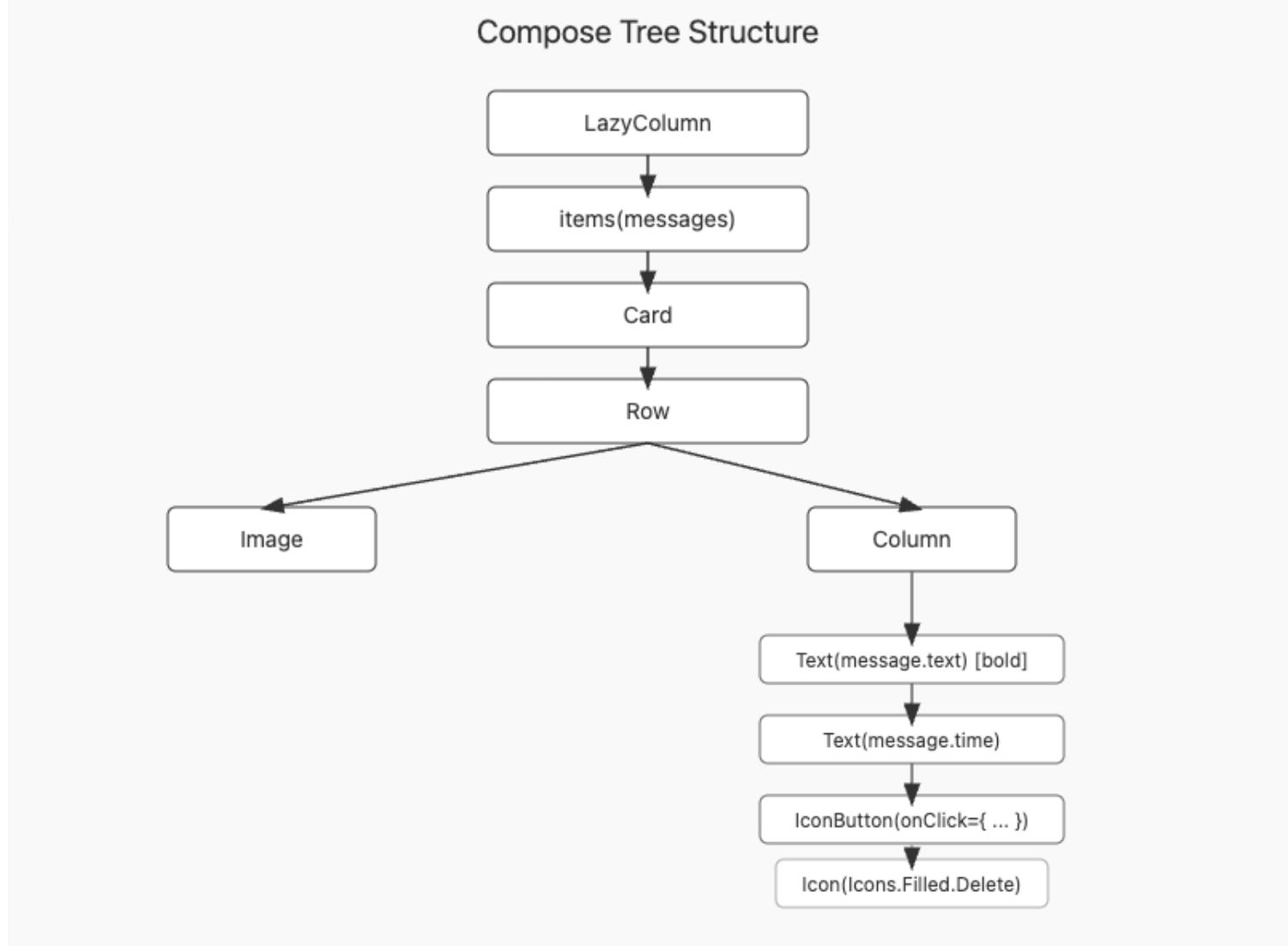
## B. Implementing LazyColumn Pt. 3

Now, it's time to create the lazy column **UI element** responsible for listing the messages.

If we try to visualise the LazyColumn, this is what we are trying to build:



If we try to think of it in the Compose UI elements, we can visualise a tree structure like this:



Here's the logic:

For each message in the messages List, we will ask LazyColumn to create a card. In LazyColumn, we can do this using the following syntax:

```
//iterate through the list using the Lambda parameter "message"
items(messages) { message ->
    Card(){
        // We can access values contained inside the message parameter here.. such as 'time' or 'text'
    }
}
```

For example, in our case, we are creating a Text display which shows the text inside each message, this is what our code will look like:

```
//iterate through the list using the Lambda parameter "message"
items(messages) { message ->
    Card(){
        // ... Other code
    }
}
```

```
// ...
Text(
    "${message.text}",
    // fontSize, fontWeightl, modifier etc..
)
}
```

So, lets start writing the full code:

```

1 LazyColumn(
2     modifier = Modifier.fillMaxWidth() {
3         //iterate through the list
4         items(messages) { message ->
5             //display each message in a card
6             Card(
7                 modifier = Modifier
8                     .fillMaxWidth()
9                     .padding(8.dp)
10            ) {
11                 Row(
12                     modifier = Modifier.padding(8.dp),
13                     horizontalArrangement = Arrangement.SpaceBetween
14                ) {
15                     Image(
16                         painter = painterResource(R.drawable.m1),
17                         contentDescription = "m1",
18                         //set the size of the image to be 60% of the width of the parent
19                         modifier = Modifier.fillWidth(0.6f)
20
21                 )
22                 Column {
23                     //display the text
24                     Text(
25                         "${message.text}",
26                         fontWeight = FontWeight.Bold,
27                         fontSize = 16.sp,
28                         modifier = Modifier.padding(8.dp)
29                     )
30                     //display the time
31                     Text("${message.time}", modifier = Modifier.padding(8.dp))
32                     IconButton(
33                         onClick = {
34                             messages = messages.filter { it != message }
35                         },
36                         modifier = Modifier.padding(8.dp)
37                     ) {
38                         //display the delete icon in red
39                         Icon(
40                             imageVector = Icons.Filled.Delete,
41                             contentDescription = "Delete",
42                             tint = Color.Red
43                         )
44                     }
45                 }
46             }
47         }
48     }
49 }
```

In the code above, we observe the following:

- Line 4 provides the list of messages to the LazyColumn UI element and starts iterating through

all of them.

- Line 6 creates a card for each message. A Card is a Material Design component that provides a container with a slightly raised appearance.
- Line 15 adds an image to the card, and line 19 sets its size to 60% of the card width.
- Line 25 adds text to show the message, while line 31 adds text to show the time.
- The icon button is to delete the current message. For the sake of simplicity, we filter the list of the message content. The best practice is to give each message a unique ID and the delete or update by ID.

## C: Navigation with Jetpack Compose

**Navigation** is the process of moving between different screens or destinations in an Android app built using Jetpack Compose. Unlike traditional Android navigation (which relies on fragments), navigation in Compose is designed to work seamlessly with composable functions, making it more intuitive and declarative.

### Key Concepts in Jetpack Compose Navigation:

- **NavController:**

- The central component that manages navigation between destinations.
- It keeps track of the backstack and allows you to navigate to new destinations.
- It provides methods to navigate to different destinations (composable screens) within your app's navigation graph.

- **NavHost:**

- A composable that acts as a container for the navigation graph.
- It displays the current destination based on the navigation state.

- **Navigation Graph:**

- A collection of composable destinations and their relationships.
- A Navigation Graph is a data structure that defines all the possible destinations (composable screens) within your app and their connections.
- It maps out how users can move from one screen to another.
- It allows you to visualize and define the relationships between different screens.

- **Destinations:**

- Composable functions that represent screens in your app.
- Each destination is associated with a unique route.

- **Route:**

- A string that uniquely identifies a destination in the navigation graph.
- Used to navigate between destinations.

What is the difference between **NavController** and **NavHost**?

<b>Feature</b>	<b>NavController</b>	<b>NavHost</b>
<b>Role</b>	The management of navigational status and logic	The composable of the current destination is displayed
<b>Function</b>	Navigation between screens is effectuated	The hosting and display of screens are performed
<b>Interaction</b>	The NavHost is directed as to what should be displayed	That which the NavController directs is displayed by the host

Let's develop an application an application that uses NavController and NavHost. Here is the expected output.

12:14



# Welcome!



home



reports



settings



1:1



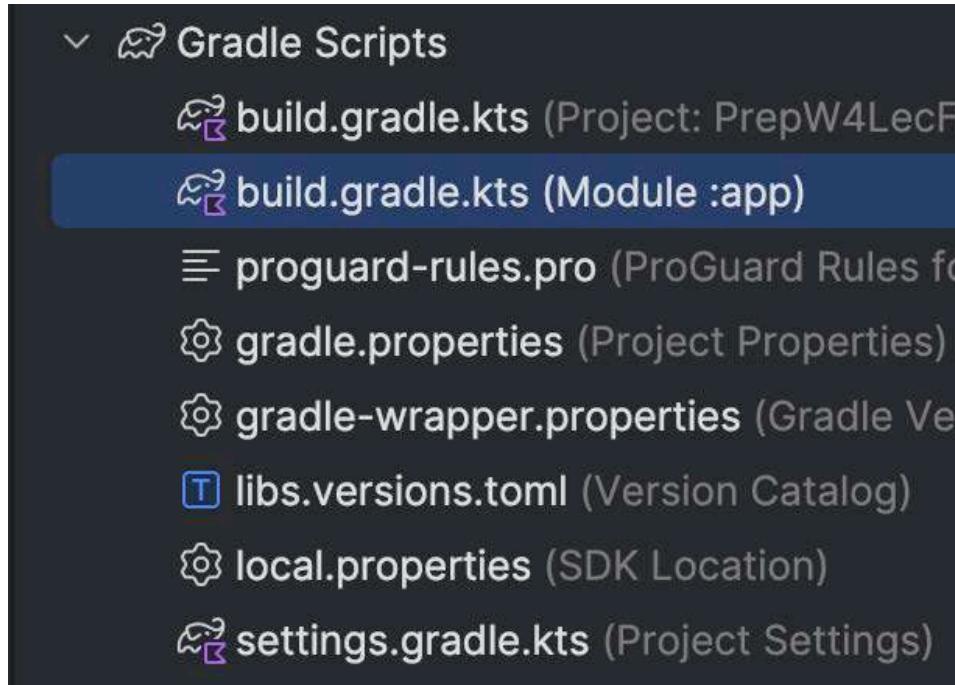
First, we need to import the Navigation Compose dependency in our gradle file.

In your **module-level** `build.gradle.kts` (the one containing your app's other Compose dependencies), add the following inside the `dependencies` block:

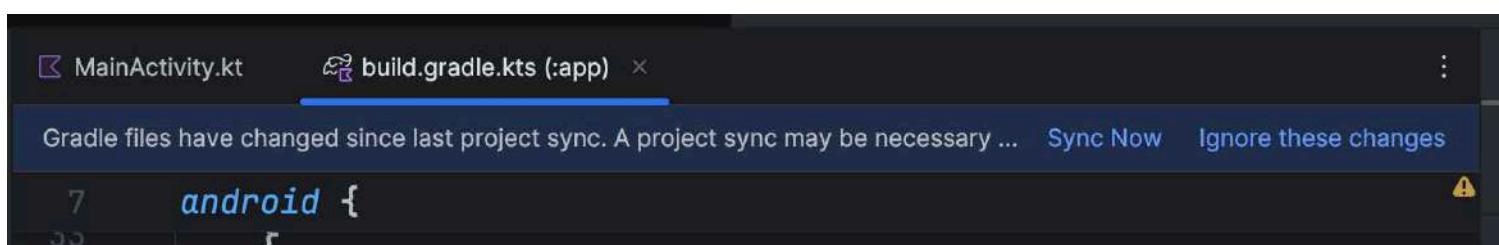
```
val nav_version = "2.8.9"  
implementation("androidx.navigation:navigation-compose:$nav_version")
```

2.8.9 is the version number. Doing this in 2 lines like this allows for easier upgrades in the future.

Remember you need to make this change to the Module: app gradle file in your Gradle Scripts section, double-check the screenshot below:



After adding the dependency, Android Studio will prompt you to Sync..



Press the "Sync now" link to sync/reload your Gradle project so the new library is downloaded and recognized.

Success! Now our navigation-related imports will work properly...

Next, go to your `MainActivity` (or whichever activity you are creating the Navigation within). In this activity, the `NavController` is instantiated **at the top level** of the composable hierarchy in your Activity file (see line 3 below).

```
setContent {  
    Week4_navTheme {  
        val navController: NavHostController = rememberNavController()  
        // Other Content  
    }  
}
```

In this lesson we are going to implement our own BottomBar (let's call this `MyBottomBar`) and our own NavHost (let's call this `MyNavHost`).

With this knowledge, lets write the rest of the code for the `setContent{}` section.

The key thing to remember is that when the NavHost composable is invoked, with the **NavController instance is being passed** as a parameter to facilitate navigation management (see line 15).

```
1 Week4_navTheme {  
2     // Initializes the NavHostController for managing navigation within the app.  
3     val navController: NavHostController = rememberNavController()  
4     Scaffold(  
5         modifier = Modifier.fillMaxSize(),  
6         bottomBar = { // defines the bottom navigation bar.  
7             MyBottomAppBar(navController)  
8         }  
9     ) { innerPadding →  
10         // Use Column to place MyNavHost correctly within Scaffold.  
11         // The padding is applied to ensure content does not overlap with system UI elements.  
12         // The innerPadding parameter provides the necessary padding values.  
13         Column(modifier = Modifier.padding(innerPadding)) {  
14             // Calls the MyNavHost composable to define the navigation graph.  
15             MyNavHost(innerPadding, navController)  
16         }  
17     }  
18 }  
19  
20 }
```

codesnap.dev

## NavHost

As we mentioned earlier, the NavHost is a composable that acts as a container for the navigation graph. It displays the current destination based on the navigation state.

In the code below, we implement our own NavHost.

Line 5 defines the composable that works as a container. Line 7 provides the NavController to the NavHost, and line 9 sets the default (i.e. start) screen. In other words, when the application starts, the

NavHost displays the default screen.

Lines 10-21 define all the destinations and their routes. For example, line 12 defines the route "home"; for this route, line 13 specifies the composable screen.

We will be manually navigating to some composable screens we will be creating in our activity (another option is to design the screens yourself in another class file and show them here, but for our purposes it is easier to define all the screens here in one file). Dont worry about this for now as we will be creating these later on (scroll to the end). All you need to know is that HomeScreen, ReportsScreen and SettingsScreens are names of composable functions.

```
// Composable function for displaying the Home screen.  
@Composable  
fun HomeScreen(innerPadding: PaddingValues) {}  
  
// Composable function for displaying the Reports screen.  
@Composable  
fun ReportssScreen(innerPadding: PaddingValues) {}  
  
// Composable function for displaying the Settings screen.  
@Composable  
fun SettingsScreen(innerPadding: PaddingValues) {}
```

```
1 // MyNavHost Composable function for navigation within the app
2 @Composable
3 fun MyNavHost(innerPadding: PaddingValues, navController: NavHostController) {
4     // NavHost composable to define the navigation graph
5     NavHost(
6         // Use the provided NavHostController
7         navController = navController,
8         // Set the starting destination to "home"
9         startDestination = "home"
10    ) {
11        // Define the composable for the "home" route
12        composable("home") {
13            HomeScreen(innerPadding) }
14        // Define the composable for the "reports" route
15        composable("reports") {
16            ReportsScreen(innerPadding) }
17        // Define the composable for the "settings" route
18        composable("settings") {
19            SettingsScreen(innerPadding)
20        }
21    }
22 }
23 }
24 }
```

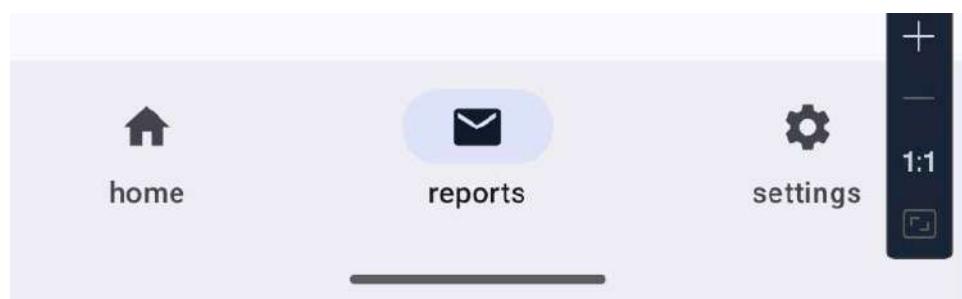
codesnap.dev

**Troubleshooting:** "I'm having issues with composable() being recognised

▶ Expand

## Bottom App Bar

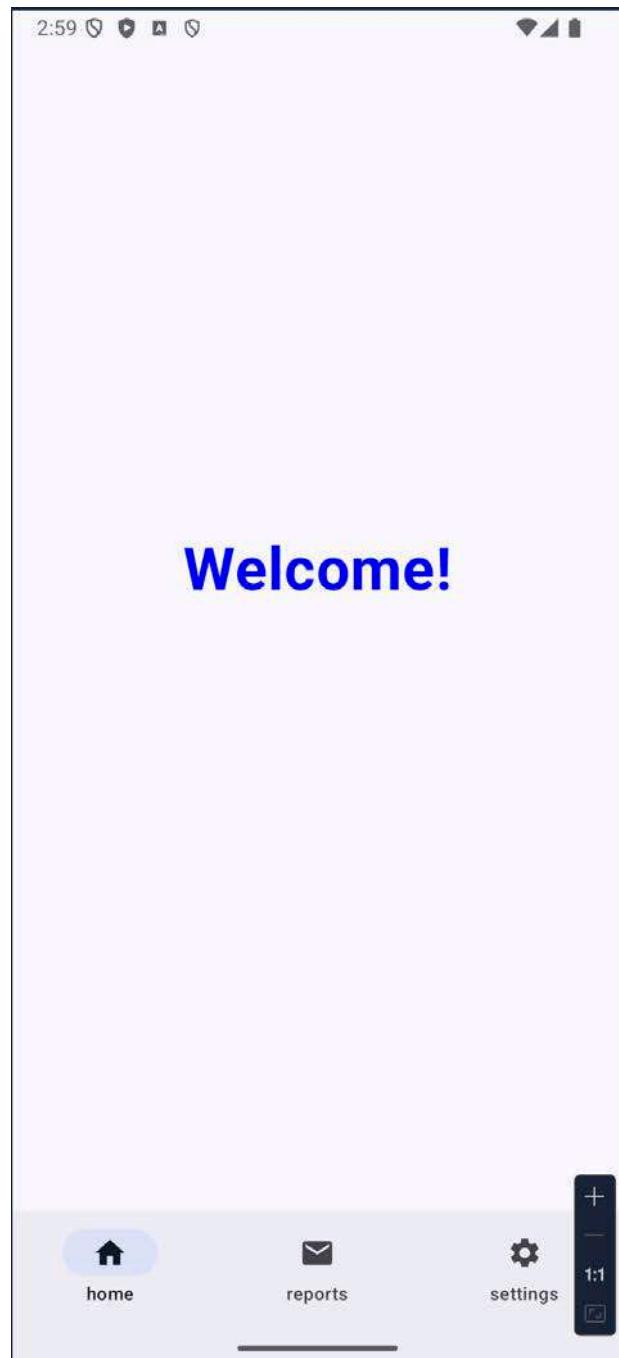
This function creates the bottom app bar that shows the three icons.



The component takes the NavController as a parameter, enabling it to respond to user selections within the bottom navigation bar and navigate to the corresponding screens (see line 38). The bottom navigation bar's items are defined by an initial list (lines 7-11). The component then iterates over this list, generating a NavigationBarItem for each item and configuring it with an appropriate icon (lines 20-27).

```
1 // Composable function for creating the bottom navigation bar.
2 @Composable
3 fun MyBottomAppBar(navController: NavHostController) {
4     // State to track the currently selected item in the bottom navigation bar.
5     var selectedItem by remember { mutableStateOf(0) }
6     // List of navigation items: "home", "reports", "settings".
7     val items = listOf(
8         "home",
9         "reports",
10        "settings"
11    )
12    // NavigationBar composable to define the bottom navigation bar.
13    NavigationBar {
14        // Iterate through each item in the 'items' list along with its index.
15        items.forEachIndexed { index, item →
16            // NavigationBarItem for each item in the list.
17            NavigationBarItem(
18                // Define the icon based on the item's name.
19                icon = {
20                    when (item) {
21                        // If the item is "home", show the Home icon.
22                        "home" → Icon(Icons.Filled.Home, contentDescription = "Home")
23                        // If the item is "reports", show the Email icon.
24                        "reports" → Icon(Icons.Filled.Email, contentDescription = "Reports")
25                        // If the item is "settings", show the Settings icon.
26                        "settings" → Icon(Icons.Filled.Settings, contentDescription = "Settings")
27                    }
28                },
29                // Display the item's name as the label.
30                label = { Text(item) },
31                // Determine if this item is currently selected.
32                selected = selectedItem == index,
33                // Actions to perform when this item is clicked.
34                onClick = {
35                    // Update the selectedItem state to the current index.
36                    selectedItem = index
37                    // Navigate to the corresponding screen based on the item's name.
38                    navController.navigate(item)
39                }
40            )
41            // Close NavigationBarItem.
42        }
43    }
44}
45
46
47 }
```

This screen will be inflated if the user selects the home route.

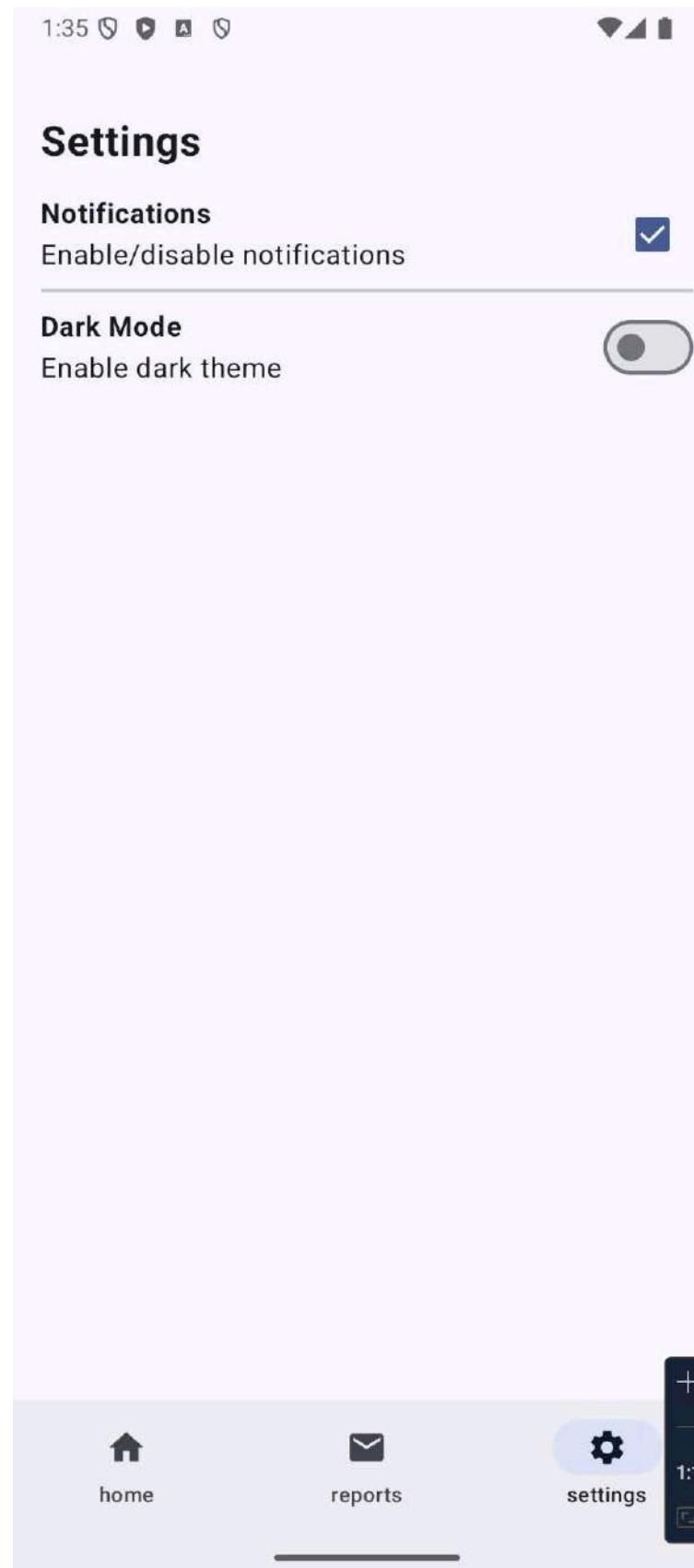


```
1 // Composable function for displaying the Home screen.
2 @Composable
3 fun HomeScreen(innerPadding: PaddingValues) {
4     Column(
5         modifier = Modifier
6             .padding(innerPadding)
7             .fillMaxSize(),
8         verticalArrangement = Arrangement.Center,
9         horizontalAlignment = Alignment.CenterHorizontally
10    ) {
11        Text(
12            text = "Welcome!",
13            fontSize = 40.sp,
14            fontWeight = FontWeight.Bold,
15            color = Color.Blue
16        )
17    }
18 }
```

codesnap.dev

## SettingsScreen

This screen will be inflated if the user selects the settings route.



As you can see, a column is needed to list all the options vertically (see line 4 below).

- It uses a `Column` to stack the UI elements vertically.
- The screen starts with a "Settings" title, styled with bold text and a larger font size.
- Below the title are setting options, each represented by the `SettingOption` composable.
- The state of this checkbox is managed using `mutableStateOf` and `remember` for proper state management.

- A `HorizontalDivider` separates the settings options for visual clarity.
- Similar to the `Checkbox`, the state of the `Switch` is managed using `mutableStateOf` and `remember`.
- Each `SettingOption` consists of a title, description, and a custom UI element for interaction.
- The `innerPadding` parameter ensures that the content is properly padded, respecting the screen edges.
- The `.padding(16.dp)` modifier adds additional padding around the content for aesthetic spacing.
- `HorizontalArrangement.SpaceBetween` and `Alignment.CenterVertically` are used within `SettingOption` to space items evenly.

```

1 // Composable function for displaying the Settings screen.
2 @Composable
3 fun SettingsScreen(innerPadding: PaddingValues) {
4     Column(
5         modifier = Modifier
6             .padding(innerPadding)
7             .padding(16.dp)
8     ) {
9         Text(
10            // Settings title text.
11            text = "Settings",
12            fontSize = 24.sp,
13            fontWeight = FontWeight.Bold,
14            modifier = Modifier.padding(bottom = 16.dp)
15        )
16        // add a new setting option. See the SettingOption composable function below.
17        //
18        SettingOption(title = "Notifications", description = "Enable/disable notifications") {
19            val checkedState = remember { mutableStateOf(true) }
20            Checkbox(
21                checked = checkedState.value,
22                onCheckedChange = { checkedState.value = it }
23            )
24        }
25        // Add a horizontal divider to separate the settings options.
26        HorizontalDivider(
27            modifier = Modifier.padding(vertical = 8.dp),
28            thickness = 2.dp,
29        )
30        // Add another setting option.
31        SettingOption(title = "Dark Mode", description = "Enable dark theme") {
32            var switchCheckedState by remember { mutableStateOf(false) }
33            Switch(
34                checked = switchCheckedState,
35                onCheckedChange = { switchCheckedState = it }
36            )
37        }
38    }
39 }
```

## SettingOption

```
1 // Composable function for creating a setting option row.
2 @Composable
3 fun SettingOption(
4     title: String,
5     description: String,
6     trailingContent: @Composable () -> Unit
7 ) {
8     Row(
9         modifier = Modifier.fillMaxWidth(),
10        horizontalArrangement = Arrangement.SpaceBetween,
11        verticalAlignment = Alignment.CenterVertically
12    ) {
13     Column {
14         // Displays the title and description of the setting option.
15         Text(text = title, fontWeight = FontWeight.Bold)
16         Text(text = description)
17     }
18     // Displays the trailing content of the setting option.
19     trailingContent()
20 }
21 }
```

codesnap.dev

## ReportScreen

- This function displays a list of reports using a LazyColumn (see line 24 below).
- Each report is represented as a CardItem, containing a title, text, and an image (see line 1 below).
- The card items list defines the data for each report.
- A LazyColumn displays the reports efficiently, only compositing and laying out the items visible on the screen.
- Each Card displays the report's title in bold, an image, and the report's text (see line 36).
- The `innerPadding` parameter provides padding around the LazyColumn, ensuring the content is not obscured by system UI elements like the navigation bar (see line 26).
- Each Card has its own padding to create space between the reports (see line 31).

- The Column inside each Card has padding to create space around the title, image, and text.
- The image uses `fillMaxWidth()` to ensure it spans the width of the Card.
- The `painterResource` function loads the image from the drawable resources.
- The text within each Card is displayed using the Text composable.

**Report 1**

This is the text for Report 1.

**Report 2**

This is the text for Report 2.

**Report 3**

home



reports



settings



Here is the source code:

```
1 data class CardItem(val title: String, val text: String, val image: Int)
2
3 // Composable function for displaying the Reports screen.
4 @Composable
5 fun ReportsScreen(innerPadding: PaddingValues) {
6     val cardItems = listOf(
7         CardItem(
8             title = "Report 1",
9             text = "This is the text for Report 1.",
10            image = R.drawable.m1
11        ),
12        CardItem(
13            title = "Report 2",
14            text = "This is the text for Report 2.",
15            image = R.drawable.m2
16        ),
17        CardItem(
18            title = "Report 3",
19            text = "This is the text for Report 3.",
20            image = R.drawable.m3
21        )
22    )
23
24    LazyColumn(
25        modifier = Modifier
26            .padding(innerPadding)
27    ) {
28        items(cardItems) { item →
29            Card(
30                modifier = Modifier
31                    .padding(16.dp)
32            ) {
33                Column(
34                    modifier = Modifier.padding(16.dp)
35                ) {
36                    Text(text = item.title, fontWeight = FontWeight.Bold)
37                    Image(
38                        painter = painterResource(id = item.image),
39                        contentDescription = "Card Image",
40                        modifier = Modifier.fillMaxWidth()
41                    )
42                    Text(text = item.text)
43                }
44            }
45        }
46    }
47}
48 }
```

## D: Sharing Data

This article will discuss how to share data with other applications.

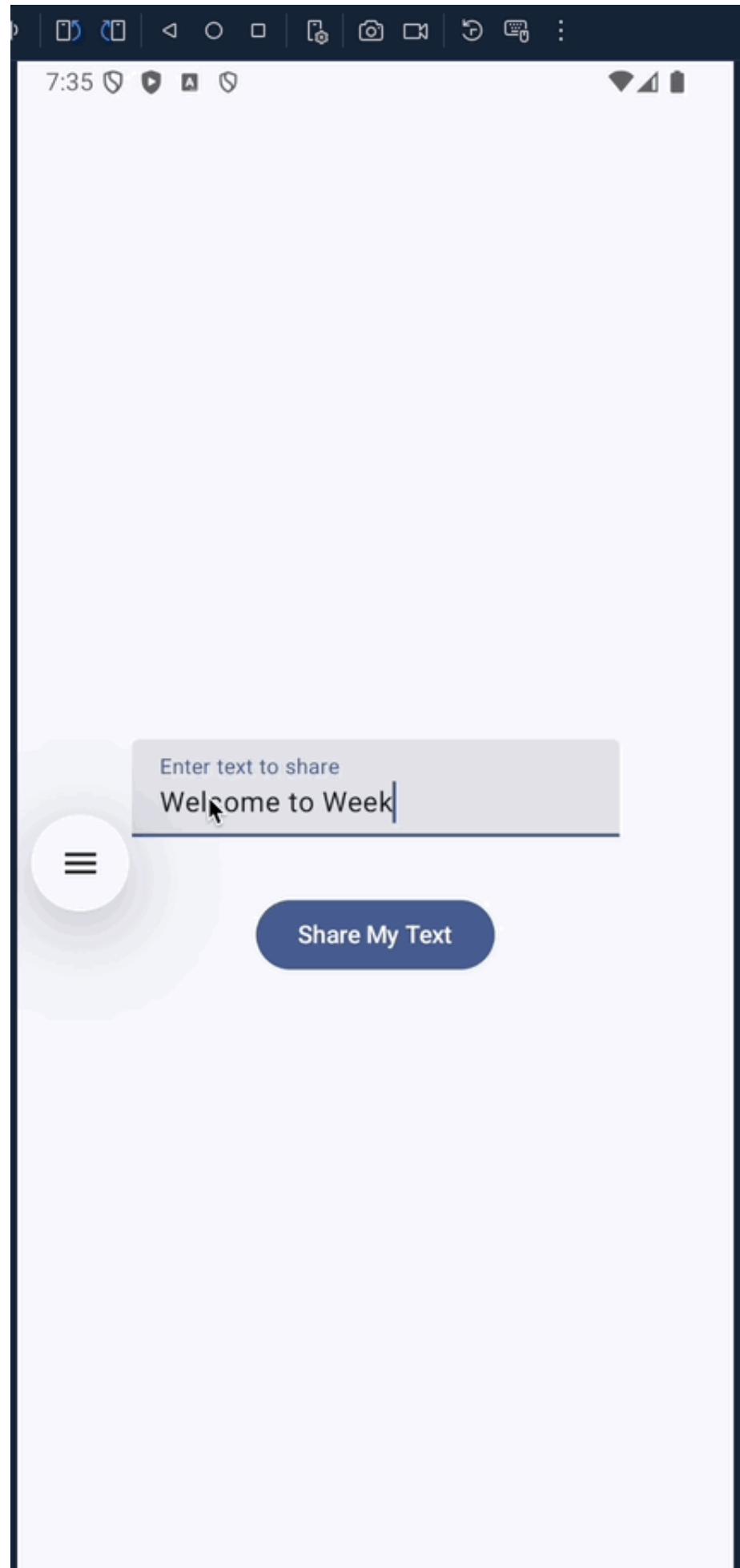
 This will help you implement the "Share with someone" button in the NutriTrack app for your design+develop assignment (30%).

Core Concept:

- Intents: Intents are fundamental to Android's inter-app communication. They act as messaging objects that allow you to request actions from other components (like activities) in your app or other apps.
- ACTION\_SEND: This standard intent action signifies the intent to share data.
- MIME Types: MIME (Multipurpose Internet Mail Extensions) types specify the format of the data you're sharing (e.g., text/plain, image/jpeg, video/mp4).
- Extras: Intents can carry data through "extras." EXTRA\_TEXT is commonly used for text, but there are others for different data types.

Let's build an application that reads user data and shares it with other applications.

Here is the expected output.



I will add a column to hold the text field and a button to give this simple layout. This should already be familiar to you, so let's re-create a simple UI with a TextField and a Button inside a Column..

```
1 Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
2     Column(
3         // Define the layout modifier for the Column
4         modifier = Modifier
5             // Apply inner padding to the column
6             .padding(innerPadding)
7             // Make the column take up all available space
8             .fillMaxSize()
9             // Add padding of 16.dp around the column
10            .padding(16.dp),
11        // Center the content horizontally
12        horizontalAlignment = Alignment.CenterHorizontally,
13        // Center the content vertically
14        verticalArrangement = Arrangement.Center
15    ) {
16        // Initialize a mutable state to hold the text to be shared
17        var shareText by remember { mutableStateOf("") }
18        // Create a TextField to allow user to input the text to be shared
19        TextField(
20            // Bind the text field's value to the shareText state
21            value = shareText,
22            // Update the shareText state when the user inputs text
23            onValueChange = { shareText = it },
24            // Label for the text field
25            label = { Text("Enter text to share") }
26        )
27        Spacer(modifier = Modifier.padding(16.dp))
28
29        // Create a Button to trigger the sharing action
30        // When clicked, the shareIntent will be started to share the text
31        Button(onClick = {})
32    }
33}
```

codesnap.dev

Now, inside the button, let's start the activity responsible for sharing my data with other applications. This is where all the action happens!

```
1   Button(onClick = {
2       //create a intent to share the text
3       val shareIntent = Intent(ACTION_SEND)
4       //set the type of data to share
5       shareIntent.type = "text/plain"
6       //set the data to share, in this case, the text
7       shareIntent.putExtra(Intent.EXTRA_TEXT, shareText)
8       //start the activity to share the text, with a chooser to select the app
9       startActivity(Intent.createChooser(shareIntent, "Share text via"))
10
11   }) {
12     Text("Share My Text")
13
14 }
```

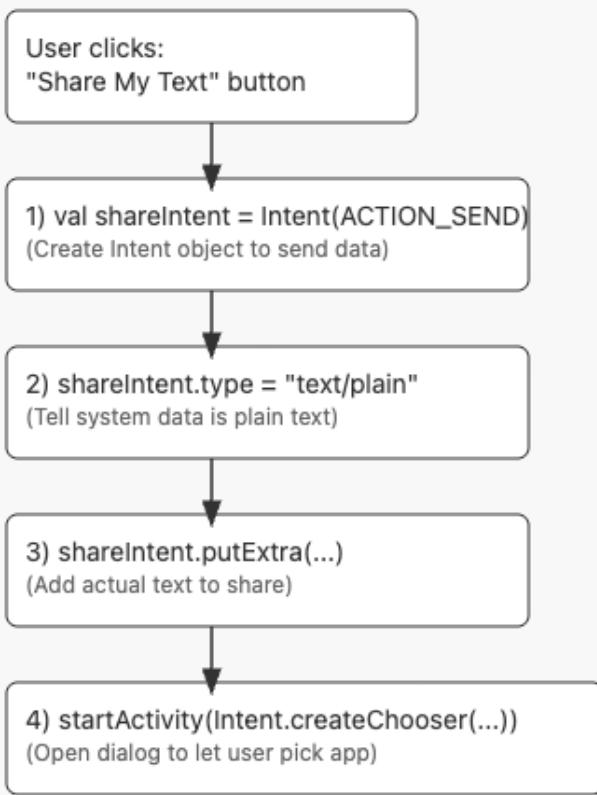
codesnap.dev

The code above handles the sharing functionality when the "Share My Text" button is clicked.

- **Step 1:** `val shareIntent = Intent(ACTION_SEND)` : An Intent object named `shareIntent` is created. An Intent is a messaging object used to request an action from another app component. `ACTION_SEND` is a constant indicating that this Intent is intended to send data to another activity.
- **Step 2:** `shareIntent.type = "text/plain"` : The type of data being shared is set to "text/plain". This tells the system that the data is plain text, and it should look for apps that can handle this type of data.
- **Step 3:** `shareIntent.putExtra(Intent.EXTRA_TEXT, shareText)` : This line adds the actual text to be shared to the Intent. - `putExtra` is used to add extra data to the Intent. - `Intent.EXTRA_TEXT` is a constant key used to identify the text data in the Intent. - `shareText` is the variable holding the text the user wants to share.
- **Step 4:** `startActivity(Intent.createChooser(shareIntent, "Share text via"))` : This line starts the sharing process. - `Intent.createChooser` creates a chooser dialog that allows the user to select which app they want to use to share the text. - `shareIntent` is the Intent containing the data and the type of data to be shared. - "Share text via" is the title of the chooser dialog, displayed to the user. - `startActivity` is used to start a new activity. In this case, it starts the chooser activity, which will then start the activity of the selected app.

**In summary**, when the button is clicked, it constructs an intent, specifies that it's sharing text, adds the text to the intent, and then opens a dialog to let the user choose the app to share it with.

## Android "Share" Flow



---

## E: Challenge Exercises

1. Update the LazyColumn application to add a unique ID for each card. Update the delete function to delete by ID instead of content.
2. Add two more screens (and hence two more icons to the bottom app bar) representing the About Us (or Contact Us) and Search Product. Add dummy content for these two screens.
3. Upgrade the Data sharing application to share images instead of text.

# FIT2081 Week 5

## Navigation & Intents

# Source Code

Download Source Code here:

 [W5LectureViewModelSimple.zip](#)

The key new concept this week is the ViewModel, see details here:

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.setValue
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

/**
 * ViewModel for managing and exposing UI state related to a counter feature.
 *
 * This class demonstrates how to use:
 * - Jetpack ViewModel for state management
 * - `mutableStateOf` for Compose reactive state
 * - Kotlin coroutines (`viewModelScope.launch`) for background tasks
 */
class CounterViewModel : ViewModel() {

    /**
     * The current value of the counter.
     *
     * `mutableStateOf` creates a state object that can be observed by Jetpack Compose.
     * This allows Compose to automatically recompose UI when the value changes.
     * `by` is Kotlin's delegated property syntax, which simplifies state access.
     */
    var counter by mutableStateOf(0)
        private set // restricts modification from outside the ViewModel

    /**
     * Indicates whether a background operation (like data fetching) is currently running.
     *
     * Can be used to show loading indicators in the UI.
     */
    var isLoading by mutableStateOf(false)
        private set

    /**
     * Increments the counter by 1.
     *
     * This is a simple synchronous operation, useful for UI interactions (e.g., tapping a button).
     */
    fun increment() {
```

```
        counter++
    }

    /**
     * Simulates fetching data from a remote source or performing a time-consuming task.
     *
     * This function demonstrates:
     * - Launching coroutines in `viewModelScope`, which is lifecycle-aware.
     * - Using `delay()` to simulate asynchronous delay.
     * - Updating Compose state during and after the background task.
     */
    fun fetchData() {
        viewModelScope.launch {
            isLoading = true // show loading spinner in UI
            delay(2000) // simulate network delay (2 seconds)
            counter += 10 // simulate data change from fetch
            isLoading = false // hide loading spinner
        }
    }
}
```

Dont forget to add the ViewModel dependency to build.gradle.kts (module :app)

```
dependencies {
    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.6.2")
}
```

**FIT2081**  
**Mobile application**  
**development (MAD)**



## Introduction to MVVM & ViewModel

**Week 5**

**Delvin Varghese**

Welcome to Week 5 of Mobile Applications Development. Today we'll explore a crucial architectural pattern in modern Android development: the Model-View-ViewModel (MVVM) architecture.

## Checklist

- Assignment 1 nearly submitted!
- Know how to debug basic issues in Android Studio

## **Announcement:**

**In-semester quiz (20%) on:  
W12, Monday 13:00**

**A2 - App Critiques (10%)  
will be released this week!**

## Learning objectives for today

1. Understand MVVM architecture principles
2. Explain the ViewModel lifecycle
3. Implement ViewModels for UI state management
4. Connect ViewModels to Jetpack Compose
5. Apply these concepts in your projects



## The problem with our apps so far...

Imagine you're developing a fitness app that tracks workout history. What are the different things the app needs to be able to do?

1. Handle user input (adding new workouts)
2. Process data (calculating calories, tracking progress)
3. Update the UI accordingly (showing workout history)
4. *Maintain its state when users rotate their device without reloading data*



Before we dive in, let's consider a real-world scenario: Imagine you're developing a fitness app that tracks workout history. The app needs to:

- Handle user input (adding new workouts)
- Process data (calculating calories, tracking progress)
- Update the UI accordingly (showing workout history)
- Maintain its state when users rotate their device without reloading data

How would you structure your code to handle these requirements efficiently?

This is where MVVM architecture becomes essential. MVVM helps separate UI logic from business logic, making your applications more maintainable, testable, and resistant to common Android lifecycle challenges.

# Your app is easily triggered!

## 💡 Dark Mode / UI Mode Change

- Compose will recompose any Composable that uses values from the theme
- Example: `MaterialTheme.colors`, `isSystemInDarkTheme()`, etc.

**Android Kills your app because after a long pause because:**

- The app is backgrounded for a while.
- The system is low on memory.
- Battery or background restrictions are applied.



# **1. MVVM**

## The Problem with Traditional Android Architecture

Traditional approach puts all logic in

Activities/Fragments leading to:

- "God Activities" with thousands of lines of code
- Data loss during configuration changes
- Difficult testing and maintenance

### God Activity Architecture — One Architecture To Rule Them All

Taylor Case · Editor  
3m read · Oct 1, 2019



### When and how to prevent a god activity in Android

[Ask Question](#)

Asked 8 years, 2 months ago · Modified 8 years, 2 months ago · Viewed 1k times

Part of Mobile Development Collective

When developing Android applications, many beginners place all their code in Activities or Fragments. This approach leads to several critical issues:

First, it creates "Massive View Controller" (or "God Activity") classes with thousands of lines of code handling UI, data, and business logic. These become challenging to debug, test, and maintain.

Second, when configuration changes occur (like screen rotation), Activities are destroyed and recreated. Without proper architecture, this leads to data loss and unnecessary data reloading.

Third, testing becomes nearly impossible when all logic is tightly coupled to Android framework components.

## Recap

Android as Terminator!

- **Data loss during configuration changes**

Jetpack Composables

- Everything is repainted by Android when activity is re-rendered...



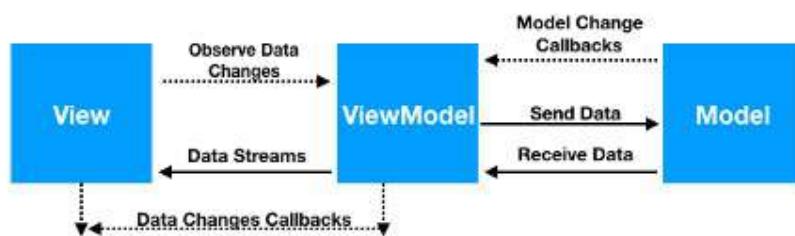
# What is MVVM?

Architectural pattern with three components:

**Model:** Data and business logic

**View:** UI elements and user interactions

**ViewModel:** Bridge between Model and View



Source: [codingwithevan.com](https://codingwithevan.com)

MVVM (Model-View-ViewModel) is an architectural pattern that separates an application into three distinct components:

1. **Model:** Represents data and business logic. This includes data classes, repositories, and other components that handle data operations.
2. **View:** Displays the UI and captures user interactions. In Android, this could be Activities, Fragments, or Composable functions.
3. **ViewModel:** Acts as a bridge between the Model and View. It prepares data for display and handles UI logic without direct references to View components.

The key advantage of MVVM is the separation of concerns. The ViewModel doesn't know about the View, which makes it testable and reusable. The View observes state changes in the ViewModel without tightly coupling to it.

```
class CounterViewModel: ViewModel() {
    var counter by mutableStateOf(0)
        private set

    var isLoading by mutableStateOf(false)
        private set

    fun increment() {
        counter++
    }

    fun fetchData() {
        viewModelScope.launch {
            isLoading = true // Indicate that data loading has started
            delay(2000) // Simulate a network request
            counter += 10 // Update count after data fetch
            isLoading = false // Indicate that data loading has finished
        }
    }
}
```

In this example, the ViewModel exposes an immutable state object that the View can observe, and provides a function to modify that state. This creates a unidirectional data flow pattern that makes your application more predictable and easier to debug.

MVVM (Model-View-ViewModel) is an architectural pattern that separates an application into three distinct components:

1. **Model:** Represents data and business logic. This includes data classes, repositories, and other components that handle data operations.
2. **View:** Displays the UI and captures user interactions. In Android, this could be Activities, Fragments, or Composable functions.
3. **ViewModel:** Acts as a bridge between the Model and View. It prepares data for display and handles UI logic without direct references to View components.

The key advantage of MVVM is the separation of concerns. The ViewModel doesn't know about the View, which makes it testable and reusable. The View observes state changes in the ViewModel without tightly coupling to it.

## Benefits: MVVM

- Separation of concerns
- Survives configuration changes
- Improved testability
- Code reusability
- UI state management

Let's emphasize the key benefits of using ViewModels:

- **Lifecycle Awareness:** ViewModels are designed to survive configuration changes, ensuring your data persists during events like screen rotations.
- **Persistence:** They retain data without requiring reloads or recomputations, improving performance and user experience.
- **Separation of Concerns:** By keeping business logic and data handling separate from the view, your code becomes more organized and maintainable.
- **State Management:** ViewModels provide a centralized place to manage UI state, making your application's behavior more predictable.

## **2. ViewModel**

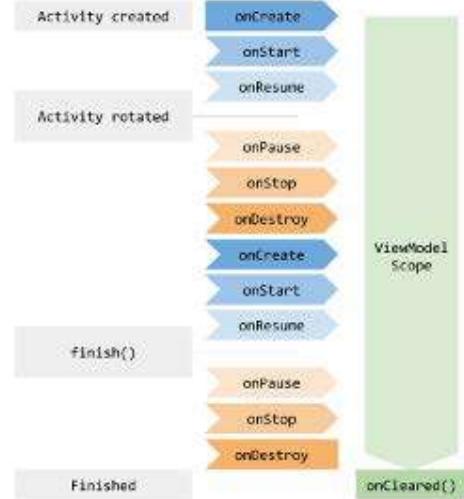
# ViewModel Lifecycle

**Created:** When first requested

**Active:** During UI controller lifecycle

**Cleared:** When UI controller permanently destroyed

```
override fun onCleared() {  
    super.onCleared()  
    // Cleanup logic here  
}
```



The lifecycle of a ViewModel is tied directly to its scope. A ViewModel remains in memory until the ViewModelStoreOwner to which it is scoped disappears. This may occur in the following contexts:

When an activity finishes, a fragment detaches, and a Navigation entry is removed from the back stack, ViewModels are a great solution for storing data that survives configuration changes.

The figure below illustrates the various lifecycle states of an activity as it undergoes a rotation and then is finished. The illustration also shows the lifetime of the ViewModel next to the associated activity lifecycle. This particular diagram illustrates the states of an activity. The same basic states apply to the lifecycle of a fragment.

This separation is what allows ViewModels to preserve state across configuration changes. When the screen rotates, the Activity is destroyed and recreated, but the ViewModel remains intact.

It's important to note that ViewModels are not preserved during process death (when the system kills your app in the background). For those scenarios, you still need to use `onSaveInstanceState()` or other persistence mechanisms.

The ViewModel lifecycle in Jetpack Compose aligns with the lifecycle of the composable's host (either an Activity or a Fragment) that it is scoped to.

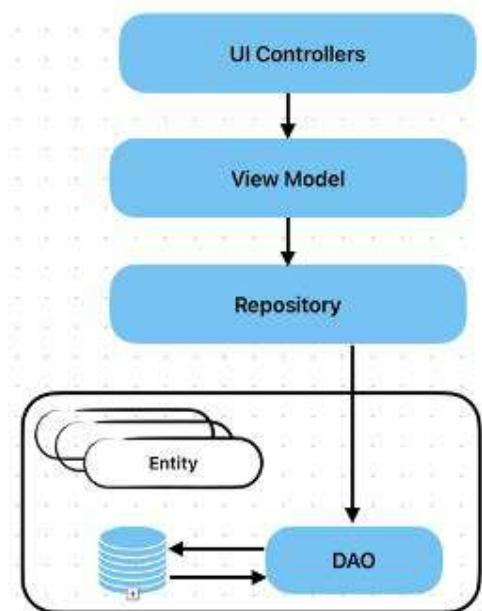
### Key Lifecycle Characteristics:

- **Creation:**
  - When you use `ViewModel()` within a composable, Jetpack Compose ensures that the ViewModel instance is created once and survives configuration changes (like screen rotations).
  - The ViewModel is typically scoped to the lifecycle of the Activity or Fragment, meaning it will persist as long as that host exists.
- **Survival Across Configuration Changes:**
  - If the Activity or Fragment is recreated (e.g., due to a configuration change like device rotation), the same ViewModel instance is returned by `viewModel()`. This allows you to retain data and avoid unnecessary recomputation or data loss.
- **Cleanup (`onCleared`):**
  - When the host's lifecycle ends permanently (e.g. when the user navigates away and the Activity is destroyed), the ViewModel is cleared, and its `onCleared()` method is called.
  - You can override `onCleared()` in your ViewModel to clean up resources, like cancelling coroutines or closing database connections.

The `onCleared()` method is called when the ViewModel is being destroyed. This is where you should clean up any resources to prevent memory leaks.

## Where to next?

- Apply MVVM in lab exercises
- Combine with Room Database (next week)
- Integrate with NutriTrack for next Assignment (Advanced).





**THE END\_**

# FIT2081 - Week 5 Lecture Notes

## Topic: Introduction to MVVM & Preparing for Databases

---

### Welcome & Context

---

- Congrats on making it to **Week 5!** 
  - Android development is complex—this is likely your first time dealing with the **full system stack**.
  - It's okay if things have felt difficult—many students report that **Weeks 5–6** are when things finally begin to "click".
  - **Stick with it:** By Week 12, you'll be able to publish your own apps to the Play Store.
- 

### Shift in Focus: From UI to Architecture

---

- Next week: You'll be learning about **Room**, Android's SQL database.
  - This week: Prepping for that by exploring **software architecture**, specifically the **MVVM pattern**.
- 

### Why MVVM? (Model - View - ViewModel)

---

#### The Problem With Our Current Approach

So far, we've written: - All logic - All UI - All state

...inside a single `MainActivity.kt` file.

#### Issues:

- Hard to **scale or maintain**
- Fragile: **Data loss on rotation** or config changes
- Poor **team collaboration**

- Results in "**God Activity**" – a well-known anti-pattern
- 

## 💥 Why We Lose Data (Real-World Examples)

When any of the following happen:  
- Orientation change  
- System night mode toggle  
- Backgrounding the app  
- Low memory or battery optimization

...Jetpack Compose **repaints** all your composables. You lose any state saved inside them.

---

## ✓ The Solution: MVVM Architecture

---

### MVVM = Model - View - ViewModel

Component	Role	Android Examples
<b>Model</b>	Business logic, data, repositories	Room DB, SharedPreferences, API clients
<b>View</b>	UI layer	Composables, NavHost, Screens
<b>ViewModel</b>	Bridge: UI ↔ Data. Holds and exposes state	<code>MyViewModel : ViewModel()</code>

---

## ⭐ Benefits of MVVM

- **Separation of concerns:** Cleaner code, easier testing
  - **Preserves state:** e.g., counter doesn't reset on screen rotate
  - Easier to **upgrade** data source (e.g., switch to Firebase)
  - Enables **async work** (like network calls) cleanly
  - Aligns with **modern Android architecture**
- 

## 🛠 Live Demo Recap: ViewModel Refactor of Counter App

---

### Key Changes:

- Move logic to `CounterViewModel.kt`

- Use `mutableStateOf` + `private set` to make state read-only
- UI reads from `viewModel.counter`, doesn't update it directly
- Use `viewModelScope.launch {}` for async background tasks (e.g., simulating network call)

## Bonus:

- State survives rotation and system changes!
  - UI thread remains unblocked (loading spinner appears)
- 

## Debug Tip: Layout Inspector

Use Android Studio's **Layout Inspector**: - View composable hierarchy - See recomposition counters - Debug why a value isn't displaying correctly

---



## Why This Matters for Your Assignment

- Week 11 assignment will require proper MVVM structure
  - Easier to plug in a **Room DB** when model & logic are separate
  - You'll need to:
    - Create ViewModel classes
    - Keep UI separate from logic
    - Implement clean data flow
- 



## ViewModel and Lifecycle Awareness

- ViewModel **outlives activity recreation** (e.g., rotation)
  - Destroyed only when activity is permanently removed
  - Ideal for:
    - User inputs
    - Background fetches
    - Preserving UI state across screen changes
- 



## Summary

Concept	Why It Matters
MVVM Architecture	Better organization, scalability
ViewModel	Persistent, lifecycle-aware state
Separation of Concerns	Easier debugging, testing, team workflows
Async with viewModelScope	Smooth user experience
Preparing for Room	Database will plug into Model layer cleanly

## Coming Up Next Week

- **Room database** (SQLite abstraction)
- Persist your data to disk
- Connect database to your ViewModel

## Practice Before the Lab

- Refactor one of your apps to use `ViewModel`
- Track state (like a counter)
- Rotate the screen — does it persist?
- Test state updates using Layout Inspector

## Announcements

- **Advanced App Assignment:** Due Week 11
- **Optional HD Interview:** In labs next week
- **Coding support & consultations available**

# Lab: ViewModel | Database Pt. 1

---

Important Announcement for Week 5 (Please Read Carefully)

# Introduction

## Week 5 Lab: Persistent Storage with Room Database & MVVM Architecture

### INTRODUCTION

**Learning Focus:** This week, you'll build the foundation for creating apps that store and manage data—a crucial skill for nearly all professional mobile applications.

**Why This Matters:** Most real-world apps need to save data between sessions and efficiently manage information: social media apps store posts and user profiles, fitness apps track workout history, and even simple note-taking apps must persist user content. The skills you learn today will directly apply to your NutriTrack assignment.

**Today's Objectives:** By the end of this lab, you will:

1. Understand database persistence in Android using Room
2. Implement the MVVM (Model-View-ViewModel) architecture
3. Create a complete data flow from UI to database and back
4. Apply Kotlin coroutines for handling asynchronous database operations

**Connection to Assignment:** As part of [A3 - Advanced App Creation + Demonstration](#), you will extend your NutriTrack app to build food logging functionality with persistent storage. The skills you learn today will allow you to implement:

- Store user data to Database storage (instead of Shared Preferences)
- Retrieval and display of saved entries
- Data management between app sessions
- Proper separation of concerns using industry-standard architecture

**Before You Begin:** Ensure Android Studio is updated and you have an emulator or physical device ready for testing. This lab builds progressively, with each section building on the previous one, so complete tasks in order.

# PART 1: UNDERSTANDING THE MVVM ARCHITECTURE

## Learning Objectives

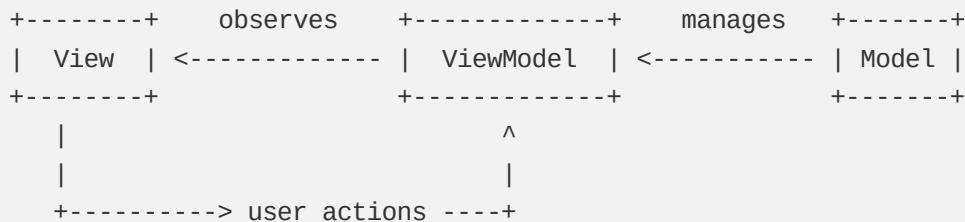
By the end of this section, you will be able to:

- Explain the purpose and benefits of the MVVM architecture
- Identify the key components in MVVM and their responsibilities
- Describe how data flows through an MVVM application
- Understand why this architecture is preferred for Android development

### 1.1 What is MVVM?

MVVM (Model-View-ViewModel) is an architectural pattern that separates an application into three main components:

- **Model:** Represents your data and business logic
- **View:** Displays the UI and observes the ViewModel
- **ViewModel:** Acts as a bridge between Model and View, preparing data for display



*MVVM Architecture Diagram*

### Benefits of MVVM:

1. **Separation of concerns:** Each component has a distinct responsibility
2. **Testability:** Logic in ViewModels can be tested independently of UI
3. **Maintainability:** Changes to one component don't require changes to others
4. **Lifecycle management:** ViewModels survive configuration changes (like screen rotation)

**Real-world Connection:** Major professional apps like Instagram, Netflix, and Google Maps use MVVM or similar architectures. These patterns allow large teams to work on different components simultaneously and make it easier to add features or fix bugs without breaking existing functionality.

# Why do we need a ViewModel?

Imagine your app screen is a *whiteboard* used by the user. When they rotate the screen, Android wipes the board clean and hands you a new whiteboard. Without a ViewModel, everything the user wrote is gone. The ViewModel gives you a *clipboard behind the scenes*—holding all the notes so even if the board gets wiped, the data survives.



Source: Generated using Dall-E

A `ViewModel` is part of the MVVM architecture and acts as a holder for all the data your UI needs to display. It's **lifecycle-aware**, so it stays alive across configuration changes like screen rotations. This keeps your UI snappy and your logic clean.

The ViewModel lifecycle in Jetpack Compose aligns with the lifecycle of the composable's host (either an Activity or a Fragment) that it is scoped to.

## Key Lifecycle Characteristics:

- **Creation:**
  - When you use `ViewModel()` within a composable, Jetpack Compose ensures that the ViewModel instance is created once and survives configuration changes (like screen rotations).
  - The ViewModel is typically scoped to the lifecycle of the Activity or Fragment, meaning it will persist as long as that host exists.

Example:

```
val myViewModel: MyViewModel = viewModel()
```

- **Survival Across Configuration Changes:**

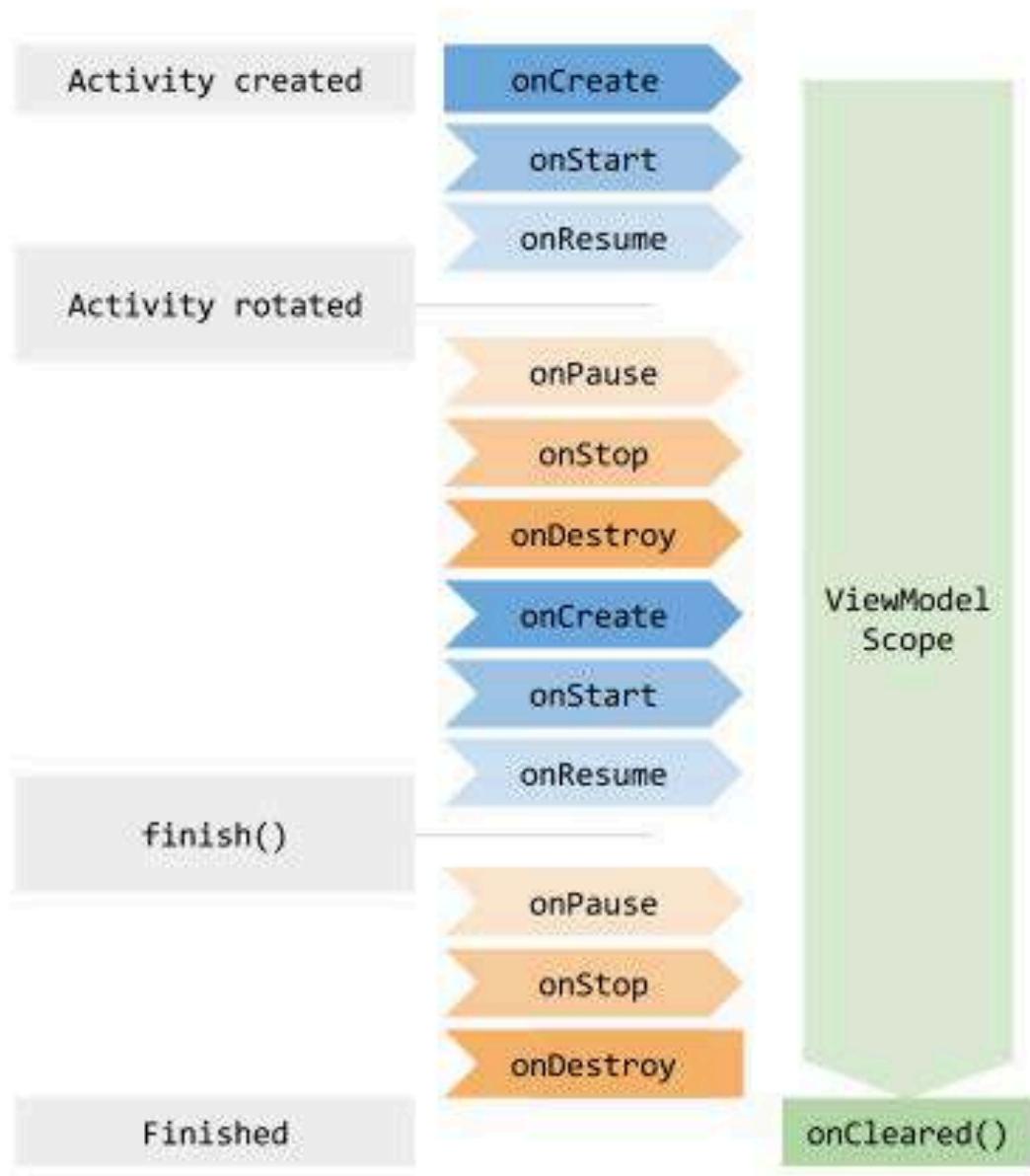
- If the Activity or Fragment is recreated (e.g., due to a configuration change like device rotation), the same ViewModel instance is returned by `viewModel()`. This allows you to retain data and avoid unnecessary recomputation or data loss.
- Cleanup (`onCleared`):
  - When the host's lifecycle ends permanently (e.g. when the user navigates away and the Activity is destroyed), the ViewModel is cleared, and its `onCleared()` method is called.
  - You can override `onCleared()` in your ViewModel to clean up resources, like cancelling coroutines or closing database connections.

```
override fun onCleared() {  
    super.onCleared()  
    // Cleanup logic here  
}
```

The lifecycle of a ViewModel is tied directly to its scope. A ViewModel remains in memory until the `ViewModelStoreOwner` to which it is scoped disappears. This may occur in the following contexts:

When an activity finishes, a fragment detaches, and a Navigation entry is removed from the back stack, ViewModels are a great solution for storing data that survives configuration changes.

The figure below illustrates the various lifecycle states of an activity as it undergoes a rotation and then is finished. The illustration also shows the lifetime of the ViewModel next to the associated activity lifecycle. This particular diagram illustrates the states of an activity. The same basic states apply to the lifecycle of a fragment.



## Summary and Key Takeaways

## References:

- <https://developer.android.com/topic/libraries/architecture/viewmodel>

# ViewModel: A Technical Breakdown

## What is ViewModel?

A ViewModel in Android is a key component in the Model-View-ViewModel (MVVM) architecture. Its primary role is to store and manage UI-related data for a specific screen or feature. The ViewModel class is designed to survive configuration changes like screen rotations, ensuring that the data persists and doesn't get lost, unlike activities or fragments that are frequently recreated.

## What's good about ViewModel

- **Lifecycle Awareness:** An Android ViewModel is a key component in the Model-View-ViewModel (MVVM) architecture. Its primary role is to store and manage UI-related data for a specific screen or feature. The ViewModel class is designed to survive configuration changes like screen rotations, ensuring that the data persists and doesn't get lost, unlike activities or fragments that are frequently recreated.
- **Persistence:** It retains data when configuration changes occur, reducing the need for reloads or recomputations.
- **Separation of Concerns:** This helps keep your UI code clean by separating business logic and data handling from the view (activity or fragment).
- **LiveData Compatibility:** ViewModel works hand-in-hand with LiveData, enabling reactive and observable data patterns to update the UI efficiently.

## Key Takeaway:

In Android, the ViewModel class is specifically designed to:

- Store and manage UI-related data
- Survive configuration changes
- Provide clean API for the View to observe data changes

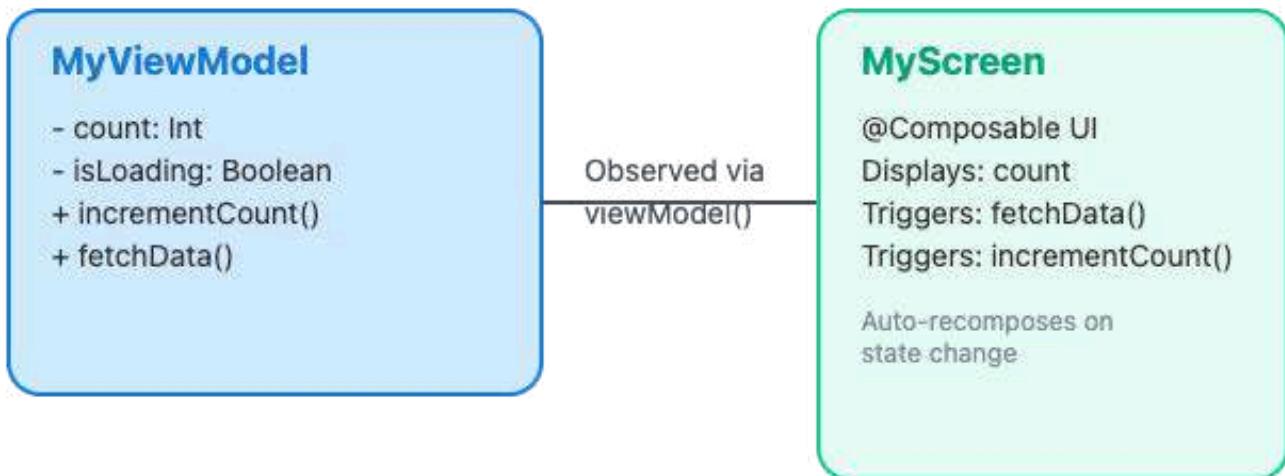
## Key Characteristics:

- Doesn't hold references to Views or Activities (prevents memory leaks)
- Typically contains LiveData or StateFlow objects that Views can observe
- Connects to repositories to access data sources
- Uses Kotlin coroutines for asynchronous operations

## Code Example

Until now, you've written each Screen composable directly in your Kotlin code. In this example, we will write a ViewModel() class to power our Screen so that the screen will survive any lifecycle changes (screen rotations etc.).

You don't need to write this code, just observe what is happening in the code below and understand the connection between ViewModel and the UI (composable).



First, let's create a ViewModel that holds some user data.

```
/**  
 * A ViewModel class responsible for managing a counter and simulating data fetching.  
 * This ViewModel tracks a number and whether we're "loading" something.  
 * You could imagine using this in a real app that fetches data from a network.  
 */  
class MyViewModel : ViewModel() {  
  
    /**  
     * Represents the current count.  
     * It's a mutable state, meaning changes to it will trigger UI updates.  
     * It's private set, so only the ViewModel can modify it.  
     */  
    var count by mutableStateOf(0)  
        private set // Make it private set to control updates from ViewModel  
  
    /**  
     * Indicates whether data is currently being loaded.  
     * It's a mutable state, so changes to it will trigger UI updates.  
     * It's private set, so only the ViewModel can modify it.  
     */
```

```

/*
var isLoading by mutableStateOf(false)
    private set

/**
 * Increments the [count] by 1.
 * This function is called to update the counter value.
 */
fun incrementCount() {
    count++
}

/**
 * Simulates fetching data asynchronously.
 * It sets [isLoading] to true, delays for 2 seconds (simulating a network request),
 * updates the [count], and then sets [isLoading] to false.
 *
 * This function uses [viewModelScope] to launch a coroutine, ensuring it's tied to the ViewMod
 */
fun fetchData() {
    viewModelScope.launch {
        isLoading = true // Indicate that data loading has started
        delay(2000) // Simulate a network request or long-running task
        count += 10 // Update count after data fetch
        isLoading = false // Indicate that data loading has finished
    }
}
}

```

## MyViewModel Class:

- Extends `ViewModel`.
- `count` and `isLoading` are declared as `var` properties using `mutableStateOf`. This makes them observable within Compose.
- `private set` is used to restrict modifications of `count` and `isLoading` to within the `ViewModel`. This enforces the principle that the `ViewModel` is the single source of truth for its data.
- `incrementCount()`: A simple function to increment the `count`.
- `fetchData()`:
  - Uses `viewModelScope.launch` to launch a coroutine. `viewModelScope` is tied to the `ViewModel`'s lifecycle, so the coroutine will be canceled when the `ViewModel` is cleared.
  - Sets `isLoading` to `true` to indicate that the data fetch is in progress.
  - Uses `delay(2000)` to simulate a 2-second network request.
  - Updates `count` after the simulated data fetch.
  - Sets `isLoading` to `false`.

```

/**
 * A Composable function that displays a counter and provides buttons to increment it or fetch data
 * It uses a [MyViewModel] to manage the counter's state and data fetching.
 *
 * @param myViewModel The [MyViewModel] instance to use. Defaults to a new instance provided by [viewModel].
 */
@Composable
fun MyScreen(myViewModel: MyViewModel = viewModel()) {
    Column(
        modifier = Modifier.fillMaxSize(), // Fill the entire available screen space
        verticalArrangement = Arrangement.Center, // Center the content vertically
        horizontalAlignment = Alignment.CenterHorizontally // Center the content horizontally
    ) {
        // Display the current count from the ViewModel.
        Text(text = "Count: ${myViewModel.count}")

        // Add vertical spacing between the Text and the first Button.
        Spacer(modifier = Modifier.height(16.dp))

        // Button to increment the counter.
        Button(onClick = { myViewModel.incrementCount() }) {
            Text(text = "Increment")
        }

        // Add vertical spacing between the Buttons.
        Spacer(modifier = Modifier.height(16.dp))

        // Button to simulate fetching data and update the counter.
        Button(onClick = { myViewModel.fetchData() }) {
            Text(text = "Fetch Data")
        }

        // Conditionally display a CircularProgressIndicator while data is loading.
        if (myViewModel.isLoading) {
            // Add vertical spacing before the progress indicator.
            Spacer(modifier = Modifier.height(16.dp))
            CircularProgressIndicator() // Display the loading indicator
        }
    }
}

```

- **MyScreen Composable:**

- `myViewModel: MyViewModel = viewModel()`: This line retrieves an instance of `MyViewModel` using the `viewModel()` composable function. This function ensures that the ViewModel is created or retrieved correctly and survives configuration changes.
- The `Column` composable arranges the UI elements vertically.
- `Text(text = "Count: ${myViewModel.count}")`: Displays the current count. Because `count` is a state, when it changes, this composable will recompose and show the new value.

count.

- `Button(onClick = { myViewModel.incrementCount() })`: A button that calls `incrementCount()` when clicked.
- `Button(onClick = { myViewModel.fetchData() })`: A button that calls `fetchData()`.
- `if (myViewModel.isLoading) { ... }`: Conditionally displays a `CircularProgressIndicator` while `isLoading` is `true`.

## Key Concepts:

- **ViewModel**: Holds UI-related data and business logic, surviving configuration changes.
- **mutableStateOf**: Creates observable state variables. When a state variable changes, Compose recomposes the composables that read it.
- **viewModel()**: A Compose function that provides a `ViewModel`.
- **viewModelScope**: A coroutine scope tied to the `ViewModel`'s lifecycle.
- **Recomposition**: Compose's mechanism for updating the UI when state changes.
- **State Hoisting**: Moving state up to a composable's caller, or in this case, a `viewModel`, for easier management.

# Checkpoint: ViewModel

Do you need a ViewModel for every screen?

Short Answer:

**No, you don't need a separate ViewModel for each screen**, but in most cases, it's **good practice** to have one **ViewModel per screen (or feature)** in Android app development, especially when following the **MVVM** (Model-View-ViewModel) pattern.

Why Use One ViewModel per Screen?

## 1. Separation of concerns

Keeps each screen's logic focused and independent. One screen = one source of state = cleaner, testable code.

## 2. Scoped lifecycle

ViewModels are lifecycle-aware. If two screens use the same ViewModel, you'll need to ensure their lifecycles align, or you risk memory leaks or stale data.

## 3. Better state management

Each ViewModel holds only the data relevant to its screen, making recomposition in Jetpack Compose predictable and efficient.

When to Share a ViewModel?

There *are* specific scenarios where sharing a ViewModel between screens makes sense:

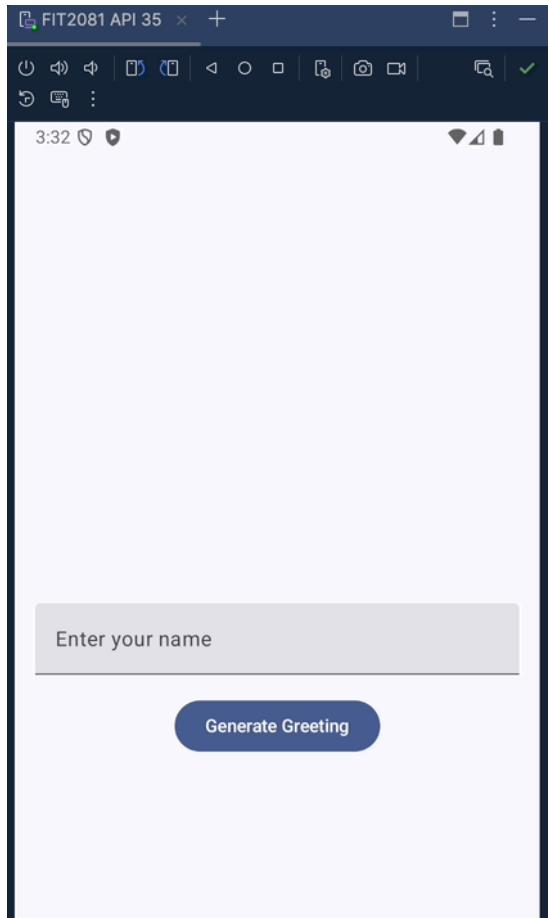
- **Shared UI state between closely related screens**, like:
  - Tabs inside the same parent `NavHost`
  - A multi-step form (e.g., registration wizard)
- **(Advanced)** Data that should survive across multiple composables in the same navigation graph

Rule of Thumb:

| “If a screen has independent UI logic or lifecycle, give it its own ViewModel.”

# Demo: Greeting App (ViewModel)

We will try to build the following application that uses ViewModel:



To demonstrate how the View Model works and to learn how to add it to your app, follow these steps.

**A.** Create a new project called week5\_viewmodel and add the following dependency:

Add the following dependency to the build.gradle file (just like last week):

```
implementation ("androidx.lifecycle:lifecycle-viewmodel-compose:2.6.2")
```

**B.** Add a new class to the root package of your project and name it GreetingViewModel.

```
1 import androidx.compose.runtime.getValue
2 import androidx.compose.runtime.mutableStateOf
3 import androidx.compose.runtime.setValue
4 import androidx.lifecycle.ViewModel
5 import androidx.lifecycle.viewModelScope
6 import kotlinx.coroutines.delay
7 import kotlinx.coroutines.launch
8 import java.util.Calendar
9
10 // GreetingViewModel: A ViewModel to manage UI state related to greeting messages.
11 class GreetingViewModel : ViewModel() {
12     // Stores the name of the user. Default is an empty string.
13     var userName by mutableStateOf("")
14         private set // Prevent external modification;
15         // can only be updated through ViewModel functions.
16
17     // Stores the greeting message to be displayed.
18     // Default is an empty string.
19     var greetingMessage by mutableStateOf("")
20         private set // Prevent external modification;
21         // can only be updated through ViewModel functions.
22
23     // Indicates whether a loading operation is in progress. Default is false.
24     var isLoading by mutableStateOf(false)
25         private set // Prevent external modification; controlled within ViewModel.
26
27     /**
28      * Updates the userName property.
29      * @param name - The new name provided by the user.
30      */
31     fun updateUserName(name: String) {
32         userName = name
33     }

```

codesnap.dev

After that, add the generateGreeting function below the updateUserName() function that returns a string based on the input string and the current time.

```

1  /**
2   * Generates a personalized greeting message
3   * based on the time of day and the user's name.
4   * The operation is asynchronous to simulate processing delay.
5   * If the userName is blank, an error message is displayed.
6   */
7  fun generateGreeting() {
8      //viewModelScope is a coroutine scope tied to the ViewModel's lifecycle.
9      //We need it to run the delay operation without blocking the UI.
10     viewModelScope.launch {
11         isLoading = true // Set loading state to true during the operation.
12         delay(1000) // Simulate a delay (e.g., fetching or processing data).
13
14         if (userName.isNotBlank()) { // Ensure userName is not empty.
15             // Get the current hour of the day (0-23).
16             val currentTime = Calendar.getInstance().get(Calendar.HOUR_OF_DAY)
17
18             // Determine greeting message based on the current time.
19             greetingMessage = when (currentTime) {
20                 in 0..11 -> "Good Morning, $userName!"
21                 in 12..17 -> "Good Afternoon, $userName!"
22                 else -> "Good Evening, $userName!"
23             }
24         } else {
25             // Display error message if no name is provided.
26             greetingMessage = "Please enter your name."
27         }
28         isLoading = false // Set loading state back
29         // to false after the operation completes.
30     }
31 }

```

codesnap.dev

Now, to call the generateGreeting function, we need to do only 2 steps in another Composable:

- Create an instance of the ViewModel

```
viewModel: GreetingViewModel = viewModel()
```

- Then, call the viewModel method inside the onClick callback code

```
viewModel.generateGreeting()
```

**C.** In your MainActivity.kt, setup your onCreate() method to call a GreetingScreen() composable

```

setContent {
    Week5_viewmodelTheme {
        Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
            GreetingScreen(innerPadding)
    }
}

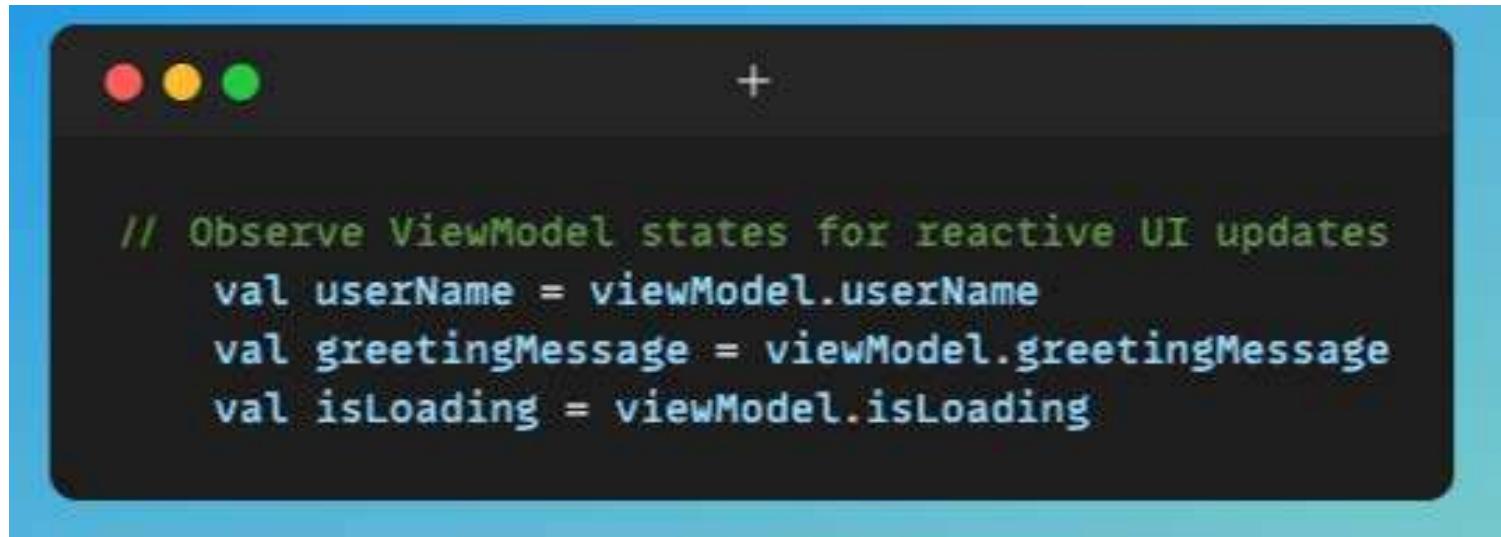
```

```
    }  
}  
}
```

D: Create the GreetingScreen method (or whatever your screen composable in MainActivity > onCreate is called) to accept a viewModel parameter:

```
/**  
 * Composable function that represents the main screen for displaying a greeting.  
 * It handles user input, greeting generation, and displays the result.  
 *  
 * @param innerPaddingValues Padding values provided by the Scaffold to handle screen insets.  
 * @param viewModel The ViewModel responsible for managing the greeting data and logic.  
 */  
@Composable  
fun GreetingScreen(  
    innerPaddingValues: PaddingValues,  
    viewModel: GreetingViewModel = viewModel()  
) {  
    // Rest of your code, BLAH BLAH..  
}
```

E: Write code for extracting the viewModel variables so that we can handle UI updates



F: Write UI Layout Code

Remember we are building a UI with the following elements (see screenshot above):

Column

----TextField

----Spacer

----Button

----Spacer

----CircularProgressIndicator()

```
// Main UI Layout: A Column for vertical arrangement of UI elements
Column(
    modifier = Modifier
        .fillMaxSize()
        .padding(innerPaddingValues)
        .padding(16.dp),
    horizontalAlignment = Alignment.CenterHorizontally,
    verticalArrangement = Arrangement.Center
) {
    // TextField for user to input their name
    TextField(
        value = userName,
        onValueChange = { viewModel.updateUserName(it) },
        label = { Text("Enter your name") },
        modifier = Modifier.fillMaxWidth()
    )
    // Spacers to add vertical space between components
    Spacer(modifier = Modifier.height(16.dp))

    // Button to trigger the generation of the greeting message
    Button(
        // Trigger the generation of the greeting message when clicked
        onClick = { viewModel.generateGreeting() },
        // Align the button to the center horizontally
        modifier = Modifier.align(Alignment.CenterHorizontally)
    ) {
        Text("Generate Greeting")
    }
    Spacer(modifier = Modifier.height(16.dp))
    // Conditional display: show loading indicator while loading, else show the greeting
    if (isLoading) {
        // Loading indicator
        CircularProgressIndicator()
    } else {
        Text(
            text = greetingMessage,
            fontSize = 20.sp,
            textAlign = TextAlign.Center
        )
    }
}
```

Notice how inside our button's onClick method, we are calling the generateGreeting() function we created:

```
// Button to trigger the generation of the greeting message
Button(
    // Trigger the generation of the greeting message when clicked
    onClick = { viewModel.generateGreeting() },
    // Align the button to the center horizontally
    modifier = Modifier.align(Alignment.CenterHorizontally)
) {
    Text("Generate Greeting")
```

```
}
```

## Next Steps

Right now, our ViewModel stores fake in-memory data. But in real apps, we'd pull real data from a database—coming up next with Room! Our ViewModel will become the bridge between the UI and Room.

Reference: Full Code for MainActivity.kt > GreetingScreen



```
/*
 * Composable function that represents the main screen for displaying a greeting.
 * It handles user input, greeting generation, and displays the result.
 *
 * @param innerPaddingValues Padding values provided by the Scaffold to handle screen insets.
 * @param viewModel The ViewModel responsible for managing the greeting data and logic.
 */
@Composable
fun GreetingScreen(
    innerPaddingValues: PaddingValues,
    viewModel: GreetingViewModel = viewModel()
) {
    // Observe ViewModel states for reactive UI updates
    val userName = viewModel.userName
    val greetingMessage = viewModel.greetingMessage
    val isLoading = viewModel.isLoading

    // Main UI Layout: A Column for vertical arrangement of UI elements
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(innerPaddingValues)
            .padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        // TextField for user to input their name
        TextField(
            value = userName,
            onValueChange = { viewModel.updateUserName(it) },
            label = { Text("Enter your name") },
            modifier = Modifier.fillMaxWidth()
        )
        // Spacer to add vertical space between components
        Spacer(modifier = Modifier.height(16.dp))

        // Button to trigger the generation of the greeting message
        Button(
            // Trigger the generation of the greeting message when clicked
            onClick = { viewModel.generateGreeting() },
            // Align the button to the center horizontally
            modifier = Modifier.align(Alignment.CenterHorizontally)
        ) {
            Text("Generate Greeting")
        }
    }
}
```

```
        }
        Spacer(modifier = Modifier.height(16.dp))
        // Conditional display: show loading indicator while loading, else show the greeting
        if (isLoading) {
            // Loading indicator
            CircularProgressIndicator()
        } else {
            Text(
                text = greetingMessage,
                fontSize = 20.sp,
                textAlign = TextAlign.Center
            )
        }
    }
}
```

Reference: Code for GreetingViewModel.kt

```
// GreetingViewModel: A ViewModel to manage UI state related to greeting messages.
class GreetingViewModel : ViewModel() {
    // Stores the name of the user. Default is an empty string.
    var userName by mutableStateOf("")
        private set // Prevent external modification;
                    // can only be updated through ViewModel functions.

    // Stores the greeting message to be displayed.
    // Default is an empty string.
    var greetingMessage by mutableStateOf("")
        private set // Prevent external modification;
                    // can only be updated through ViewModel functions.

    // Indicates whether a loading operation is in progress. Default is false.
    var isLoading by mutableStateOf(false)
        private set // Prevent external modification; controlled within ViewModel.

    /**
     * Updates the userName property.
     * @param name - The new name provided by the user.
     */
    fun updateUserName(name: String) {
        userName = name
    }

    /**
     * Generates a personalized greeting message
     * based on the time of day and the user's name.
     * The operation is asynchronous to simulate processing delay.
     * If the userName is blank, an error message is displayed.
     */
    fun generateGreeting() {
        //viewModelScope is a coroutine scope tied to the ViewModel's lifecycle.
        //We need it to run the delay operation without blocking the UI.
        viewModelScope.launch {
            isLoading = true // Set loading state to true during the operation.
            delay(1000) // Simulate a delay (e.g., fetching or processing data).

            if (userName.isNotBlank()) { // Ensure userName is not empty.
                // Get the current hour of the day (0-23).
                val currentTime = Calendar.getInstance().get(Calendar.HOUR_OF_DAY)

                // Determine greeting message based on the current time.
                greetingMessage = when (currentTime) {
                    in 0..11 -> "Good Morning, $userName!"
                    in 12..17 -> "Good Afternoon, $userName!"
                    else -> "Good Evening, $userName!"
                }
            } else {
                // Display error message if no name is provided.
                greetingMessage = "Please enter your name."
            }
            isLoading = false // Set loading state back
                            // to false after the operation completes.
        }
    }
}
```

## PART 2: Understanding Databases



**Note:** we will continue working on Part 2 in next week's lab. Try to complete as much as you can this week, but we will cover this again next week so that we can go deeper into understanding databases..

A **database** is an organized collection of structured data stored and accessed electronically. It allows for efficient retrieval, insertion, updating, and deletion of data. Databases are managed by **Database Management Systems (DBMS)**, which ensure data integrity, security, and performance.

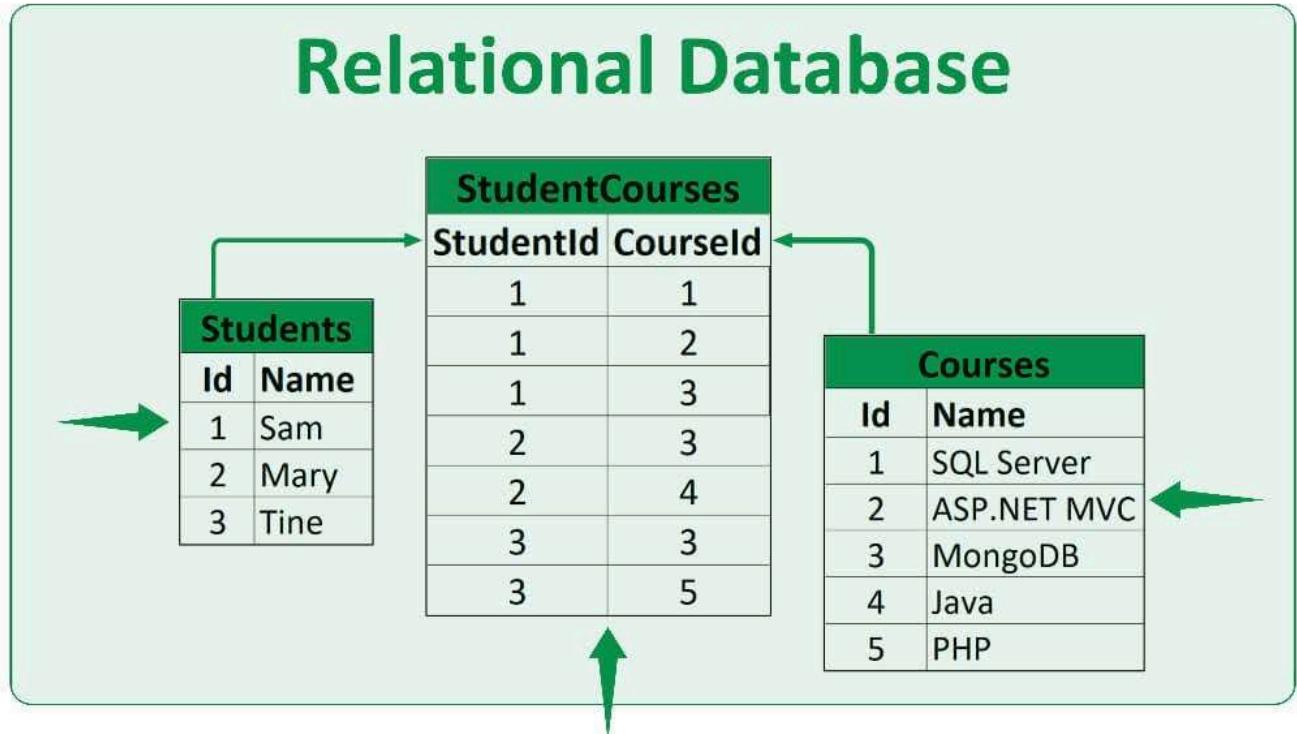
### Key Features of a Database:

- **Structured Storage** – Data is stored in tables (in relational databases) or other formats (like documents, graphs, etc.).
- **Easy Retrieval** – Allows querying data using languages like **SQL (Structured Query Language)**.
- **Data Integrity** – Ensures accuracy and consistency using constraints (e.g., primary keys, foreign keys).
- **Concurrency Control** – Manages multiple users accessing data simultaneously.
- **Security & Access Control** – Protects data with user permissions and encryption.
- **Scalability** – Handles growing amounts of data efficiently.

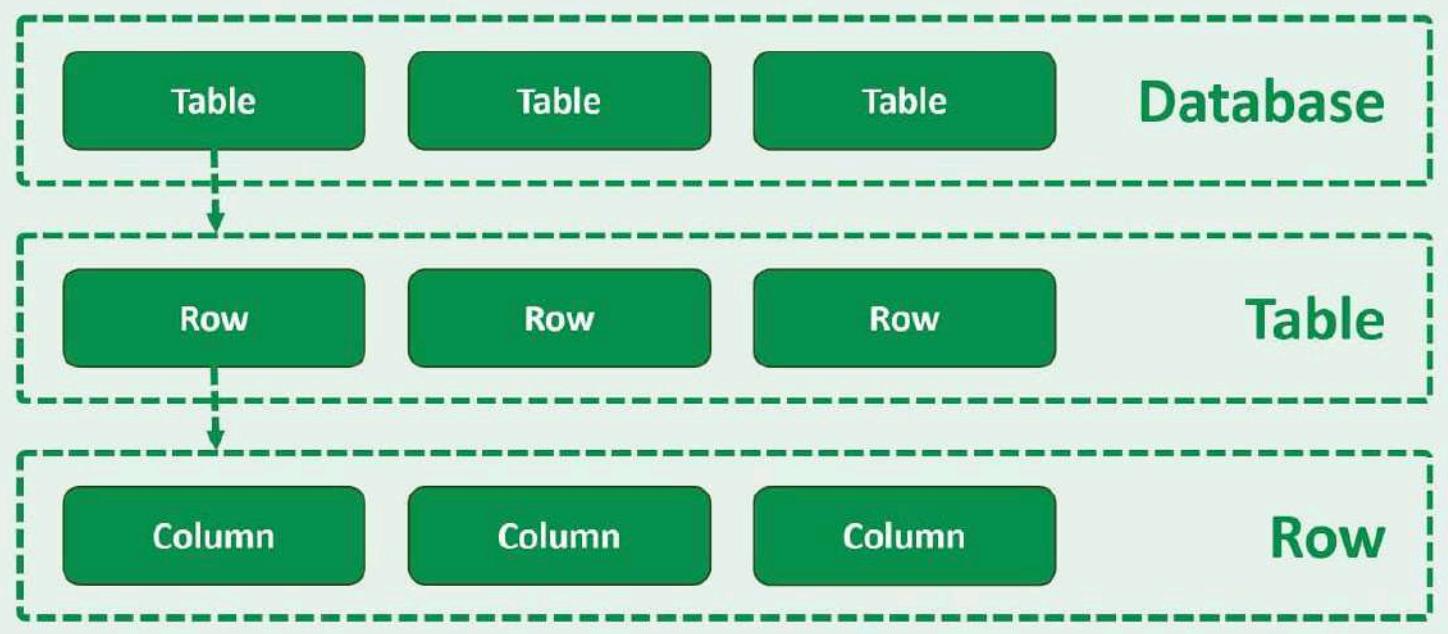
### Types of Databases:

- **Relational Databases (RDBMS)**
  - Stores data in tables with rows and columns.
  - Examples: **MySQL, PostgreSQL, Oracle, SQL Server**.
  - Uses **SQL** for querying.

# Relational Database



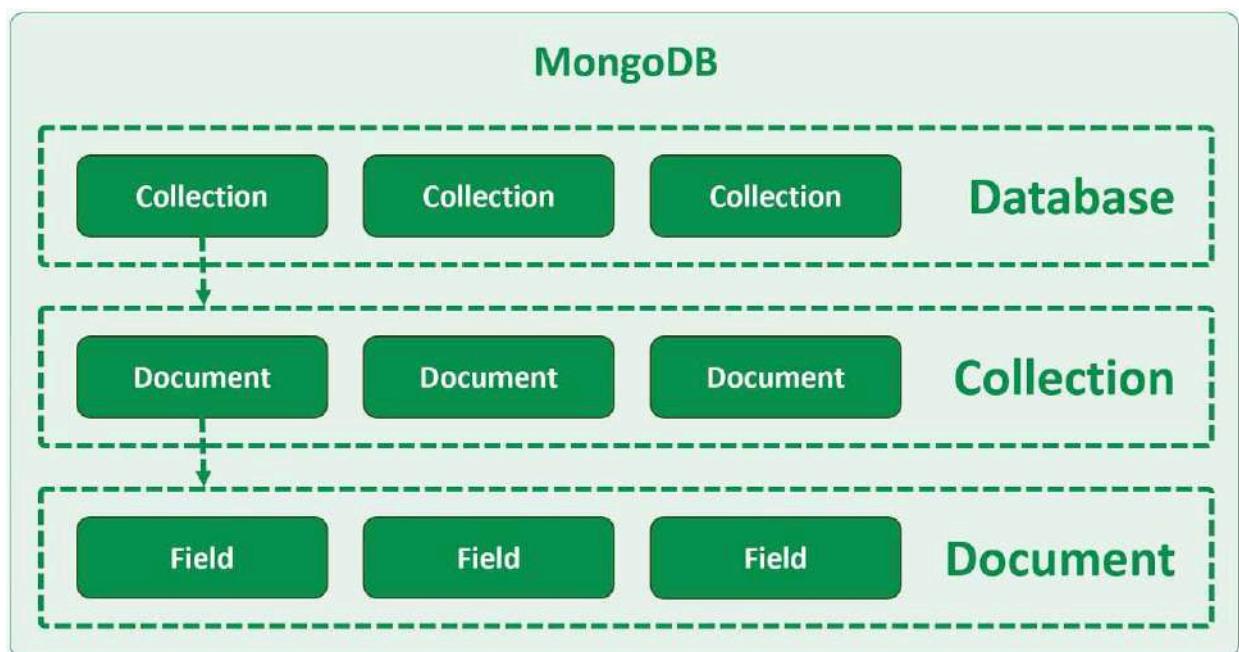
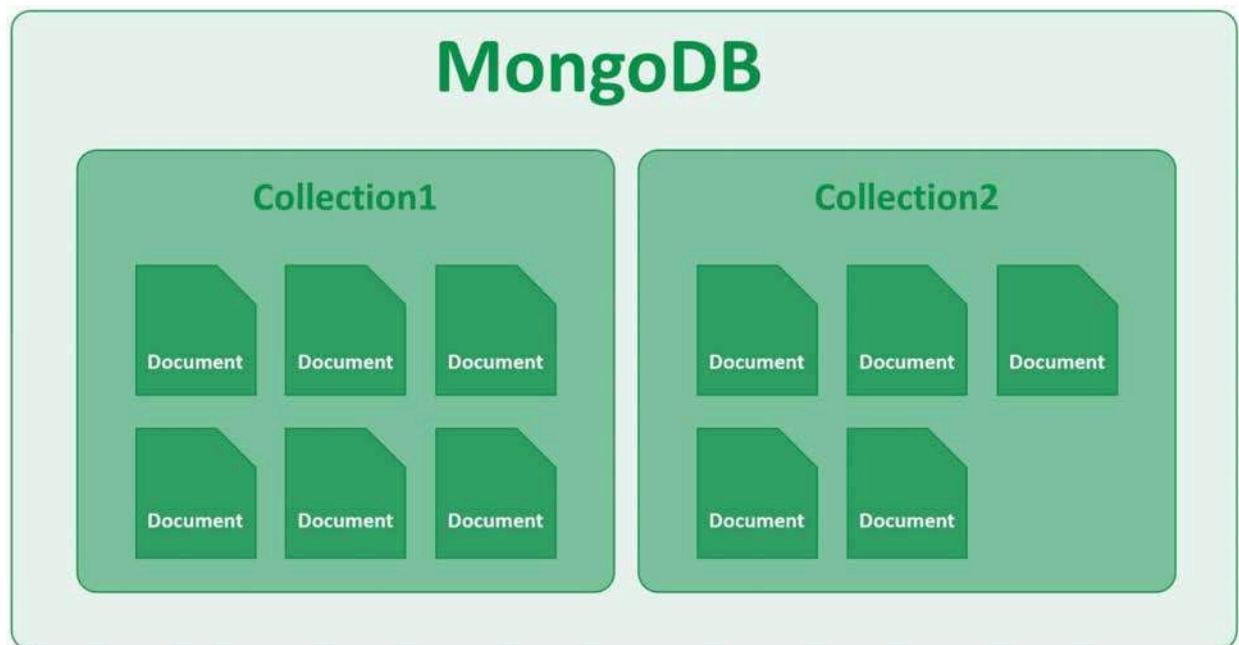
## Relational Database



- **NoSQL Databases**

- Handles unstructured or semi-structured data.
- Types:
  - **Document-based** (MongoDB, CouchDB)
  - **Key-Value** (Redis, DynamoDB)
  - **Column-family** (Cassandra, HBase)

- **Graph-based** (Neo4j)



- **In-Memory Databases**
  - Stores data in RAM for faster access (e.g., **Redis**).
- **Time-Series Databases**
  - Optimized for time-stamped data (e.g., **InfluxDB**).
- **Distributed Databases**
  - Spreads data across multiple servers (e.g., **Cassandra**, **Google Spanner**).

## Why Use a Database?

- **Efficient Data Management** – Avoids redundancy and inconsistency.
- **Fast Access** – Indexing improves search speed.

- **Multi-user Support** – Handles concurrent access.
- **Backup & Recovery** – Prevents data loss.
- **Supports Business Applications** – Used in banking, e-commerce, healthcare, etc.

#### References:

- <https://www.pragimtech.com/blog/mongodb-tutorial/relational-and-non-relational-databases/>

# What is Room Database?

**Room Database** is a persistence library part of the Android Jetpack Architecture Components. It acts as an abstraction layer over SQLite, simplifying database operations in Android apps while leveraging the full power of SQLite. Room streamlines tasks like defining schemas, writing queries, and managing database connections, while adding compile-time verification of SQL queries to prevent runtime errors.

## Room Database Key Components

The Room library is structured around three main components that work together to simplify database operations. Here's a breakdown of those key components:

- **Entities:**

- These are classes that represent tables within your database.
- Each instance of an entity class corresponds to a row in the table.
- Fields within the entity class define the columns of the table.
- Entities are annotated with `@Entity` to signify their role.

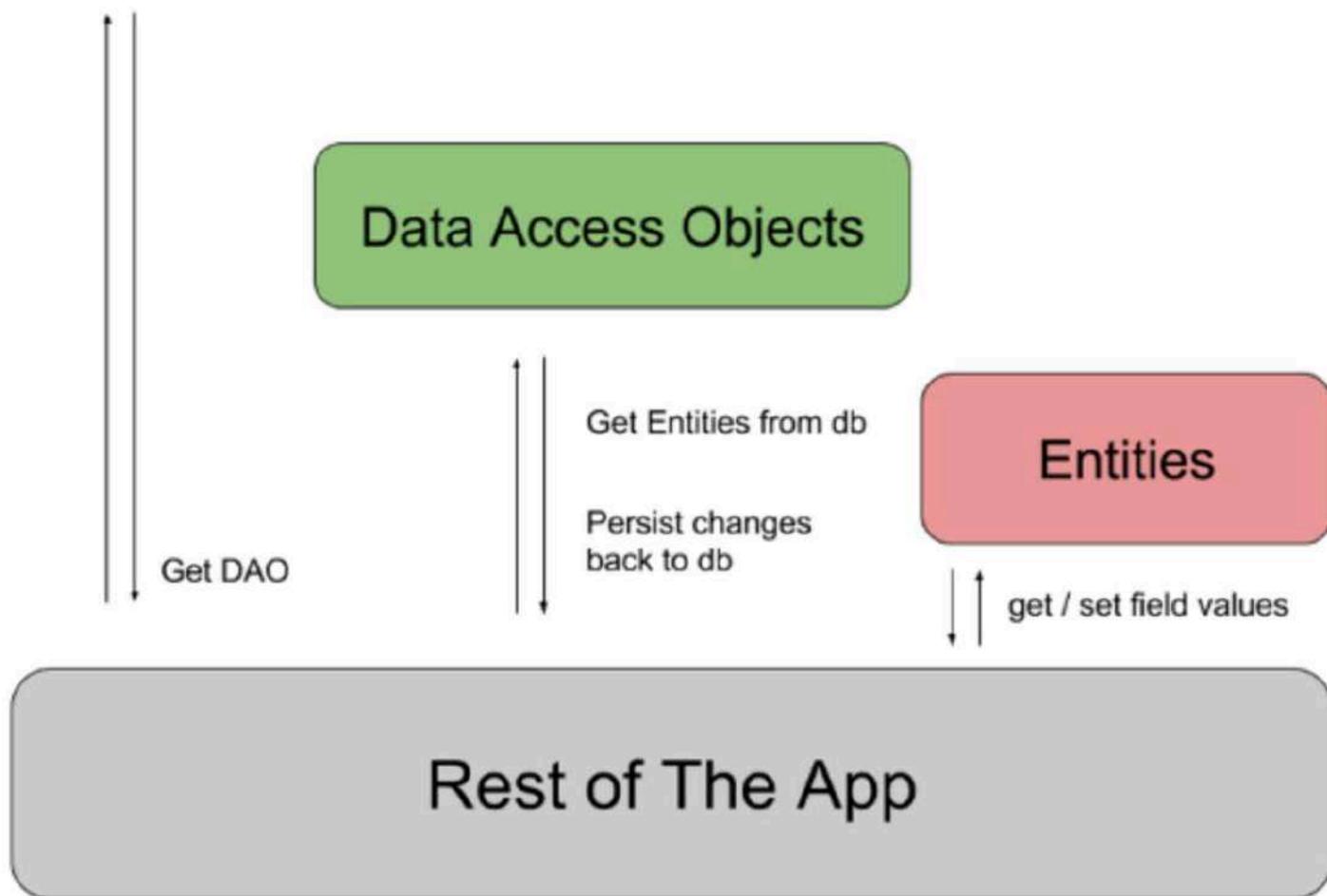
- **DAOs (Data Access Objects):**

- DAOs are interfaces or abstract classes that define your app's methods to interact with the database.
- They contain methods for querying, inserting, updating, and deleting data.
- Room generates the necessary SQL code for methods like `@Query`, `@Insert`, `@Update`, and `@Delete` using annotations.
- DAOs abstract the underlying SQL queries, making your code cleaner and more maintainable.

- **Database:**

- This abstract class provides the main access point to the underlying database connection.
- It holds the database and serves as your app's central point for interacting with the persisted data.
- The database class is annotated with `@Database` and includes a list of entities that define the database tables.
- It also contains abstract methods that return instances of the DAOs.
- This class is used to create the database instance.

# Room Database



## What is a Repository in Room Database?

Repository is not part of the Room Database itself, but a design pattern used to abstract and centralize data operations. It mediates between different data sources (e.g., Room database, network APIs, caches) and your app's ViewModel or UI layer. When working with Room, repositories simplify interactions with the database and help manage data flow in a clean, scalable way.

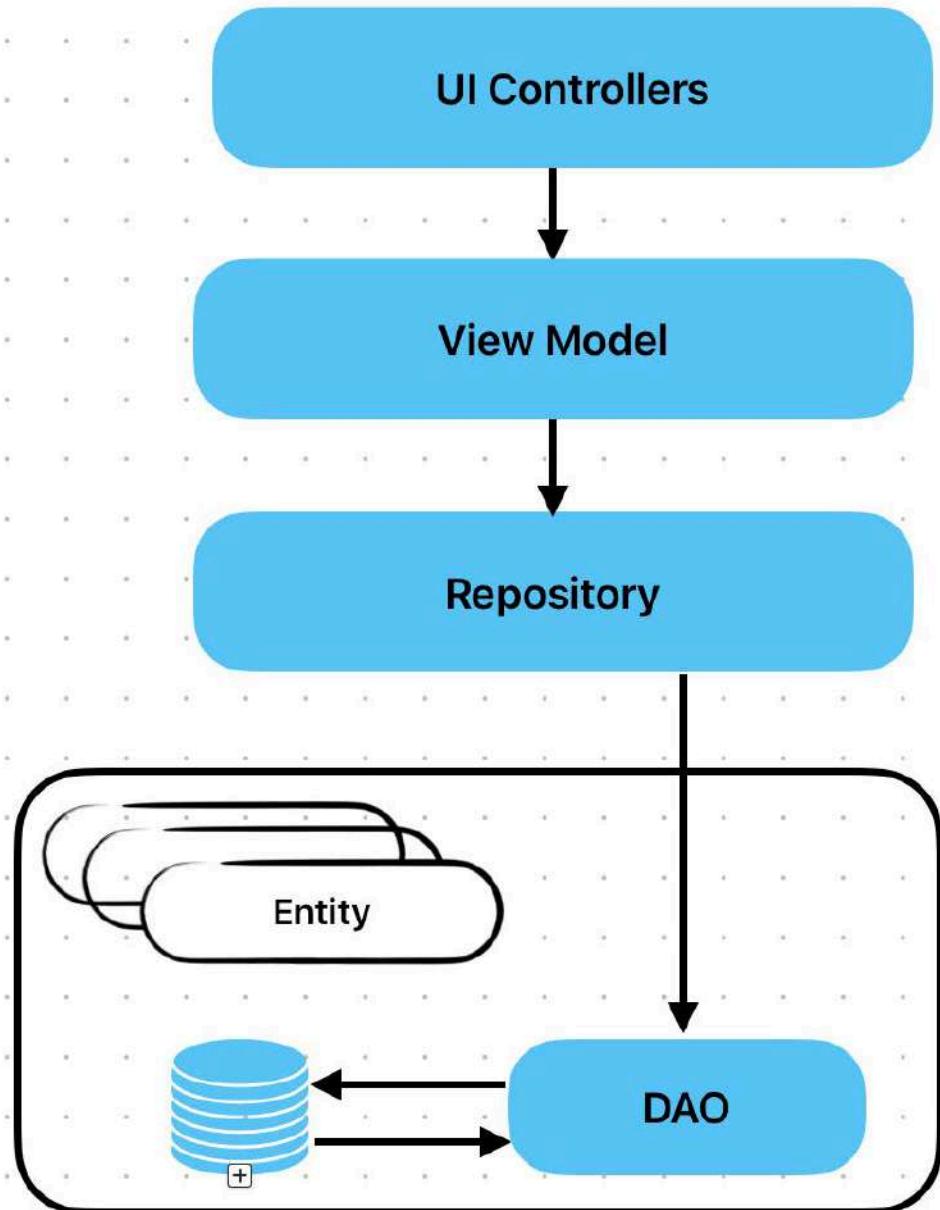
## Role of a Repository with Room Database

- **Single Source of Truth:**
  - The repository decides whether to fetch data from the Room database (local cache), a remote API, or another source.

- Example: Fetch data from the network, cache it in Room, and return the cached data for future requests.
- **Hide Data Complexity:**
  - The ViewModel doesn't need to know if data comes from Room, an API, or a file. The repository handles the logic.
- **Centralized Data Operations:**
  - Combines Room DAO methods, API calls, and transformations into cohesive operations.
- **Thread Management:**
  - Ensures Room database operations (which must run off the UI thread) are executed using coroutines, RxJava, or `ExecutorService`.

## How Repositories Work with Room

- **Repository ↔ DAO ↔ Room Database**
- The repository uses the DAO (Data Access Object) to interact with the Room database.
- It may also coordinate with other data sources (e.g., Retrofit for APIs).

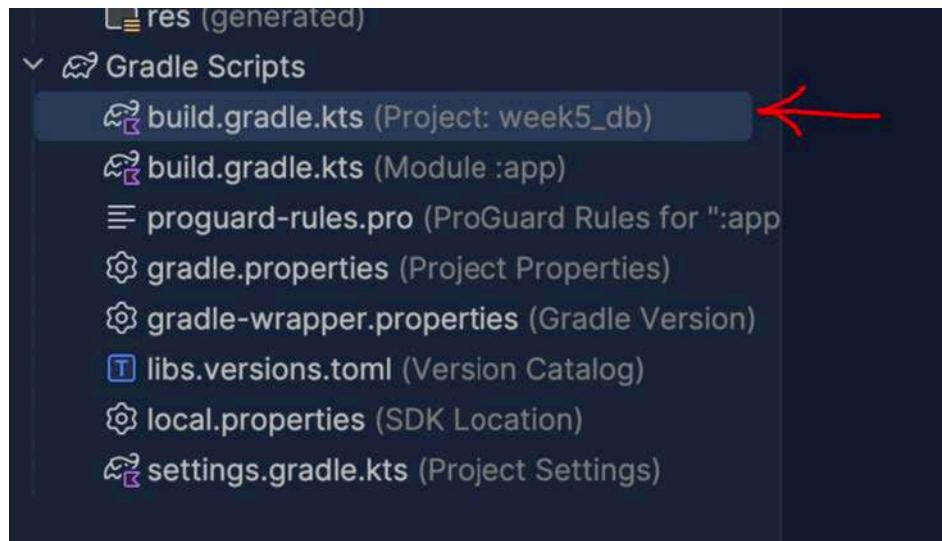


# Room DB Dependencies

For your app to have a Room database, several dependencies and plugins must be added.

- Open build.gradle.kts (Project Level) and add the following line to the list of plugins.

```
id("com.google.devtools.ksp") version "2.0.21-1.0.27" apply false
```



- Open build.gradle.kts (Module Level), and add the following line to the list of plugins

```
id("com.google.devtools.ksp")
```

## Gradle Scripts

build.gradle.kts (Project: week5\_db)

build.gradle.kts (Module :app) 

proguard-rules.pro (ProGuard Rules for ":app")

gradle.properties (Project Properties)

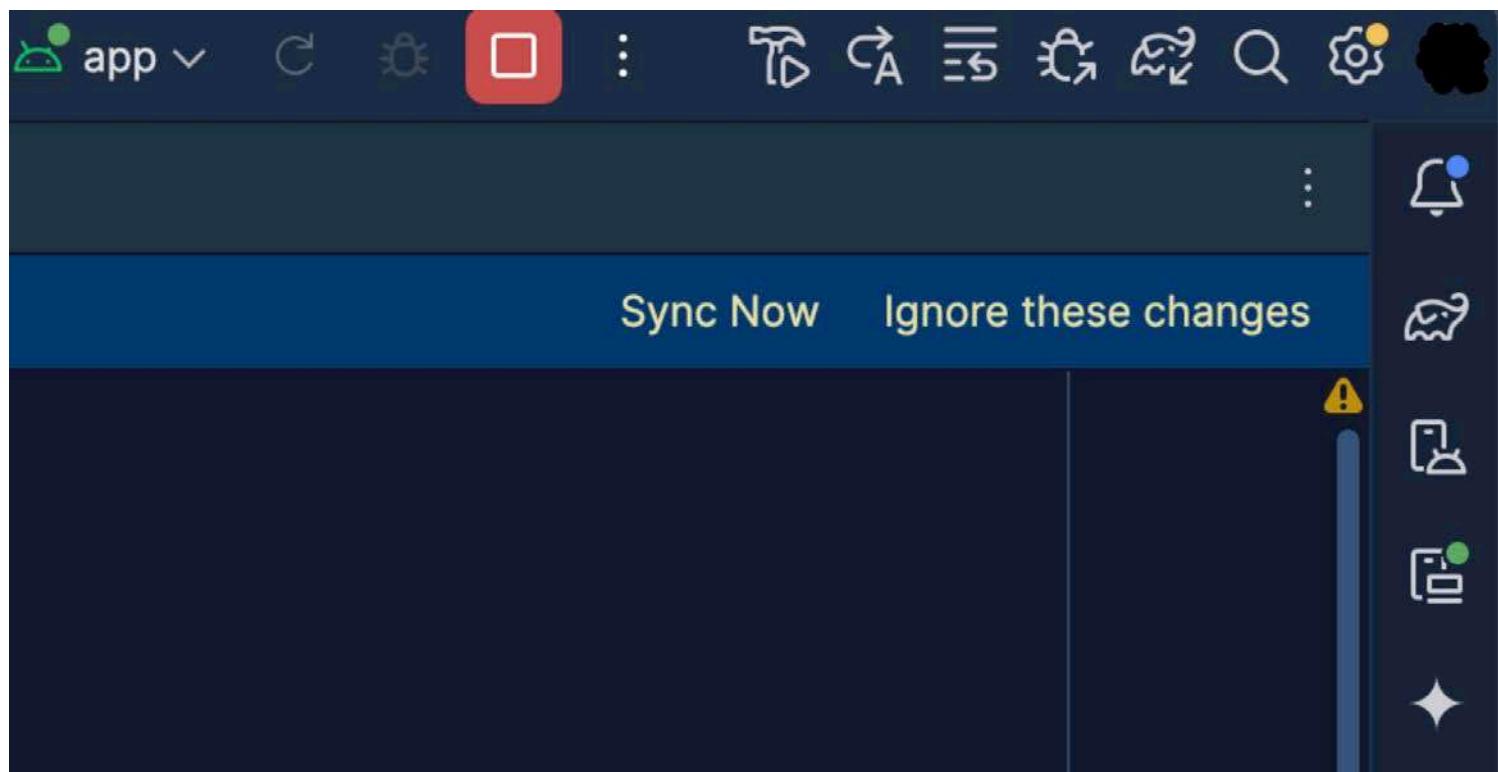
gradle-wrapper.properties (Gradle Version)

libs.versions.toml (Version Catalog)

local.properties (SDK Location)

settings.gradle.kts (Project Settings)

- Sync your gradle files by clicking on "Sync Now"



- In the same file, add the following to the dependencies:

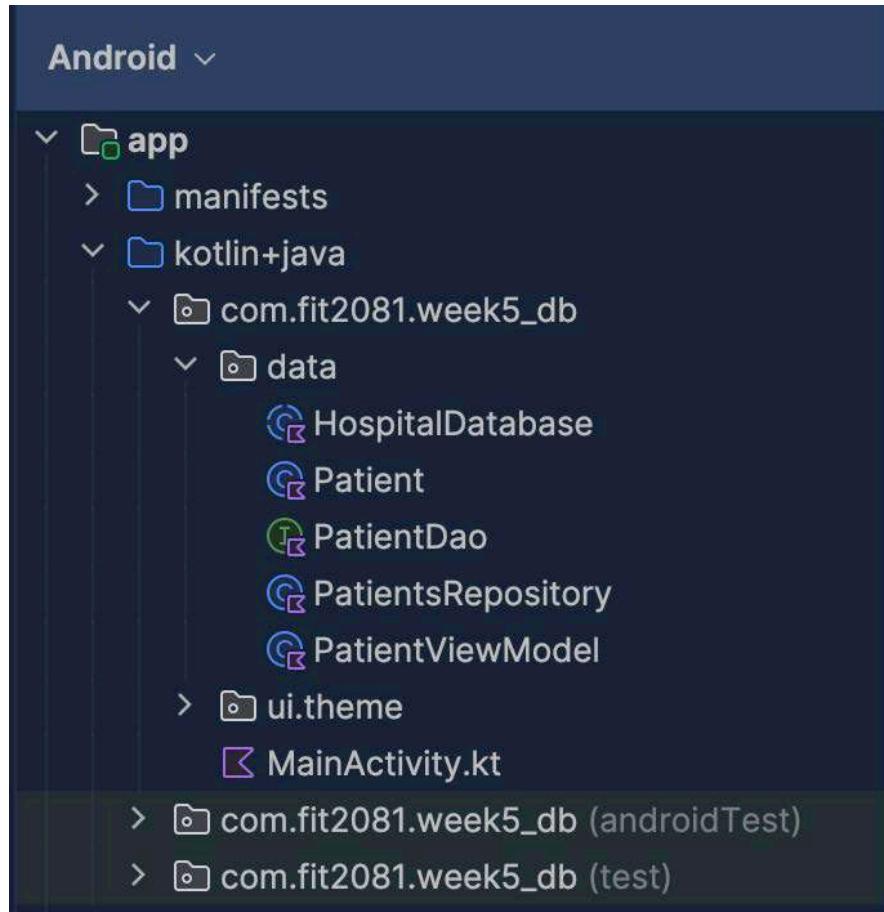
```
val room_version = "2.6.1"
implementation("androidx.room:room-runtime:$room_version")
```

```
// If this project uses any Kotlin source, use Kotlin Symbol Processing (KSP)
// See Add the KSP plugin to your project
ksp("androidx.room:room-compiler:$room_version")
// If this project only uses Java source, use the Java annotationProcessor
// No additional plugins are necessary
annotationProcessor("androidx.room:room-compiler:$room_version")
// optional - Kotlin Extensions and Coroutines support for Room
implementation("androidx.room:room-ktx:$room_version")
implementation ("androidx.lifecycle:lifecycle-viewmodel-compose:$room_version")
```

- Sync your Gradle files again

# Room DB Project Structure

A typical project structure for an Android app using Room database generally follows a clean architecture pattern, separating concerns into distinct packages or modules. Here is an example:

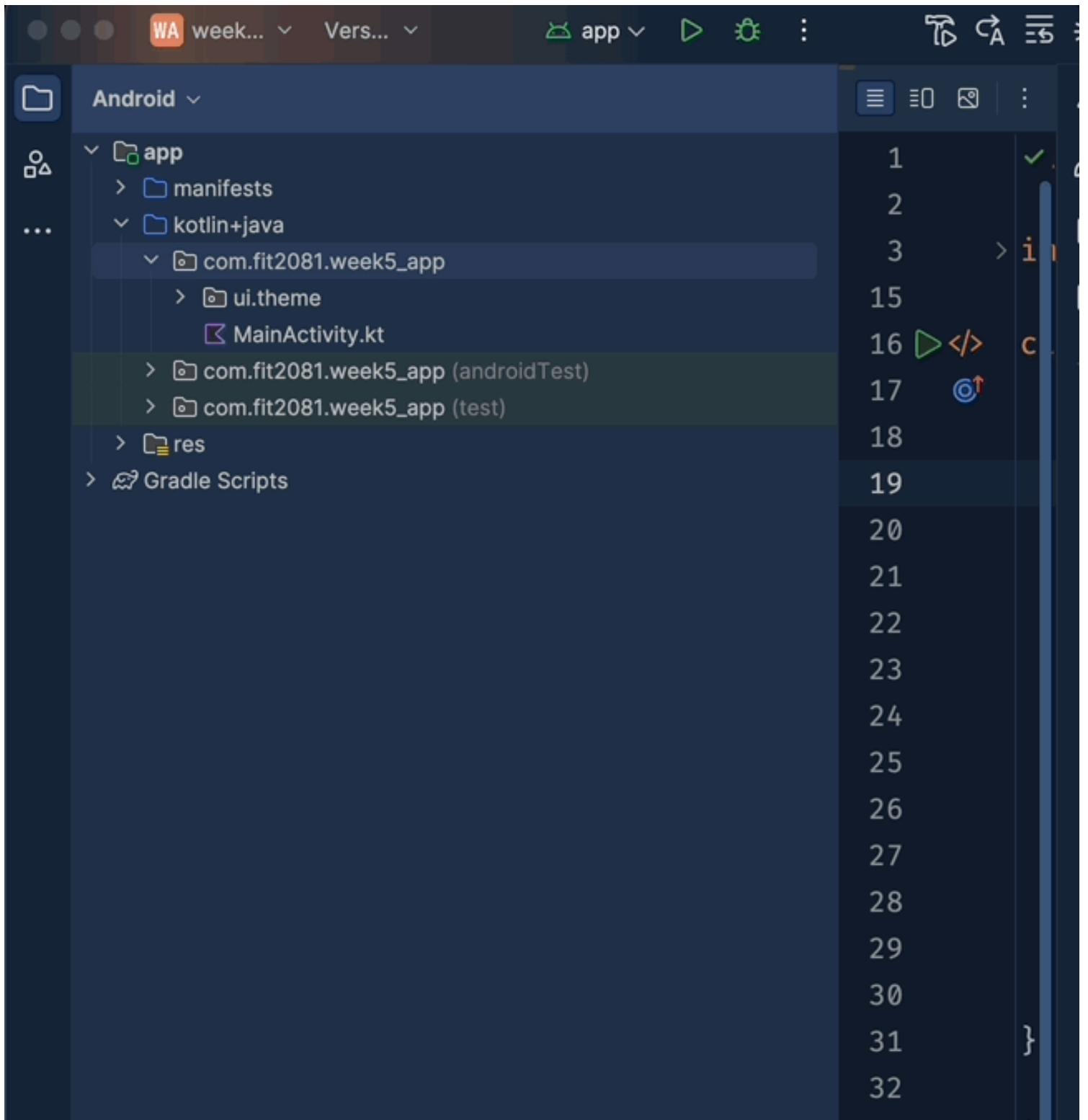


As you can see in the project structure above, we placed all the files that are related to the Room database in a separate package called 'data'. This helps to improve code maintainability, readability, scalability, and testability. If your database has more files and details, you might create sub-packages to categorise related files further. For instance, a sub-package called 'database' for the database and dao files, another sub-package called 'repositories' for all the repositories files, and a 'models' sub-package for all entities. We will follow the design above for the sake of simplicity.

## How to create a package?

- Right-click on the directory where you want to create the package.
- Select New > Package.
- Enter the name of your package (e.g., com.yourpackage.data.database).
- Press Enter.
- Android Studio will create the package (which is essentially a folder) for you.

See below for instructions on how to add a new package and add a class to it within your Android Studio project.



# Room DB Components

## Entity

Entity in Room database is a Kotlin data class representing a table in your SQLite database. It's a key component for defining your database schema and interacting with your data.

Annotations:

- @Entity Annotation:
  - This annotation from the Room library is used to mark a Kotlin class as a database entity.
  - You can specify the table name using the tableName parameter within the @Entity annotation.
- @PrimaryKey Annotation:
  - The @PrimaryKey annotation is used to designate the primary key column(s) of the table.
  - You can use the autoGenerate parameter to have Room automatically generate primary key values.



See this article for the full list of annotation you can use with Entity classes:

<https://developer.android.com/training/data-storage/room/defining-data>

```
1 import androidx.room.Entity
2 import androidx.room.PrimaryKey
3
4 /**
5  * Represents a patient entity in the database.
6  *
7  * This data class is annotated with `@Entity` to
8  * indicate that it represents a table in the database.
9  * The `tableName` property specifies the name
10 * of the table as "patient".
11 */
12 @Entity(tableName = "patients")
13 data class Patient(
14     /**
15      * The unique identifier for the patient.
16      * It is automatically generated by the database.
17      */
18     @PrimaryKey(autoGenerate = true)
19     val id: Int = 0,
20     /**
21      * The name of the patient.
22      */
23     val name: String,
24     /** The age of the patient */
25     val age: Int,
26     /** The address of the patient */
27     val address: String
28 )
```

codesnap.dev

In the code above, the class `Patient` serves as a data model for storing patient information within the Room Persistence Library in Android. Marked with the `@Entity` annotation, it represents a table named "patients". Each instance stores a unique patient's details: an auto-generated id (primary key), name, age, and address. Room utilizes this class to create and manage a corresponding database table, allowing for seamless data persistence and retrieval. The primary key ensures each patient record is distinct, and the auto-generation simplifies the process of adding new patients. Attributes define the structure and types of the table's data columns.

# Database Class

The **database class** in Room is an abstract class that extends RoomDatabase. It serves as the main access point to your persisted data, managing the underlying SQLite database and providing access to your Data Access Objects (DAOs).

## Key Roles:

- **Extends RoomDatabase:**
  - The database class must extend the RoomDatabase abstract class.
- **Abstract Class:**
  - It's an abstract class, meaning you cannot directly instantiate it. Room will generate the implementation for you.
- **@Database Annotation:**
  - It's annotated with @Database, which specifies the entities (tables) in the database and the database version.
- **DAOs as Abstract Methods:**
  - It defines abstract methods that return instances of your DAOs. These DAOs provide the methods for interacting with your database tables.
- **Database Instance Management:**
  - It's typically implemented as a singleton to ensure that only one instance of the database is created.

```
1 import android.content.Context
2 import androidx.room.Database
3 import androidx.room.Room
4 import androidx.room.RoomDatabase
5
6 /**
7  * This is the database class for the application. It is a Room database.
8  * It contains one entity: [Patient].
9  * The version is 1 and exportSchema is false.
10 */
11 @Database(entities = [Patient::class], version = 1, exportSchema = false)
12 // this is a room database
13 abstract class HospitalDatabase: RoomDatabase() {
14
15     /**
16      * Returns the [PatientDao] object.
17      * This is an abstract function that is implemented by Room.
18      */
19     abstract fun patientDao(): PatientDao
20
21     companion object {
22         /**
23          * This is a volatile variable that holds the database instance.
24          * It is volatile so that it is immediately visible to all threads.
25          */
26         @Volatile
27         private var Instance: HospitalDatabase? = null
28
29         /**
30          * Returns the database instance.
31          * If the instance is null, it creates a new database instance.
32          * @param context The context of the application.
33          */
34         fun getDatabase(context: Context): HospitalDatabase {
35             // if the Instance is not null, return it, otherwise create a new database instance.
36             //synchronized means that only one thread can access this code at a time.
37             //hospital_database is the name of the database.
38             return Instance ?: synchronized(this) {
39                 Room.databaseBuilder(context, HospitalDatabase::class.java, "hospital_database")
40                     .build().also { Instance = it }
41             }
42         }
43     }
44 }
```

codesnap.dev

The HospitalDatabase class serves as the main access point for the application's persisted data, leveraging the Room persistence library. This class is abstract, and Room will generate an implementation at compile time. It's defined as a singleton, ensuring that only one database instance exists across the application, which is crucial for efficiency and data consistency. It includes a single entity, Patient, representing patient data stored in the database. The database version is set to 1, and schema export is disabled. The class also provides a way to access PatientDao to perform database operations on the patient entity.

# Data Access Object

A DAO (Data Access Object) is an interface or abstract class that defines the methods used to access your database. It acts as an abstraction layer over the SQLite database, allowing you to interact with your data using Kotlin functions instead of raw SQL queries.

## Annotations:

Room uses annotations to define the SQL queries and database operations.

- @Dao annotation marks the interface or abstract class as a DAO.
- @Insert, @Update, @Delete, @Query and other annotations define database operations.

```
1 import androidx.room.Dao
2 import androidx.room.Delete
3 import androidx.room.Insert
4 import androidx.room.Query
5 import androidx.room.Update
6 import kotlinx.coroutines.flow.Flow
7
8 // This interface defines the data access object (DAO) for the Patient entity.
9 @Dao
10 interface PatientDao {
11     // Inserts a new patient into the database.
12     //suspend is a coroutine function that can be paused and resumed at a later time.
13     //suspend is used to indicate that the function will be called from a coroutine.
14     @Insert
15     suspend fun insert(patient: Patient)
16
17     // Updates an existing patient in the database.
18     @Update
19     suspend fun update(patient: Patient)
20
21     // Deletes a specific patient from the database.
22     @Delete
23     suspend fun delete(patient: Patient)
24
25     // Deletes all patients from the database.
26     @Query("DELETE FROM patients")
27     suspend fun deleteAllPatients()
28
29     // Retrieves all patients from the database, ordered by their ID in ascending order.
30     //The return type is a Flow, which is a data stream that can be observed for changes.
31     @Query("SELECT * FROM patients ORDER BY id ASC")
32     fun getAllPatients(): Flow<List<Patient>>
33 }
```

The PatientDao is a crucial component of the Room persistence library, acting as an interface that provides methods for accessing and manipulating patient data within the database. It is annotated with @Dao, indicating it's a Data Access Object. The PatientDao defines several abstract methods,

which Room will implement at compile time. These methods are mapped to SQL queries or database operations through annotations. `insert`, `update`, `delete` and `deleteAllPatients` are annotated with `@Insert`, `@Update`, `@Delete` and `@Query` respectively, representing the corresponding database operations. These functions can directly interact with the database to manage patient data, `getAllPatients` is another important function, that returns a `Flow<List>`, which is a stream of patient lists that automatically emits new lists whenever the underlying data changes, making it ideal for reactive UIs.

- i In Kotlin, the `suspend` keyword is a crucial part of the coroutines feature, designed to handle asynchronous operations in a more readable and efficient way. The primary purpose of `suspend` is to enable functions to perform long-running operations (like network requests or database queries) without blocking the main thread. This prevents your app from freezing and ensures a smooth user experience. For more info, click [HERE](#).
- i `Flow` is a type that represents an asynchronous stream of data. It's part of the Kotlin Coroutines library and is designed to handle sequences of values that are produced over time. It allows you to work with data that's produced asynchronously, meaning it doesn't block the main thread. This is crucial for tasks like network requests, database operations, and UI updates. For more info, click [HERE](#).

# Repository and ViewModel

## Repository

The primary roles of a repository class are:

- The Repository acts as a mediator, hiding the complexities of how data is obtained. This means that the parts of your application that need data (like ViewModels) don't need to know whether the data comes from a local database (Room), a network API, or any other source.
- This abstraction allows for greater flexibility and maintainability. If you need to change your data sources, you can do so within the Repository without affecting other app parts.
- The Repository handles data operations, such as fetching, inserting, updating, and deleting data.
- It can also implement logic for deciding which data source to use. For example, if unavailable locally, it might check the local database first and then fetch data from a network.

```
1 import android.content.Context
2 import kotlinx.coroutines.flow.Flow
3
4 class PatientsRepository {
5     // Property to hold the PatientDao instance.
6     var patientDao: PatientDao
7     // Constructor to initialize the PatientDao.
8     constructor(context: Context) {
9         // Get the PatientDao instance from the HospitalDatabase.
10        patientDao = HospitalDatabase.getDatabase(context).patientDao()
11    }
12    // Function to insert a patient into the database.
13    suspend fun insert(patient: Patient) {
14        // Call the insert function from the PatientDao.
15        patientDao.insert(patient)
16    }
17    // Function to delete a patient from the database.
18    suspend fun delete(patient: Patient) {
19        // Call the delete function from the PatientDao.
20        patientDao.delete(patient)
21    }
22    // Function to update a patient in the database.
23    suspend fun update(patient: Patient) {
24        // Call the update function from the PatientDao.
25        patientDao.update(patient)
26    }
27    // Function to delete all patients from the database.
28    suspend fun deleteAllPatients() {
29        // Call the deleteAllPatients function from the PatientDao.
30        patientDao.deleteAllPatients()
31    }
32    // Function to get all patients from the database as a Flow.
33    fun getAllPatients(): Flow<List<Patient>> = patientDao.getAllPatients()
34 }
```

The PatientsRepository class is an intermediary between the application's components (like ViewModels) and the database access object (PatientDao). It encapsulates the data access logic for patient-related operations. When instantiated, it receives a Context to retrieve a database instance from HospitalDatabase. Through this database instance, it obtains a PatientsRepository object, which contains the methods for direct database interaction. HospitalDatabase exposes several methods like insert, delete, update, and deleteAllPatients to perform CRUD (Create, Read, Update, Delete) operations on the patient data. These methods simply delegate the call to the underlying PatientDao. In addition, getAllPatients method retrieves all patients from the database as a Flow, which is a stream of data changes. This class centralizes patient data management, abstracting away the complexity of direct database interactions.

## ViewModel

The ViewModel separates the UI from the data layer (Room database). The UI is responsible for displaying data, while the ViewModel handles data retrieval and preparation. The ViewModel survives configuration changes (like screen rotations). It prevents data loss and unnecessary database queries, ensuring a smooth user experience. In other words, the UI components are destroyed and recreated, but the ViewModel persists.

```
1 import android.content.Context
2 import androidx.lifecycle.ViewModel
3 import androidx.lifecycle.ViewModelProvider
4 import androidx.lifecycle.viewModelScope
5 import kotlinx.coroutines.flow.Flow
6 import kotlinx.coroutines.launch
7
8 class PatientViewModel(context: Context) : ViewModel() {
9
10
11    //creates a repo object that will be used to interact with the database
12    private val patientRepo = PatientsRepository(context)
13    //gets all the patients stored in the repo
14    val allPatients: Flow<List<Patient>> = patientRepo.getAllPatients()
15
16    //inserts a patient into the repo
17    fun insert(patient: Patient) = viewModelScope.launch {
18        patientRepo.insert(patient)
19    }
20    //deletes a patient from the repo
21    fun delete(patient: Patient) = viewModelScope.launch {
22        patientRepo.delete(patient)
23    }
24    //updates a patient in the repo
25    fun update(patient: Patient) = viewModelScope.launch {
26        patientRepo.update(patient)
27    }
28    //deletes all the patients in the repo
29    fun deleteAllPatients() = viewModelScope.launch {
30        patientRepo.deleteAllPatients()
31    }
32    //a view model factory that sets the context for the viewmodel
33    //The ViewModelProvider.Factory interface is used to create view models.
34    class PatientViewModelFactory(context: Context) : ViewModelProvider.Factory {
35        private val context = context.applicationContext
36
37        override fun <T : ViewModel> create(modelClass: Class<T>): T =
38            PatientViewModel(context) as T
39    }
40 }
```

# Main Activity

In the main activity, we must do the following:

- Get an instance of the ViewModel using the following statement. You can place this statement in the onCreate function.

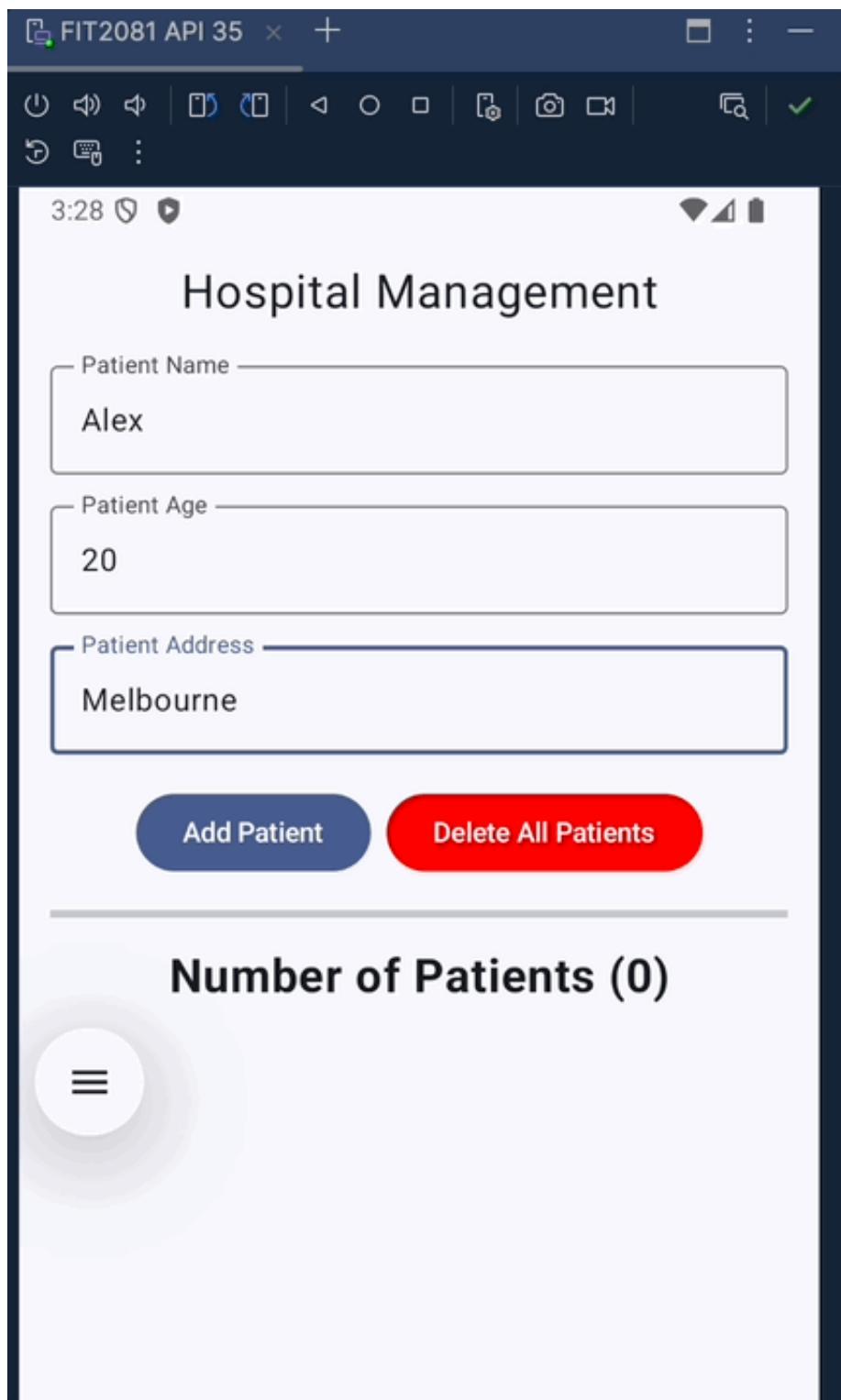
```
val viewModel: PatientViewModel = ViewModelProvider(  
    this, PatientViewModel.PatientViewModelFactory(this@MainActivity))  
)[PatientViewModel::class.java]
```

- Get the number of patients. The following statement retrieves the list of patients as a state. It gets updated every time the list of patients changes.

```
val numberOfPatients by viewModel.allPatients.collectAsState(initial = emptyList())
```

- Create Patient. This can be done by creating an instance of the Entity Patient and pass it to the insert function in the repository.

```
viewModel.insert(  
    Patient(  
        name = patientName.text,  
        age = patientAge.text.toInt(),  
        address = patientAddress.text  
    )  
)
```



# FIT2081 Week 6

## Data Handling &

## Persistence

# Lecture: Complete Guide to Databases

---

## Introduction

Welcome to our lecture on Room Database in Android development. Last week, we explored the MVVM architecture and the ViewModel component. Today, we'll build on that foundation by integrating persistent data storage through Room Database.

Consider our previous fitness app example: While the ViewModel helped manage the UI state and survive configuration changes, what happens when the user closes the app and reopens it later? All workout history would be lost because ViewModels don't persist data between app sessions. This is where Room Database comes in.

Room provides an abstraction layer over SQLite, making database operations more structured and manageable while seamlessly integrating with the MVVM architecture we've already learned.

## Learning Objectives

By the end of this lecture, you will be able to:

1. Understand the role of Room Database in the MVVM architecture
2. Implement the core components of Room: Entities, DAOs, and Database classes
3. Connect Room Database to ViewModels using repositories
4. Observe database changes in real-time with Flow/LiveData
5. Apply these concepts to create apps with persistent storage

# Room DB & MVVM

Room fits neatly into the MVVM architecture we discussed last week:

- **Model:** Now includes Room Entities (data classes annotated with `@Entity`) and the Room Database
- **ViewModel:** Remains responsible for preparing and managing UI state, but now sources data from the database
- **View:** Still observes the ViewModel and remains unchanged

This creates a clean, layered architecture:

1. **UI Layer:** Composables, Activities, Fragments
2. **UI Logic Layer:** ViewModels
3. **Data Layer:** Repository, Room Database (local storage), Network Client (remote storage)

The key difference from last week is that we're adding persistence. ViewModels still manage UI state during the lifecycle of a screen, but the actual data comes from and is saved to the Room Database, ensuring it survives app restarts.

## Room's Core Components

Room consists of three main components:

### 1. Entity

Entities are Kotlin data classes annotated with `@Entity`. Each entity represents a table in your database.

```
@Entity(tableName = "workouts")
data class Workout(
    @PrimaryKey(autoGenerate = true) val id: Long = 0,
    val type: String,
    val duration: Int,
    val caloriesBurned: Int,
    val date: Long
)
```

Entities can also define relationships, indexes, and constraints:

```
@Entity(
    tableName = "workout_sets",
    foreignKeys = [
        ForeignKey(
            entity = Workout::class,
            parentColumns = ["id"],
```

```

        childColumns = ["workoutId"],
        onDelete = ForeignKey.CASCADE
    )
],
indices = [Index("workoutId")]
)
data class WorkoutSet(
    @PrimaryKey(autoGenerate = true) val id: Long = 0,
    val workoutId: Long,
    val exerciseName: String,
    val reps: Int,
    val weight: Float
)

```

## 2. DAO (Data Access Object)

DAOs are interfaces annotated with `@Dao` that define the operations you can perform on your database.

```

@Dao
interface WorkoutDao {
    @Query("SELECT * FROM workouts ORDER BY date DESC")
    fun getAllWorkouts(): Flow<List<Workout>>

    @Query("SELECT * FROM workouts WHERE id = :workoutId")
    fun getWorkoutById(workoutId: Long): Flow<Workout>

    @Insert
    suspend fun insertWorkout(workout: Workout): Long

    @Update
    suspend fun updateWorkout(workout: Workout)

    @Delete
    suspend fun deleteWorkout(workout: Workout)
}

```

Notice how we're returning `Flow<List<Workout>>` rather than just `List<Workout>`. This is a key integration point with MVVM as it allows ViewModels to observe database changes in real-time, automatically updating the UI when data changes.

## 3. Database

The Database is an abstract class that extends `RoomDatabase` and serves as the main access point to your database.

```

@Database(
    entities = [Workout::class, WorkoutSet::class],
    version = 1
)
abstract class FitnessDatabase : RoomDatabase() {

```

```
abstract fun workoutDao(): WorkoutDao
abstract fun workoutSetDao(): WorkoutSetDao

companion object {
    @Volatile
    private var INSTANCE: FitnessDatabase? = null

    fun getDatabase(context: Context): FitnessDatabase {
        return INSTANCE ?: synchronized(this) {
            val instance = Room.databaseBuilder(
                context.applicationContext,
                FitnessDatabase::class.java,
                "fitness_database"
            )
                .fallbackToDestructiveMigration()
                .build()
            INSTANCE = instance
            instance
        }
    }
}
```

# Connecting Room to ViewModels

Now let's see how Room Database connects with ViewModels. The best practice is to use a Repository pattern to abstract the data source from the ViewModel.

## 1. Repository

The Repository mediates between different data sources (local database, network, etc.) and provides a clean API to the ViewModel.

```
class WorkoutRepository(private val workoutDao: WorkoutDao) {  
    // Expose data from the database  
    val allWorkouts: Flow<List<Workout>> = workoutDao.getAllWorkouts()  
  
    // Provide methods to modify the database  
    suspend fun insertWorkout(workout: Workout): Long {  
        return workoutDao.insertWorkout(workout)  
    }  
  
    suspend fun updateWorkout(workout: Workout) {  
        workoutDao.updateWorkout(workout)  
    }  
  
    suspend fun deleteWorkout(workout: Workout) {  
        workoutDao.deleteWorkout(workout)  
    }  
  
    fun getWorkoutById(id: Long): Flow<Workout> {  
        return workoutDao.getWorkoutById(id)  
    }  
}
```

## 2. ViewModel

The ViewModel now uses the Repository to access data and expose it to the UI.

```
class WorkoutViewModel(application: Application) : AndroidViewModel(application) {  
    private val repository: WorkoutRepository  
    val allWorkouts: StateFlow<List<Workout>>  
  
    init {  
        val database = FitnessDatabase.getDatabase(application)  
        repository = WorkoutRepository(database.workoutDao())  
  
        // Convert Flow to StateFlow for Compose  
        allWorkouts = repository.allWorkouts  
            .stateIn(  
                viewModelScope,
```

```

        SharingStarted.Lazily,
        emptyList()
    )
}

fun insertWorkout(type: String, duration: Int, caloriesBurned: Int) {
    viewModelScope.launch {
        val workout = Workout(
            type = type,
            duration = duration,
            caloriesBurned = caloriesBurned,
            date = System.currentTimeMillis()
        )
        repository.insertWorkout(workout)
    }
}

fun deleteWorkout(workout: Workout) {
    viewModelScope.launch {
        repository.deleteWorkout(workout)
    }
}
}

```

### 3. UI (Compose)

Finally, the UI observes the ViewModel's state and displays it:

```

@Composable
fun WorkoutListScreen(viewModel: WorkoutViewModel = viewModel()) {
    val workouts by viewModel.allWorkouts.collectAsState()

    LazyColumn {
        items(workouts) { workout ->
            WorkoutItem(
                workout = workout,
                onDeleteClick = { viewModel.deleteWorkout(workout) }
            )
        }
    }
}

```

# Best Practices for Room with MVVM

When integrating Room with MVVM, follow these best practices:

- 1. Use Repositories:** Always implement the Repository pattern to abstract data sources from ViewModels.
- 2. Return Flow/LiveData from DAOs:** This enables reactive UI updates when data changes.
- 3. Move Database Operations Off the Main Thread:** Use `suspend` functions with Kotlin Coroutines for database operations to keep the UI responsive.
- 4. Consider Database Migration Strategies:** Plan how to handle schema changes as your app evolves.
- 5. Use Type Converters for Complex Types:** Implement `TypeConverter` to store complex objects like Date or custom classes.
- 6. Be Mindful of Query Performance:** Optimize your queries, especially for large datasets.
- 7. Keep Entities Focused:** Each entity should represent a single concept in your app.

## Common Challenges and Solutions

When working with Room and ViewModels, you might encounter these challenges:

- 1. Database Operations Blocking the UI**
  - Solution: Use Coroutines with proper dispatchers (`Dispatchers.IO`)
- 3. Complexity with Many-to-Many Relationships**
  - Solution: Use junction tables or `@Relation` annotations
- 5. Type Mismatches in Queries**
  - Solution: Implement TypeConverters
- 7. Managing Database Migrations**
  - Solution: Use Migration classes or `fallbackToDestructiveMigration()` for development
- 9. Testing Database Code**
  - Solution: Use in-memory databases for unit tests

## Conclusion

Today we've seen how Room Database builds upon the MVVM architecture we learned last week:

- ViewModels manage UI state and business logic
- Room provides persistent storage for data
- Repositories connect ViewModels to Room
- Flow/LiveData enables reactive UI updates

This combination creates a powerful, maintainable architecture for Android apps with persistent data storage.

# Code Breakdown: Factory Class

## What does the code do?

```
class PlayerViewModelFactory(context: Context) : ViewModelProvider.Factory {  
    private val context = context.applicationContext  
  
    override fun <T : ViewModel> create(modelClass: Class<T>): T =  
        PlayerViewModel(context) as T  
}
```

It's ok if you are confused! This is one of the key new coding concepts in Week 6 - Android.

Answer: This code defines a **factory class** that knows how to create instances of `PlayerViewModel`, which likely requires a `Context` as a constructor parameter.

## What is a Factory?

In software design, a **factory** is a pattern used to create objects without exposing the creation logic to the client. Instead of calling `PlayerViewModel(...)` directly from the outside, we ask the factory to produce it for us.

In Android, we use `ViewModelProvider.Factory` to tell the system **how to construct custom ViewModels**, especially if they need constructor parameters (like `Context`, `Repository`, etc.).

By default, Android's ViewModel system expects a no-arg constructor. If your ViewModel needs something passed in, like a `Context`, you must create your own Factory.

## What's this `<T : ViewModel>` thing?

This is **generics** in Kotlin/Java:

```
override fun <T : ViewModel> create(modelClass: Class<T>): T
```

Let's break it down:

- `<T : ViewModel>`: This means **T is a generic type that must extend ViewModel**.
- `modelClass: Class<T>`: The caller passes in a class object for the type of ViewModel they want.
- `as T`: We're creating a `PlayerViewModel` and **casting it to type T**. This cast is safe because we know that's the class we're supporting.

Summary: This lets us write **one method that works for any ViewModel** (as long as we're careful to cast the correct one).

## What does `context.applicationContext` do?

```
private val context = context.applicationContext
```

This ensures that the **application-level context** is used instead of an Activity or Fragment context. This is a **best practice**, because application context lives longer and avoids memory leaks.

## Article of the Week

<https://lucianonooijen.com/blog/why-i-stopped-using-ai-code-editors/>

# Why I stopped using AI code editors

April 1, 2025

Reading time: 12 minutes

Categories: Tools

Tags: AI

*TL;DR: I chose to make using AI a manual action, because I felt the slow loss of competence over time when I relied on it, and I recommend everyone to be cautious with making AI a key part of their workflow.*

# **W6\_Lecture: Databases in Android with Room & Jetpack Compose**

## **Understanding Data Persistence**

---

Mobile applications frequently need to store and manage data beyond a single app session. While beginners might start with simple key-value storage like SharedPreferences, real-world applications require more robust data management solutions.

### **Key Challenges in Mobile Data Storage**

1. **Persistence:** Maintaining data between app launches
2. **Scalability:** Supporting increasingly complex data structures
3. **Performance:** Efficient data retrieval and storage
4. **Data Integrity:** Ensuring data remains consistent and reliable

### **Why Room Database?**

Room solves multiple challenges faced by mobile developers: - **Compile-time Verification:** Catches database-related errors early - **Boilerplate Reduction:** Minimizes repetitive database code - **Kotlin Integration:** Provides seamless, type-safe database interactions - **Performance Optimization:** More efficient than raw SQLite - **Android Jetpack Compatibility:** Works perfectly with modern Android architecture components



**Industry Insight:** Major companies like Uber, Airbnb, and LinkedIn rely on similar database patterns in their mobile applications.

## **Part 1: Why Databases Matter in Mobile Development**

---

As Android applications grow in complexity, we often need to persist user data across app launches. Up to this point, you've worked with **SharedPreferences**, which is great for storing small amounts of key-value data (e.g., a boolean for dark mode, or a user's login state). However, SharedPreferences is not designed for structured or relational data.

This week, we transition to using **Room Database**, which is Google's modern wrapper

around SQLite. It gives us a much more scalable, testable, and Kotlin-friendly way to manage local data. In fact, Room is part of **Jetpack Architecture Components**, so it plays nicely with ViewModels, LiveData/StateFlow, and Compose.

🎯 Today's example application: **Tennis Player Manager** - The app allows users to add tennis players (name, ranking, top venue). - Data is stored in a Room database on the device. - Each player is displayed as a card in a `LazyColumn`.

This practical example aligns closely with **Assignment 2**, where you will design and build a Room-backed Compose app. We're intentionally moving away from Shared Preferences to get you familiar with relational data storage using modern practices.

---

## Part 2: Conceptual Overview – Room Architecture

---

Room operates like a mini-database engine within your app. Here's what you need to build a Room-enabled app:

Component	Description
<b>Entity</b>	A Kotlin class representing a table in your database. Each field = a column.
<b>DAO (Data Access Object)</b>	Interface that defines how you interact with your DB (insert, delete, query).
<b>Room Database</b>	Abstract class that ties everything together, including creating the actual SQLite database.
<b>Repository</b>	Optional, but strongly encouraged. Helps decouple your database logic from your UI code.
<b>ViewModel</b>	Stores and manages UI-related data and survives configuration changes. Exposes the data to Compose.

Using Room this way follows the **MVVM architecture**, which promotes separation of concerns and makes your codebase more maintainable.

---

## Part 3: Setting Up Room in a Kotlin Project

---

Before coding, you need to update your project dependencies.

### Why?

Room uses **annotations** (`@Entity`, `@Dao`, etc.) that require Kotlin Symbol Processing (KSP) to generate boilerplate code under the hood. If we skip this step, our app won't compile.

## ✓ Gradle Setup

### Project-level `build.gradle.kts`

```
kotlin plugins { id("com.google.devtools.ksp") version "1.9.0-1.0.13" }
```

### App-level `build.gradle.kts`

```
dependencies { implementation("androidx.room:room-runtime:$roomversion")
    ksp("androidx.room:room-compiler:$roomversion")
    implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:2.6.2") } ````
```

Now sync Gradle. You are ready to use Room.

---

## Part 4: Building the Database Layer

---

We'll build the **Tennis Player Manager** app one layer at a time.

### 1. Entity – Represents a Table

```
@Entity(tableName = "players")
data class Player(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val name: String,
    val ranking: Int,
    val topVenue: String
)
```

Each instance of `Player` is one row in the `players` table.

- `@Entity` tells Room to treat this class as a table.
  - `@PrimaryKey(autoGenerate = true)` ensures unique, auto-incremented IDs.
  - You don't need to write SQL table creation code. Room handles that for you!
- 

### 2. DAO – Interface to Define DB Operations

```

@Dao
interface PlayerDao {
    @Insert suspend fun insertPlayer(player: Player)
    @Delete suspend fun deletePlayer(player: Player)
    @Query("SELECT * FROM players ORDER BY id ASC") fun getAllPlayers(): List<Player>
    @Query("DELETE FROM players") suspend fun deleteAll()
}

```

## Why use **Flow** ?

`Flow<List<Player>>` allows the UI to **automatically update** whenever the data changes. It's a reactive stream. When the DB changes, Compose recomposes the UI. This replaces the old `LiveData` approach.

---

## 3. Database – Main Access Point

```

@Database(entities = [Player::class], version = 1, exportSchema = false)
abstract class PlayerDatabase : RoomDatabase() {
    abstract fun playerDao(): PlayerDao

    companion object {
        @Volatile private var INSTANCE: PlayerDatabase? = null

        fun getDatabase(context: Context): PlayerDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    PlayerDatabase::class.java,
                    "player_database"
                ).build()
                INSTANCE = instance
                instance
            }
        }
    }
}

```

## Why `synchronized`, `@Volatile`, and `singleton`?

This ensures that only **one instance** of the database exists across the whole app. Multiple instances would lead to crashes or inconsistent data.

---

## Part 5: Repository Pattern – A Clean Abstraction

```
class PlayerRepository(context: Context) {  
    private val dao = PlayerDatabase.getDatabase(context).playerDao()  
  
    val allPlayers = dao.getAllPlayers()  
  
    suspend fun insert(player: Player) = dao.insertPlayer(player)  
    suspend fun deleteAll() = dao.deleteAll()  
}
```

Repositories abstract away data sources. In the future, you could:

- Add **networking support** (Firebase, Retrofit)
- Switch to a **remote DB** with zero changes to your ViewModel or UI

## Part 6: ViewModel + ViewModelFactory

```
class PlayerViewModel(context: Context) : ViewModel() {  
    private val repo = PlayerRepository(context)  
    val players = repo.allPlayers  
  
    fun insert(player: Player) = viewModelScope.launch {  
        repo.insert(player)  
    }  
  
    fun deleteAll() = viewModelScope.launch {  
        repo.deleteAll()  
    }  
}
```

Because `PlayerViewModel` needs a `Context`, we use a **factory**:

```
class PlayerViewModelFactory(private val context: Context) : ViewModelProvider.Factory {  
    override fun <T : ViewModel> create(modelClass: Class<T>): T {  
        return PlayerViewModel(context) as T  
    }  
}
```

## Part 7: Jetpack Compose UI – Wiring It All Together

Here's how your UI might look:

```
val viewModel: PlayerViewModel = viewModel(
    factory = PlayerViewModelFactory(context)
)
val players by viewModel.players.collectAsState(initial = emptyList())

LazyColumn {
    items(players) { player ->
        PlayerCard(player, onDelete = { viewModel.deletePlayer(player) })
    }
}
```

## Why Compose + Flow rocks:

- Compose watches the `Flow` data stream.
  - When the DB updates, `collectAsState()` triggers a recomposition.
  - No manual observers. Just clean, declarative UI.
- 

## Part 8: Sharing, Deleting, and Actions

```
fun sharePlayer(context: Context, player: Player) {
    val message = "Player: ${player.name}, Rank: ${player.ranking}, Venue: ${player.venue}"
    val intent = Intent(Intent.ACTION_SEND).apply {
        type = "text/plain"
        putExtra(Intent.EXTRA_TEXT, message)
    }
    context.startActivity(Intent.createChooser(intent, "Share via"))
}
```

User actions like delete or share can be handled in buttons within the `PlayerCard()` composable.

---

## Summary: Key Takeaways

- **Room is essential** for scalable, persistent data on-device.
- It plays nicely with Compose and ViewModel thanks to `Flow`.
- Use a **clean architecture**: DAO → Database → Repository → ViewModel → UI
- Learn it now – you'll use this structure in all real-world apps and internships.
- This approach prepares you for adding APIs, cloud sync, and Firebase in coming weeks.

---

## What's Next?

---

- In **Week 7**, you'll learn how to **connect to external APIs** using Retrofit and parse JSON.
- 

## Further Reading

---

- [Room Documentation](#)
- [Jetpack Compose + State](#)
- [Flow in Kotlin](#)

# Why SharedPreferences is Not Suitable for Complex Data Storage

## Understanding SharedPreferences: A Limited Storage Mechanism

---

SharedPreferences is often the first data storage method that Android developers encounter. While it serves a purpose for very simple data storage, it quickly becomes inadequate for robust mobile applications. This section will explore the critical limitations that make SharedPreferences unsuitable for complex data management.

### 1. Structural Limitations

#### Key-Value Storage Constraints

SharedPreferences is fundamentally a key-value storage system. This means:

- You can only store primitive types and simple string values
- Complex objects require manual serialization and deserialization
- No support for relational or structured data

**Example of Limitations:** ````kotlin // SharedPreferences can only store simple data  
val prefs = context.getSharedPreferences("MyApp", Context.MODE\_PRIVATE)

```
// Simple storage works fine prefs.edit().putString("username", "johndoe").apply()  
prefs.edit().putInt("user_age", 30).apply()
```

```
// Complex object? Not so straightforward class User(val name: String, val email: String, val  
preferences: List) // Storing this requires manual conversion to JSON or other serialization  
method ````
```

### 2. Performance Bottlenecks

#### Synchronous Writing and Performance Issues

- Writes to SharedPreferences block the main thread by default
- Entire file is rewritten on every edit
- Becomes significantly slower with increased data volume

- Unsuitable for frequent updates or large datasets

**Performance Comparison:** I Operation I Shared Preferences I Room Database I I-----I--  
-----|-----I I Writing Small Data I Acceptable I Efficient I I Writing Large Data I  
Slow I Optimized I I Concurrent Writes I Problematic I Thread-safe I I Query Performance I  
Limited I Indexed and Fast I

### **3. Data Integrity and Validation Challenges**

## Lack of Built-in Data Validation

- No schema enforcement
  - No type checking beyond basic primitives
  - Prone to data inconsistency
  - Manual validation required for every operation

**Risks of Unvalidated Data:** - Potential app crashes - Unpredictable behavior - Security vulnerabilities - Increased development complexity

#### **4. Scalability Limitations**

## Scaling Challenges

As your application grows, SharedPreferences becomes increasingly problematic:

- No support for complex queries
- Cannot handle relationships between data entities
- Limited to in-memory storage
- No built-in mechanism for data migrations

## 5. Security Concerns

## Basic Security Vulnerabilities

- Stored in plain text
  - Easily accessible by rooted devices
  - No encryption support by default
  - Sensitive information can be exposed

## 6. Complex Data Modeling Impossibility

## **Relationship and Relational Data**

SharedPreferences cannot handle: - One-to-many relationships - Many-to-many relationships - Complex object graphs - Structured data with multiple interdependent entities

**Comparison Scenario:** ````kotlin // SharedPreferences: Difficult to represent complex data // Imagine storing user details with multiple addresses

```
// Room Database: Clean, structured approach @Entity data class User( @PrimaryKey val id: Int, val name: String, val email: String )
```

```
@Entity data class Address( @PrimaryKey val id: Int, val userId: Int, // Foreign key relationship val street: String, val city: String )```
```

## When to Use SharedPreferences

---

Despite its limitations, SharedPreferences remains useful for: - Storing app settings - Saving simple user preferences - Caching small amounts of data - Temporary flag storage

## Recommended Alternatives

---

1. **Room Database:** Full-featured, type-safe local storage
2. **DataStore:** Modern Jetpack replacement for SharedPreferences
3. **SQLite:** For more complex local database needs
4. **Realm:** Third-party mobile database solution

## Conclusion

---

While SharedPreferences served as a simple storage mechanism in early Android development, modern mobile applications require more robust, performant, and secure data storage solutions. Room Database and other alternatives provide the scalability, performance, and features necessary for professional mobile application development.

## Learning Progression

1. Understand SharedPreferences limitations
2. Learn Room Database fundamentals
3. Practice complex data modeling
4. Implement best practices in data persistence

**Pro Tip:** Treat data storage as a critical architectural decision in your mobile application design.

# Lab: Database Pt. 2

---

## Assignment 1: Interview - Important Notes

Hi everyone,

Here are crucial notes on the interview for Assignment 1, which will be held during your lab session in week 6.

- **No interview, no mark** for the - Coding Interview (6 marks / 20%).
- No submission on Moodle, no interview.
- You must attend your allocated lab to finish it.
- Tutors will follow a randomly ordered list of students displayed on the screen.
  - Before entering the table, if you're not sure which lab you're in, just go to Allocate+ and check your timetable — you'll be able to see your lab time and lab number there.
  - You can access this [link](#) to find your lab, and find your name to get your interview order. We will conduct the interviews in order, so if your turn is later, you can come a bit later (but not too late).
    - At the bottom of the spreadsheet, there are multiple sheets. You need to go to the one corresponding to your lab to find your name. **Please double-check!** Make sure you enter the correct lab page.
  - If you're not present when the tutor calls your name, your turn will be moved to the end of the list.
  - Please use your student account to access the link above.
- The interview should take no more than 7 minutes.
- The interview is to measure your understanding of the source code you have submitted, You can go to this [link](#) to find the sample questions, and be prepared for your interview.
- Be ready for the interview. When your name is next on the list, be ready with the following open on your Laptop before approaching the interview tutor:
  - Open Android Studio and load the project.
  - Open the Moodle assignment submission page.
- Since the interview will take place during the lab, you need to study the Week 6 lab content in your own time.
  - We will provide a video specifically explaining the Week 6 lab content. You can watch it while waiting for your interview.
  - If you have any questions, feel free to attend the consultation.
- **After completing the interview**, please check the table to see if the interviewer's name is listed next to yours. If not, contact your interviewer as soon as possible and ask them to fill in their name.

# Overview

## This Week in Lab 6: Building on Room Database

In Week 5, you started exploring **Room Database** — the tool we use in Android for storing data locally on the device.

This week, we're taking that a step further.

## What You'll Be Building

You'll create a **complete data management flow**:

- Adding new items
- Viewing a list of saved items
- Deleting items from that list

That means we're covering **Create, Read, and Delete** — the **C, R, D** from CRUD.

## What's New This Week?

The big new addition is:

**LazyColumn** — Jetpack Compose's smart way to display long scrollable lists.

Why is this important?

It only renders what's currently visible on screen.

Super useful when dealing with large datasets.

(And yes, you'll likely need it in your assignments )

## What You'll Learn

By the end of this lab, you'll be able to:

- Set up persistent storage using Room
- Connect that database to your UI
- Show the data efficiently using `LazyColumn`
- Let users **add** and **delete** entries in real time

## Why This Matters

This lab covers patterns you'll use again and again in real-world Android apps.

Whether you're building a to-do list, saving user preferences, or syncing with a cloud database — these core concepts are foundational.

## Before You Come to Lab...

Make sure to complete **Part 2 of Lab 5** if you haven't already.

If you've already done that, this will feel like your **second run-through** — and you'll pick things up much faster.

## Further Notes

- LazyColumn - Jetpack Compose's efficient solution for displaying scrollable lists of data.
- The lab will guide you through creating a complete data management system where you'll define entities with Room annotations, implement a DAO (Data Access Object) with queries for adding, retrieving, and deleting items, and set up a Room Database instance.
- You'll then connect this database to your UI using a ViewModel as the intermediary. The highlight of this lab will be implementing LazyColumn to efficiently display database items, which only renders the visible items on screen rather than the entire list at once - an essential technique for performance optimization when working with large datasets. You'll also add interactive elements to create new entries and delete existing ones, providing hands-on experience with the full data lifecycle.

---

## Instructions for Today's Lab

In lieu of a lab-based explanation (due to interviews), please watch this recording in the next tab, we will cover:

- Database fundamentals
- Room database layers
- How to configure the Room database
- How to add packages, classes and interfaces to the Room database
- How to read and write from the Room database
- How to use LazyColumn with the Room database

# Lab Recording

In this recording, we will cover:

- Database fundamentals
- Room database layers
- How to configure the Room database
- How to add packages, classes and interfaces to the Room database
- How to read and write from the Room database
- How to use LazyColumn with the Room database

Can't See the Video? Here's How to Fix It:

- **Step 1: Open a new browser tab.**

Just press `Ctrl+T` (or `Cmd+T` on Mac) to open a fresh tab.

- **Step 2: Go to any Monash login page.**

The easiest way is to visit [my.monash](https://my.monash) or open your Monash Gmail inbox.

- **Step 3: Log in with your Monash email and Okta.**

Complete the MFA authentication if prompted.

- **Step 4: Come back to this tab and refresh the page.**

Once you're logged in, return here and hit the refresh button — the Panopto video should now load properly!

# LazyColumn and Room DB

As mentioned in the **Overview** section, we will add more features to the DB app we developed in week 5, part 2. We will list all the items in the hospital database using the LazyColumn UI element. Additionally, we will implement deleting a patient by ID and sharing the patient's details with other apps.

Let's start with the database layers.

- **Entity**

No need to make any change to this class as we have no intention to introduce a new attribute to the patient

- **Database**

No change is needed as we will keep the database name and version.

- **DAO**

In this interface, we will add a new function that accepts the patient's ID as a parameter from the repository class and then invokes a query to delete that patient only.

```
1 // Deletes a patient from the database by ID
2 @Query("DELETE FROM patients WHERE id = :patientId")
3 suspend fun deletePatientById(patientId: Int)
```

codesnap.dev

- **Repository**

We will add a new function that takes the patient's ID from the ViewModel and passes it to the DAO.

```
1 // Function to delete a patient by ID from the database.
2 suspend fun deletePatientById(id: Int) {
3     patientDao.deletePatientById(id)
4 }
5
```

codesnap.dev

- **ViewModel**

In the ViewModel, the new function takes the patient's ID from the screen controller and sends it to the Repository.

```
1 // deletes a patient by its id
2 fun deletePatientById(patientId: Int) = viewModelScope.launch {
3     patientRepo.deletePatientById(patientId)
4 }
```

codesnap.dev

- **Screen Controller**

- The first step is fetching the patient list from the view model.

```
val listOfPatients by viewModel.allPatients.collectAsState(initial = emptyList())
```

- Now, add a LazyColumn beneath the horizontal separator we implemented in week 5.

```
1 LazyColumn(
2     modifier = Modifier
3         .fillMaxWidth()
4         .weight(1f) // Makes LazyColumn take up remaining space
5 ) {
6     // iterates through the list of patients and creates a card for each patient
7     items(listOfPatients) { patient ->
8 }
```

codesnap.dev

- For each patient, let's add a new Card.

```
1 Card(
2     modifier = Modifier
3         .padding(8.dp)
4         .height(100.dp)
5         .weight(1f),
6         elevation = CardDefaults.cardElevation(defaultElevation = 6.dp)
7 ) {}
```

codesnap.dev

Inside each card, we will add a row with two columns. The first column will show the patient's

details, while the second will list two buttons vertically as depicted below.

**Name: Alex**

Age: 23

Address: Melbourne



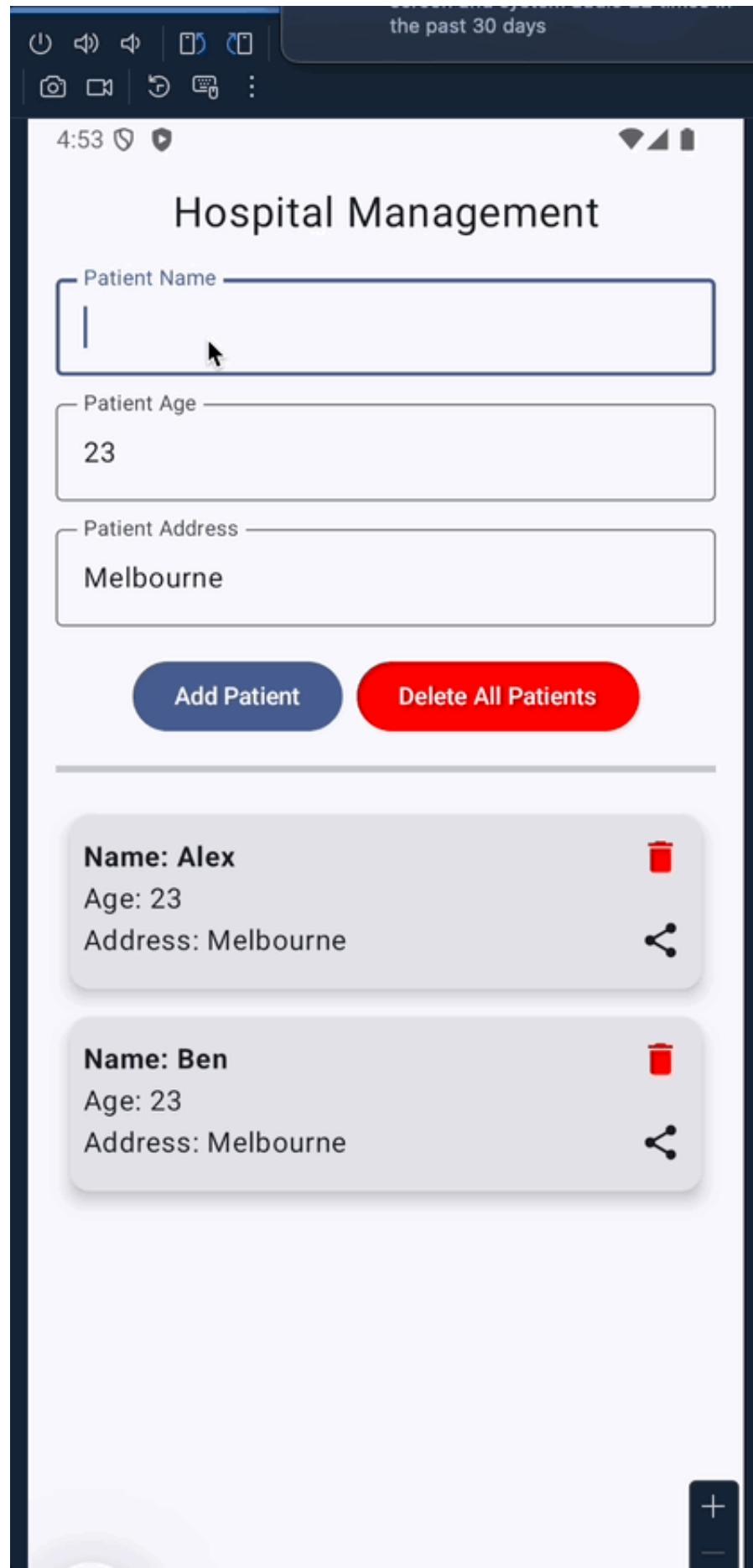
```

1  {
2      Row(
3          modifier = Modifier.fillMaxWidth(),
4          verticalAlignment = Alignment.CenterVertically
5      ) {
6          //First column to display patient details
7          Column(modifier = Modifier
8              .padding(8.dp)
9              .weight(1f)) {
10             Text(text = "Name: ${patient.name}", fontWeight = FontWeight.Bold)
11             Text(text = "Age: ${patient.age}")
12             Text(text = "Address: ${patient.address}")
13         }
14         //Second column to display buttons
15         Column {
16             // This button deletes the patient when clicked
17             IconButton(onClick = {
18                 viewModel.deletePatientById(patient.id)
19             })
20         }
21         Icon(
22             Icons.Filled.Delete,
23             contentDescription = "Delete",
24             tint = Color.Red
25         )
26     }
27     // this button shares the patient details when clicked
28     IconButton(onClick = {
29         val message = "Patient Name: ${patient.name}\n" +
30             "Age: ${patient.age}\n" +
31             "Address: ${patient.address}"
32         val intent = Intent(Intent.ACTION_SEND).apply {
33             type = "text/plain"
34             putExtra(Intent.EXTRA_TEXT, message)
35         }
36
37         context.startActivity(
38             Intent.createChooser(
39                 intent,
40                 "Share patient details"
41             )
42         )
43     })
44     Icon(
45         Icons.Filled.Share, contentDescription = "Share"
46     )
47 }
48 }
49
50
51 }
52
}

```

As you can see in the code above, we retrieved the patient's details (lines 10-12) and placed them in Text UI elements. We also called the deletePatientById function (see line 18), which is implemented in the ModelView, and passed it the patient's ID to be deleted.

Here is the expected output:



# Source Code

## A Note on Using This Week's Source Code

We've released this week's source code a little earlier than usual — this is to support you while tutors are busy with interview assessments. But please keep these important tips in mind:

- **Only check the source code if you're genuinely stuck.**

Give yourself a bit of time to explore and debug first — this helps you learn how to problem-solve, which is *one of the most important developer skills*.

- **Struggling is part of the process.**

The best learning often happens when you hit a roadblock and figure it out yourself. That moment of "ah-ha!" sticks with you far more than copying a solution.

- **Jumping straight to the answer short-circuits your growth.**

You'll find future topics more difficult if you rely on the code too early. The more effort you invest in understanding what's going wrong, the more confident you'll be later when things get complex.

- **Use the source as a learning tool, not a solution.**

If you do look at the code, try to *reverse engineer* it: ask yourself why each part is written the way it is. Try modifying it or re-implementing a piece from scratch.

- **Be honest with yourself.**

Would you know how to do it again next week in a new project? If not, keep digging a little longer before peeking.

## Room DB + LazyColumn



[week6\\_db.zip](#)

# Week 6 Self-Paced Guide: Room Database + LazyColumn Integration

## 1. Introduction: Why Use a Database?

---

In many real-world apps, you need to **store information** that persists even when the app is closed or the device is restarted. A database allows you to:

- Save data in a **structured** format.
- **Filter and retrieve** specific data easily.
- Maintain **data integrity** (e.g., no duplicate entries for unique fields).
- Handle **multiple users or background processes** accessing the data at the same time.
- **Scale** your app as the data grows.

Compared to plain text files or SharedPreferences, databases are more flexible and powerful—especially when working with lists, relationships between entities, or large volumes of data.

---

## 2. What is a Relational Database?

---

A **relational database** organizes data into **tables**. Each table contains:

- **Columns** (attributes of the data, e.g., `name`, `age`)
- **Rows** (individual records)
- A **Primary Key** (a unique identifier for each row)

Tables can relate to one another. For example:

- A `students` table could relate to a `courses` table using a `student_id`.
  - This structure enables efficient data lookups and updates.
- 

## 3. Introducing Room in Android

---

**Room** is a persistence library in Android that provides an abstraction layer over SQLite. It simplifies working with databases by using Kotlin/Java code instead of writing SQL queries

directly.

Room handles:

- Creating and updating tables
- Managing data retrieval and storage
- Reducing boilerplate code

Room components include:

- **Entity**: Represents a table.
  - **DAO (Data Access Object)**: Interface containing methods to access the database.
  - **RoomDatabase**: Abstract class that ties everything together.
- 

## 4. Room Architecture Layers

---

To effectively manage your app's data, Room relies on a layered architecture. Here's how it breaks down:

Layer	Description
<b>Entity</b>	A Kotlin <code>data class</code> that represents a table structure. Marked with <code>@Entity</code> .
<b>DAO</b>	An interface where you define methods like <code>insert()</code> , <code>delete()</code> , and custom queries. Marked with <code>@Dao</code> .
<b>RoomDatabase</b>	An abstract class that ties together your DAO and entities.
<b>Repository</b>	Optional class that abstracts away data sources (e.g., from Room or a web API).
<b>ViewModel</b>	Stores and manages UI-related data. Interacts with the Repository and exposes data to the UI using <code>Flow</code> or <code>LiveData</code> .
<b>UI (Composable)</b>	Displays the data and triggers actions via ViewModel functions.

---

## 5. Defining a Room Entity

---

Each table is represented by a Kotlin data class, annotated with `@Entity`.

Example:

```

@Entity(tableName = "patients")
data class Patient(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val name: String,
    val age: Int,
    val address: String
)

```

## 6. Creating a DAO Interface

The DAO (Data Access Object) defines methods for interacting with your data.

**Example:**

```

@Dao
interface PatientDao {
    @Insert
    suspend fun insert(patient: Patient)

    @Delete
    suspend fun delete(patient: Patient)

    @Query("SELECT * FROM patients ORDER BY id ASC")
    fun getAllPatients(): Flow<List<Patient>>

    @Query("DELETE FROM patients WHERE id = :id")
    suspend fun deleteById(id: Int)
}

```

`@Insert, @Delete, @Query` tell Room what to do behind the scenes.

Use `Flow<List<T>>` to observe changes in the data.

**## 7. Setting Up the Database Class**

The Room database class connects your entities and DAO.

**Example:**

```

```kotlin
@Database(entities = [Patient::class], version = 1)
abstract class HospitalDatabase : RoomDatabase() {
    abstract fun patientDao(): PatientDao
}

```

```
version
```

 is required to manage schema migrations.

Use a singleton pattern to ensure there's only one instance of your database at runtime.

## 8. Creating a Repository (Optional)

A Repository acts as a middleman between the ViewModel and DAO. It's especially helpful when you have multiple data sources (e.g., network and local storage).

```
class PatientRepository(private val dao: PatientDao) {  
    val allPatients = dao.getAllPatients()  
  
    suspend fun insert(patient: Patient) = dao.insert(patient)  
    suspend fun delete(patient: Patient) = dao.delete(patient)  
    suspend fun deleteById(id: Int) = dao.deleteById(id)  
}
```

You can skip the repository if you're working with a single data source.

## 9. Connecting to ViewModel

The ViewModel accesses the Repository or DAO directly to fetch and expose data to the UI.

```
class PatientViewModel(application: Application) : AndroidViewModel(application){  
    private val dao = HospitalDatabase.getDatabase(application).patientDao()  
    private val repository = PatientRepository(dao)  
  
    val patients: Flow<List<Patient>> = repository.allPatients  
  
    fun insert(patient: Patient) = viewModelScope.launch {  
        repository.insert(patient)  
    }  
  
    fun delete(patient: Patient) = viewModelScope.launch {  
        repository.delete(patient)  
    }  
  
    fun deleteById(id: Int) = viewModelScope.launch {  
        repository.deleteById(id)  
    }  
}
```

## 10. Displaying Data with LazyColumn in Compose

The UI (in Jetpack Compose) should interact only with the ViewModel, not with Repository or DAO directly.

Use LazyColumn to efficiently display lists.

Example:

```
val patients by viewModel.patients.collectAsState(initial = emptyList())

LazyColumn {
    items(patients) { patient ->
        Card {
            Column {
                Text(text = "Name: ${patient.name}")
                Text(text = "Age: ${patient.age}")
                Text(text = "Address: ${patient.address}")
                IconButton(onClick = { viewModel.deleteById(patient.id) })
                    Icon(Icons.Default.Delete, contentDescription = "Delete")
            }
        }
    }
}
```

LazyColumn only renders what's on screen, improving performance.

Use collectAsState() to observe changes from Flow.

## 11. Using App Inspection to View the Database

---

To inspect your Room database:

- Run your app on the emulator.
- Open View → Tool Windows → App Inspection.
- Select your device and app.
- Explore the databases tab to view tables and records.
- Enable Live Updates to see changes in real-time.

## 12. Adding Room Dependencies in Gradle

---

You need to configure Gradle correctly for Room to work.

Project-level build.gradle:

```
plugins {
    <Previous entries here....>
    id("com.google.devtools.ksp") version "2.0.21-1.0.27" apply false
}
```

App-level build.gradle:

```
plugins {
    <Previous entries here....>
    id 'com.google.devtools.ksp'
}

dependencies {
    val room_version = "2.6.1"
    implementation("androidx.room:room-runtime:$room_version")

    // If this project uses any Kotlin source, use Kotlin Symbol Processor
    // See Add the KSP plugin to your project
    ksp("androidx.room:room-compiler:$room_version")

    // If this project only uses Java source, use the Java annotationProcessor
    // No additional plugins are necessary
    annotationProcessor("androidx.room:room-compiler:$room_version")

    // optional - Kotlin Extensions and Coroutines support for Room
    implementation("androidx.room:room-ktx:$room_version")
    implementation ("androidx.lifecycle:lifecycle-viewmodel-compose:$room_version")
    <Other entries here....>
}
```

## 13. Summary

---

By completing this lab, you should understand how to:

- Define Room entities and DAO interfaces
- Set up a Room database
- Connect it to a ViewModel
- Display data using LazyColumn
- Delete data entries by ID
- Use coroutines for database operations
- Inspect your database using Android Studio tools

These are foundational skills for any mobile developer working with persistent data.

## 14. Suggested Next Steps

---

- Explore update operations in Room.
- Add input validation before inserting data.
- Practice designing and populating your own tables.
- In next week, try integrating with a remote API and using a Repository to switch between local and remote sources.

# FIT2081 Week 7

## Networking & APIs

# Week 7 - Lecture Reading Materials

---

## Introduction

Welcome to our lecture on Networking & APIs in Android development. Last week, we explored the Room Database. Today, we'll learn how to manage external data sources, such as data retrieved from Web APIs.

Consider our previous fitness app example: if you want to add useful features such as offering educational materials, including recipe ideas, meal planning tips, and nutritional information from online resources (e.g., [Edamam Recipe](#), [Edamam Nutrition Analysis](#), etc), you don't have to download and save such information into your app manually. Instead, you can access and show the information from the online resource in real time on demand with HTTP requests.

**HTTP (Hypertext Transfer Protocol)** is a protocol used for **communication between web browsers and servers**. It defines how data is **requested, transmitted, and received** over the internet.

**Web Services** are function calls made using **HTTP** (i.e., made over the Web). They are a way of CRUDing remote data, allowing software components running on different devices or platforms to **exchange data and perform operations remotely**. They used to involve (and many still do) rigorous messaging protocols and data formats.

## Learning Objectives

By the end of this lecture, you will be able to:

- understand fundamental concepts, such as what HTTP, Web Services, RESTful API, and JSON are;
- understand the importance of using JSON to store and transfer data, as well as how to parse JSON to human-readable format;
- understand what Retrofit is;
- and use Retrofit in Compose projects.

# HTTP Basics

## What is HTTP

**HTTP (Hypertext Transfer Protocol)** is a protocol used for **communication between web browsers and servers**. It defines how data is **requested, transmitted, and received** over the internet.

## How HTTP Works

- **Client (Browser/App) makes a request** → A user enters a URL or clicks a link.
- **Server processes the request** → The server finds the requested resource (e.g., a webpage or image).
- **Server sends a response** → The response includes the requested data and a status code.
- **Client renders the response** → The browser displays the webpage.

## Key Features of HTTP

- **Stateless** → Each request is independent; the server doesn't remember previous interactions.
- **Request-Response Model** → Clients send requests; servers send responses.
- **Text-Based Protocol** → Requests and responses are in human-readable format.

## HTTP Methods/Verbs

HTTP has defined a set of methods that indicates the type of action to be performed on the resources.

- **GET** method requests data from the resource and should not produce any side effects. E.g. **GET /actors** returns the list of all actors.
- **POST** method requests the server to create a resource in the database, mostly when a web form is submitted. E.g **POST /movies** creates a new movie where the movie's property could be in the request's body. **POST** is non-idempotent which means multiple requests will have different effects.
- **PUT** method requests the server to update a resource or create the resource if it doesn't exist. E.g. **PUT /movie/23** will request the server to update, or create, if doesn't exist, the movie with id =23. **PUT** is idempotent, which means multiple requests will have the same effects.
- **DELETE** method requests that the resources, or their instance, should be removed from the database. E.g. **DELETE /actors/103** will request the server to delete an actor with ID=103 from

the actor's collection.

## HTTP Response Status Codes

When the client sends an HTTP request to the server, the client should get feedback on whether it passed or failed. HTTP status codes are a set of standardized codes which has various explanations in various scenarios. For example:

- Successful responses
  - 200 OK
  - 201 Created
  - 202 Accepted
- Client error responses
  - 400 Bad Request
  - 404 Not Found
  - 405 Method Not Allowed
- Server error responses
  - 500 Internal Server Error
  - 501 Not Implemented
  - 503 Service Unavailable

The full list of status codes can be found [HERE](#).

# Web Services

**Web Services** are function calls made using HTTP (i.e., made over the Web). They are a way of CRUDing remote data, allowing software components running on different devices or platforms to **exchange data and perform operations remotely**. They used to involve (and many still do) rigorous messaging protocols and data formats.

Modern Web Services use a RESTful paradigm, which is much simpler. It uses the standard HTTP verbs GET, POST, PUT and DELETE to signal R, C, U, and D data operations. It also employs a URL scheme such that each URL on the server represents a resource, which can be a collection resource or an element resource.

For example, to get the details about a country from a Web Service that offers such data, we could make the following HTTP GET request: "<https://restcountries.com/v3.1/name/australia>". You can try this now. Just paste the URL into your browser's address bar. The Web Service will send back data in JSON format. Your browser will realise this is not HTML, so it will not attempt to render a Web page but just dump the JSON data onto the screen (JSON is just text, so this is possible).

```
[{"name":{"common":"Australia","official":"Commonwealth of Australia","nativeName":{"eng":{"official":"Commonwealth of Australia","common":"Australia"}}, "tld": ".au","cca2":"AU","ccn3":"036","cca3":"AUS","cioc":"AUS","independent":true,"status":"officially-assigned","unMember":true,"currencies":{"AUD": {"name":"Australian dollar","symbol":"$"}}, "id": {"root": "+6","suffixes":["1"]}, "capital":["Canberra"], "altSpellings": ["AU"], "region": "Oceania", "subregion": "Australia and New Zealand", "languages": {"eng": "English"}, "translations": {"ara": {"official": "أستراليا", "common": "أستراليا"}, "bre": {"official": "Kenglad Aostralia", "common": "Aostralia"}, "ces": {"official": "Australské společenství", "common": "Austrálie"}, "cym": {"official": "Cymanwlad Awstralia", "common": "Awstralia"}, "deu": {"official": "Commonwealth Australien", "common": "Australien"}, "est": {"official": "Austraalia Ühendus", "common": "Austraalia"}, "fin": {"official": "Australian liittovaltio", "common": "Australia"}, "fra": {"official": "Australie", "common": "Australie"}, "hrv": {"official": "Commonwealth of Australia", "common": "Australija"}, "hun": {"official": "Ausztrál Államzövetség", "common": "Ausztrália"}, "ita": {"official": "Commonwealth dell'Australia", "common": "Australia"}, "jpn": {"official": "オーストラリア連邦", "common": "オーストラリア"}, "kor": {"official": "오스트레일리아 연방", "common": "호주"}, "nld": {"official": "Gemenebest van Australië", "common": "Australië"}, "per": {"official": "Australia"}, "pol": {"official": "Związek Australiski", "common": "Australia"}, "por": {"official": "Comunidade da Austrália", "common": "Austrália"}, "rus": {"official": "Содружество Австралии", "common": "Австралия"}, "slk": {"official": "Australsky zväz", "common": "Austrália"}, "spa": {"official": "Mancomunidad de Australia", "common": "Australia"}, "srp": {"official": "Комонвент Аустралија", "common": "Аустралија"}, "swe": {"official": "Australiska statsförbundet", "common": "Australien"}, "tur": {"official": "Avustralya Federal Devleti", "common": "Avustralya"}, "urd": {"official": "اُسٹریلیا", "common": "اُسٹریلیا"}, "zho": {"official": "澳大利亚联邦", "common": "澳大利亚"}, "latlng": [-27.0, 133.0], "landlocked": false, "area": 7692024.0, "demonyms": {"eng": {"F": "Australian", "M": "Australian"}, "fra": {"F": "Australienne", "M": "Australien"}, "flag": "\u2b89D83C\u2b89DDE6\u2b89D83C\u2b89DDFA", "maps": {"googleMaps": "https://goo.gl/maps/Dcjada7ubhnZTndH6", "openStreetMaps": "https://www.openstreetmap.org/relation/80500"}, "population": 25687041, "gini": {"2014": 34.4}, "fifa": "AUS", "car": {"signs": [{"AUS": "left"}], "timezones": [{"UTC+05:00": "UTC+06:30", "UTC+06:30": "UTC+07:00", "UTC+07:00": "UTC+08:00", "UTC+08:00": "UTC+09:30", "UTC+09:30": "UTC+10:00", "UTC+10:00": "UTC+10:30", "UTC+10:30": "UTC+11:30"}]}, "continents": ["Oceania"], "flags": {"png": "https://flagcdn.com/w320/au.png", "svg": "https://flagcdn.com/au.svg", "alt": "The flag of Australia has a dark blue field. It features the flag of the United Kingdom – the Union Jack – in the canton, beneath which is a large white seven-pointed star. A representation of the Southern Cross constellation, made up of one small five-pointed and four larger seven-pointed white stars, is situated on the fly side of the field."}, "coatOfArms": {"png": "https://mainfacts.com/media/images/coats_of_arms/au.png", "svg": "https://mainfacts.com/media/images/coats_of_arms/au.svg"}, "startOfWeek": "monday", "capitalInfo": {"latlng": [-35.27, 149.13]}, "postalCode": {"format": "####", "regex": "^(\\d{4})$"}}]}
```

## Two Main Types of Web Services

### 1. SOAP (Simple Object Access Protocol)

SOAP is a **protocol-based web service** that uses **XML** for communication. It follows strict standards and works over **HTTP, SMTP, TCP/IP, and more**.

#### Features of SOAP:

- Uses **XML** for structured messaging.
- Supports **WS-Security** (encryption, authentication).

- Works with multiple protocols (HTTP, SMTP, etc.).
- Requires a **WSDL (Web Services Description Language)** file to describe the service.

## SOAP Request Example (XML)

A SOAP request to get user details:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"  
                  xmlns:web="http://example.com/user">  
    <soapenv:Header/>  
    <soapenv:Body>  
        <web:GetUser>  
            <web:UserId>123</web:UserId>  
        </web:GetUser>  
    </soapenv:Body>  
</soapenv:Envelope>
```

## SOAP Response Example

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">  
    <soapenv:Body>  
        < GetUserResponse>  
            < User>  
                < Id>123</Id>  
                < Name>John Doe</Name>  
                < Email>john@example.com</Email>  
            </User>  
        </ GetUserResponse>  
    </soapenv:Body>  
</soapenv:Envelope>
```

## When to Use SOAP?

- ✓ High **security** (e.g., **banking, healthcare, enterprise apps**).
- ✓ Needs **strict rules** (e.g., formal contracts via **WSDL**).
- ✓ **Multi-protocol support** (not limited to HTTP).
- ✗ **Not recommended** for lightweight web services due to its complexity.

## 2. REST (Representational State Transfer)

REST is an **architectural style** for designing web services that use **HTTP methods** to perform operations on resources.

### Features of REST:

- Uses standard **HTTP methods** (`GET`, `POST`, `PUT`, `DELETE`).

- Returns data in **JSON** or **XML** (JSON is preferred).
- **Stateless** (each request is independent).
- Faster and simpler than SOAP.
- No need for **WSDL** (unlike SOAP).

## REST HTTP Methods

- **GET**: Retrieve data (e.g., fetch user details).
- **POST**: Create new data (e.g., add a user).
- **PUT**: Update existing data (e.g., update user profile).
- **DELETE**: Remove data (e.g., delete a user).

## REST API Example (JSON)

A REST API request to get user details:

```
GET https://api.example.com/users/123
```

### Response (JSON)

```
{
  "id": 123,
  "name": "John Doe",
  "email": "john@example.com"
}
```

## When to Use REST?

- ✓ **Web applications, mobile apps** (e.g., social media, e-commerce).
  - ✓ **Fast and lightweight APIs** (e.g., public APIs like Twitter, Google Maps).
  - ✓ Works well with **JavaScript-based front-end frameworks** (React, Angular, Vue).
- ✗ **Not suitable** for applications needing strict security or formal contracts.

## RESTful API vs. SOAP API

Feature	RESTful API	SOAP API
Protocol	Uses HTTP	Uses XML-based protocol
Data Format	JSON (mostly), XML	Only XML
Speed	Fast	Slower
Security	HTTPS, OAuth, JWT	WS-Security, SSL
Best for	Web & mobile apps	Banking, enterprise apps

There are other technologies of Web Services, such as **GraphQL** (Best for flexible, client-driven APIs (e.g., Facebook, Shopify)) and **gRPC** (Best for high-speed microservices and real-time applications).

# RESTful API

## What is RESTful API?

A **RESTful API** (Representational State Transfer API) is a web service that follows **REST architecture** principles, allowing applications to communicate over the internet using **HTTP methods**. It is a stateless architecture (sender or receiver retains, i.e. no information) that uses the Web's existing protocols and technologies. The REST architecture consists of:

- clients
- servers
- resources
- a vocabulary of HTTP operations known as request methods (such as GET, PUT, or DELETE).

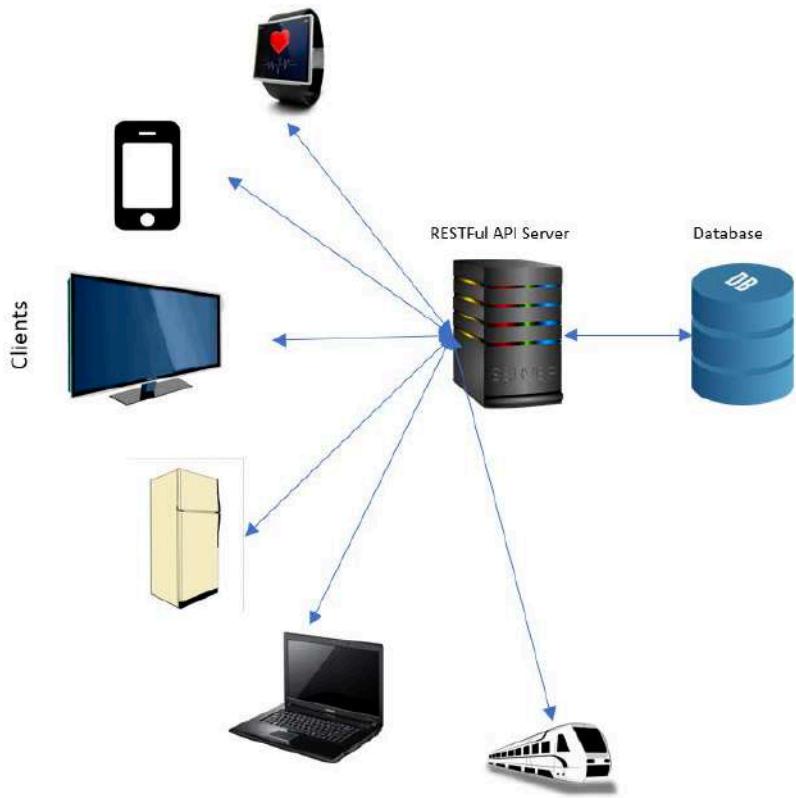
## Key Characteristics of RESTful APIs

- **Stateless** → Each request is independent; the server does not store the client state.
- **Client-Server Architecture** → The client (e.g., mobile app) and server (backend) are separate.
- **Uses HTTP Methods** → (GET, POST, PUT, DELETE, etc.).
- **Resource-Based** → Everything (users, products, posts) is treated as a **resource**.
- **Supports Multiple Data Formats** → Mostly **JSON**, but also XML, etc.

## Benefits of REST

- The separation between the client and the server
- The REST API is always independent of the type of platform or language.
  - With REST, data produced and consumed is separated from the technologies that facilitate production and consumption. As a result, REST performs well, is highly scalable, simple, and easy to modify and extend.
- Since REST is based on standard HTTP operations, it uses verbs with specific meanings, such as "get" or "delete", which avoids ambiguity. Resources are assigned individual URIs, adding flexibility.

The following diagram depicts the RESTful architecture that consists of a set of different types of clients sending requests and receiving responses from a RESTful API server, which is, in turn, connected to a database (such as MongoDB).



## Example REST API Endpoints

### a) GET Request (Retrieve User Data)

```
GET https://api.example.com/users/123
```

#### Response (JSON)

```
{
  "id": 123,
  "name": "John Doe",
  "email": "john@example.com"
}
```

### b) POST Request (Create a New User)

```
POST https://api.example.com/users
Content-Type: application/json
```

#### Request Body (JSON)

```
{
  "name": "Jane Doe",
  "email": "jane@example.com",
  "password": "securepassword"
}
```

## Response

```
{  
  "id": 124,  
  "message": "User created successfully"  
}
```

## c) PUT Request (Update User Information)

```
PUT https://api.example.com/users/123  
Content-Type: application/json
```

### Request Body

```
{  
  "name": "John Smith",  
  "email": "johnsmith@example.com"  
}
```

## Response

```
{  
  "message": "User updated successfully"  
}
```

## d) DELETE Request (Remove a User)

```
DELETE https://api.example.com/users/123
```

## Response

```
{  
  "message": "User deleted successfully"  
}
```

# JSON and JSON Parsing

 Question: What is JSON?

 Answer: JSON is a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is developed by [Douglas Crockford](#).

## Example:

Represent the following data items in JSON: Name is Tim, Age is 23, Address is Melbourne, and units taken are: FIT1051, FIT2095, and FIT2081.

```
{  
  "name": "Tim",  
  "age": 23,  
  "address": "Melbourne",  
  "units": [  
    "FIT1051",  
    "FIT2095",  
    "FIT2081"  
  ]  
}
```

## Syntax Rules

JSON syntax is derived from JavaScript object notation syntax:

- Data is in name/value pairs
- Commas separate data
- Curly braces hold objects
- Square brackets hold arrays

## Nested Data

Arrays and objects can be nested in JSON format. For example:

```
{  
  "firstname": "Tim",  
  "lastname": "John",  
  "websites": [  
    {  
      "description": "Company",  
      "url": "http://company.com",  
      "live": false  
    }  
  ]  
}
```

```
},
{
  "description": "School",
  "url": "http://school.com",
  "live": true
}
]
}
```

Another example of a JSON message received in response to  
<https://restcountries.com/v3.1/name/australia>

```
[
{
  "name": {
    "common": "Australia",
    "official": "Commonwealth of Australia",
    "nativeName": {
      "eng": {
        "official": "Commonwealth of Australia",
        "common": "Australia"
      }
    }
  },
  "tld": [
    ".au"
  ],
  "cca2": "AU",
  "ccn3": "036",
  "cca3": "AUS",
  "cioc": "AUS",
  "independent": true,
  "status": "officially-assigned",
  "unMember": true,
  "currencies": {
    "AUD": {
      "name": "Australian dollar",
      "symbol": "$"
    }
  },
  "idd": {
    "root": "+6",
    "suffixes": [
      "1"
    ]
  },
  "capital": [
    "Canberra"
  ],
  "altSpellings": [
    "AU"
  ],
  "region": "Oceania",
```

```
"subregion": "Australia and New Zealand",
"languages": {
    "eng": "English"
},
"translations": {
    "ara": {
        "official": "کومونولٹ اسٹرالیا",
        "common": "اُسٹرالیا"
    },
    "bre": {
        "official": "Kenglad Aostralia",
        "common": "Aostralia"
    },
    "ces": {
        "official": "Australské společenství",
        "common": "Austrálie"
    },
    "cym": {
        "official": "Cymanwlad Awustralia",
        "common": "Awustralia"
    },
    "deu": {
        "official": "Commonwealth Australien",
        "common": "Australien"
    },
    "est": {
        "official": "Austraalia Ühendus",
        "common": "Austraalia"
    },
    "fin": {
        "official": "Australian liittovaltio",
        "common": "Australia"
    },
    "fra": {
        "official": "Australie",
        "common": "Australie"
    },
    "hrv": {
        "official": "Commonwealth of Australia",
        "common": "Australija"
    },
    "hun": {
        "official": "Ausztrál Államszövetség",
        "common": "Ausztrália"
    },
    "ita": {
        "official": "Commonwealth dell'Australia",
        "common": "Australia"
    },
    "jpn": {
        "official": "オーストラリア連邦",
        "common": "オーストラリア"
    },
    "kor": {
```

```
"official": "오스트레일리아 연방",
"common": "호주"
},
"nld": {
  "official": "Gemenebest van Australië",
  "common": "Australië"
},
"per": {
  "official": "قلمرو همسود استراليا",
  "common": "استراليا"
},
"pol": {
  "official": "Związek Australijski",
  "common": "Australia"
},
"por": {
  "official": "Comunidade da Austrália",
  "common": "Austrália"
},
"rus": {
  "official": "Содружество Австралии",
  "common": "Австралия"
},
"slk": {
  "official": "Austrálsky zväz",
  "common": "Austrália"
},
"spa": {
  "official": "Mancomunidad de Australia",
  "common": "Australia"
},
"srp": {
  "official": "Комонвелт Аустралија",
  "common": "Аустралија"
},
"swe": {
  "official": "Australiska statsförbundet",
  "common": "Australien"
},
"tur": {
  "official": "Avustralya Federal Devleti",
  "common": "Avustralya"
},
"urd": {
  "official": "دولتِ مشترکہ آسٹریلیا",
  "common": "آسٹریلیا"
},
"zho": {
  "official": "澳大利亚联邦",
  "common": "澳大利亚"
}
},
"latlng": [
-27,
```

133  
],  
"landlocked": false,  
"area": 7692024,  
"demonyms": {  
 "eng": {  
 "f": "Australian",  
 "m": "Australian"  
 },  
 "fra": {  
 "f": "Australienne",  
 "m": "Australien"  
 }  
},  
"flag": "AU",  
"maps": {  
 "googleMaps": "https://goo.gl/maps/DcjaDa7UbhnZTndH6",  
 "openStreetMaps": "https://www.openstreetmap.org/relation/80500"  
},  
"population": 25687041,  
"gini": {  
 "2014": 34.4  
},  
"fifa": "AUS",  
"car": {  
 "signs": [  
 "AUS"  
 ],  
 "side": "left"  
},  
"timezones": [  
 "UTC+05:00",  
 "UTC+06:30",  
 "UTC+07:00",  
 "UTC+08:00",  
 "UTC+09:30",  
 "UTC+10:00",  
 "UTC+10:30",  
 "UTC+11:30"  
],  
"continents": [  
 "Oceania"  
],  
"flags": {  
 "png": "https://flagcdn.com/w320/au.png",  
 "svg": "https://flagcdn.com/au.svg",  
 "alt": "The flag of Australia has a dark blue field. It features the flag of the United Kingdom in the canton and a red Commonwealth Saltire superimposed on a light blue field."}  
},  
"coatOfArms": {  
 "png": "https://mainfacts.com/media/images/coats\_of\_arms/au.png",  
 "svg": "https://mainfacts.com/media/images/coats\_of\_arms/au.svg"  
},  
"startOfWeek": "monday",  
"capitalInfo": {

```
"latlng": [
    -35.27,
    149.13
],
},
"postalCode": {
    "format": "####",
    "regex": "^(\d{4})$"
}
}
]
```

So, in this case, one array with a single object element contains many property name/value pairs. Some of these values are themselves arrays or arrays of objects or objects.

## JSON Parsing in Kotlin

### Example using Gson

```
dependencies {
    implementation("com.google.code.gson:gson:2.8.8")
}

import com.google.gson.Gson

// Define a data class representing your JSON data
data class Person(val name: String, val age: Int)

// JSON string
val json = "{\"name\":\"John\", \"age\":30}"

// Deserialize JSON string to Kotlin object
val person: Person = Gson().fromJson(json, Person::class.java)

// Serialize Kotlin object to JSON string
val jsonString: String = Gson().toJson(person)
```

# Retrofit

Retrofit is a powerful HTTP client library developed by [Square](#), designed specifically for Android applications. It simplifies network requests and interaction with RESTful APIs by providing an elegant and easy-to-use API. Built on top of OkHttp, Retrofit supports JSON parsing, asynchronous requests, interceptors, and more. Below is a detailed breakdown of Retrofit from multiple perspectives.

## What is Retrofit?

**Retrofit is a type-safe HTTP client for Android that transforms HTTP APIs into Java/Kotlin interfaces. It allows developers to define network requests using annotations, making API calls more readable and maintainable.**

## Core Components of Retrofit

Retrofit consists of the following key components:

- **Retrofit.Builder** – Used to create a Retrofit instance, configuring the base URL, converter, call adapter, etc.
- **API Service Interface** – Defines API endpoints using Java/Kotlin interfaces with annotations like `@GET`, `@POST`, etc.
- **Call<T> / Observable<T>** – Represents network requests, supporting synchronous, asynchronous, and RxJava-based calls.
- **Converter** – Converts server responses into Java/Kotlin objects (e.g., Gson, Moshi, Jackson).
- **CallAdapter** – Adapts `Call<T>` to other return types, such as LiveData or RxJava's `Observable`.

## Basic Usage of Retrofit

| (1) Add Dependencies in "build.gradle(.kts) (Module :app)"

### Groovy DSL:

```
dependencies {  
    implementation 'com.squareup.retrofit2:retrofit:2.11.0'  
    implementation 'com.squareup.retrofit2:converter-gson:2.11.0' // Gson Converter  
}
```

### Kotlin DSL:

```
dependencies {  
    implementation("com.squareup.retrofit2:retrofit:2.11.0")  
    implementation("com.squareup.retrofit2:converter-gson:2.11.0") // Gson Converter
```

```
}
```

## (2) Add Internet Permission in "AndroidManifest.xml"

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

## (3) Create "Post" data class in a new Kotlin data class file "Post"

```
data class Post(  
    val userId: Int,  
    val id: Int,  
    val title: String,  
    val body: String  
)
```

## (4) Define API Interface in a new Kotlin Interface file "MyAPI"

```
import retrofit2.Call  
import retrofit2.http.GET  
  
interface MyAPI {  
    @GET("posts")  
    fun getPosts(): Call<List<Post>>  
}
```

## (5) Create a Retrofit Instance in the "MainActivity" file

```
val BASE_URL = "https://jsonplaceholder.typicode.com/"  
  
val api = Retrofit.Builder()  
    .baseUrl(BASE_URL)  
    .addConverterFactory(GsonConverterFactory.create())  
    .build()  
    .create(MyAPI::class.java)
```

## (6) Make a Network Request in the "MainActivity" file

```
val TAG: String = "CHECK_RESPONSE"  
  
api.getPosts().enqueue(object : Callback<List<Post>>{  
    override fun onResponse(p0: Call<List<Post>>, p1: Response<List<Post>>) {  
        if(p1.isSuccessful){  
            p1.body()?.let {  
                for (post in it){  
                    Log.i(TAG, "onResponse: ${post.body}")  
                }  
            }  
        }  
    }  
  
    override fun onFailure(p0: Call<List<Post>>, p1: Throwable) {  
        Log.i(TAG, "onFailure: ${p1.message}")  
    }  
}
```

```
    }  
})
```

## Advantages of Retrofit

**Easy to Use** – API definitions are clean and concise with annotation-based configuration.

**Highly Extensible** – Supports various converters (Gson, Moshi) and call adapters (RxJava, LiveData).

**Compatible with OkHttp** – Fully supports interceptors, request modification, and custom configurations.

**Asynchronous & Synchronous Support** – Developers can choose the appropriate method based on their needs.

## Limitations of Retrofit

⚠ **No Built-in WebSocket Support** – Retrofit is designed for REST APIs, not real-time WebSocket communication.

⚠ **No Automatic Retry Mechanism** – Requires manual configuration via OkHttp's `RetryInterceptor`.

⚠ **No Built-in Streaming Support** – Large file downloads must be handled manually.

## Relationship Between Retrofit and OkHttp

Retrofit relies on OkHttp for low-level HTTP communication, and OkHttp provides features such as:

- Connection management
- Automatic retries
- Connection pooling
- Interceptors (e.g., logging, authentication tokens)
- WebSocket support (which Retrofit does not natively support)

## Summary

Retrofit is one of the most popular HTTP clients for Android, offering a **clean and flexible** way to interact with RESTful APIs. Its key advantages include:

- **Annotation-based API design**
- **Built-in JSON conversion**

- **Highly extensible**
- **Support for synchronous, asynchronous, RxJava, and coroutines**



MONASH  
University

**FIT2081**  
**Mobile Application**  
**Development**



# Networking & APIs

**Week 7**

Jiazhou 'Joe' Liu



# Dr Jiazhou 'Joe' Liu

Assistant Lecturer

**Embodied Visualisation Group**

Department of Human-Centred Computing  
Faculty of IT

## Teaching Experience:

*Data Visualisation*

*Web Fundamentals*

*Full-Stack Development*

*User Interface Design and Usability*

*Computer/Data Science Project*

*Research Methods*

My research focuses on novel and effective interactions in immersive environments (VR, AR) for visualisation view management and AI in digital health and construction domain

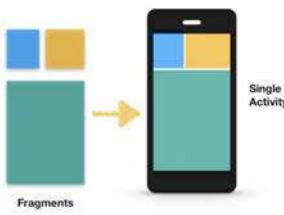
# Learning Objectives

- **Fundamental concepts**
  - Understanding HTTP and RESTful Web Services
    - HTTP on Android
  - JSON and JSON parsing
  - What is Retrofit
- **Using Retrofit in Compose projects**
  - Handling permissions (Internet)

# Android Development Learning Path



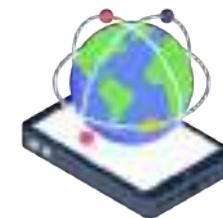
Views and Layouts



Activities and  
Fragments



Handle Events

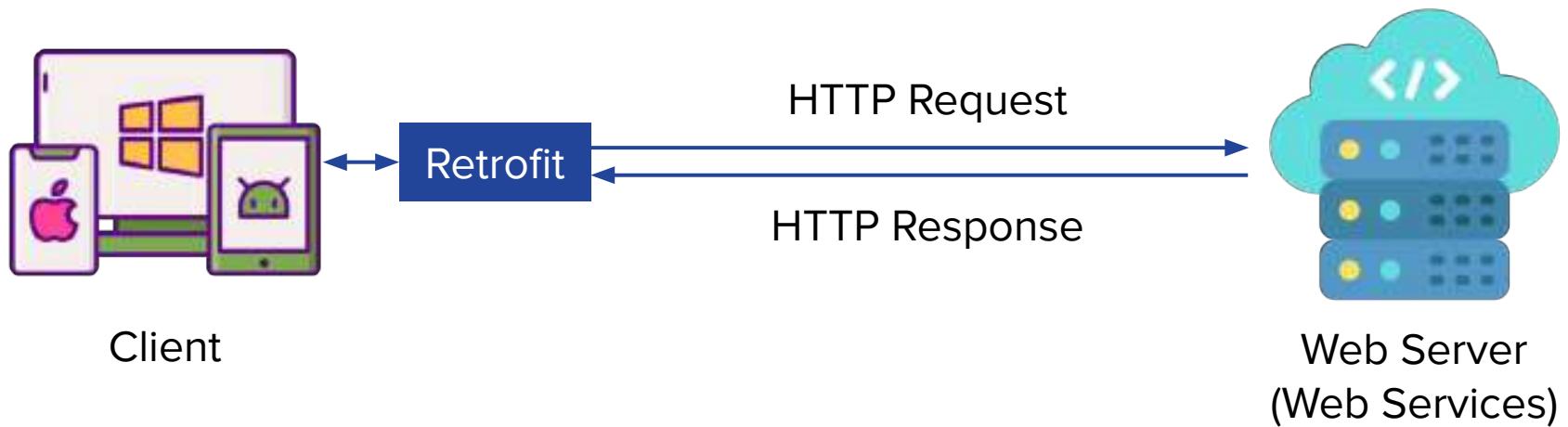


Network Connection



MONASH  
University

# Client, Web Services, HTTP, and Retrofit



MONASH  
University

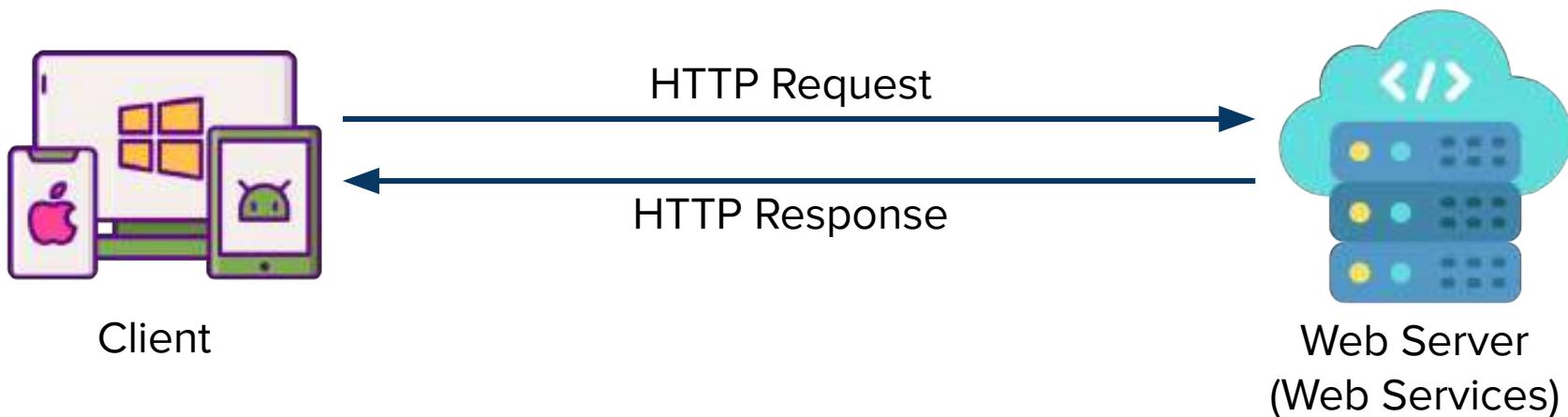
# HTTP Fundamentals

# Communication between client and server

**HTTP** Hypertext Transfer Protocol

It is an application-layer protocol for transmitting hypermedia documents, such as HTML.

<https://developer.mozilla.org/en-US/docs/Web/HTTP>



MONASH  
University

# HTTP Request

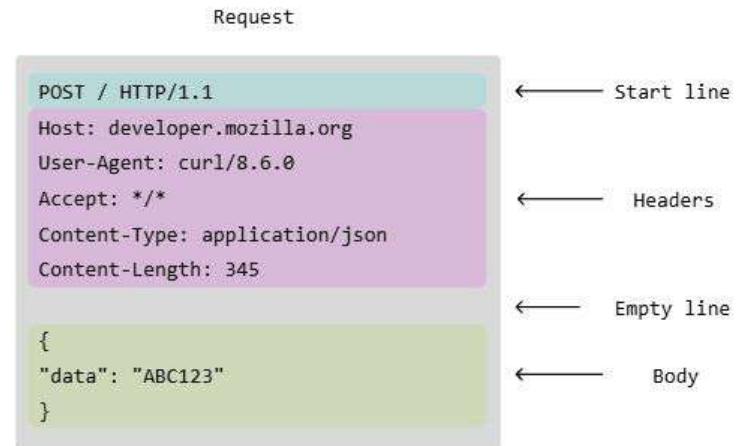
Start Line: HTTP method (HTTP verb), request-target (URL), HTTP version

HTTP headers: description of the message – Metadata

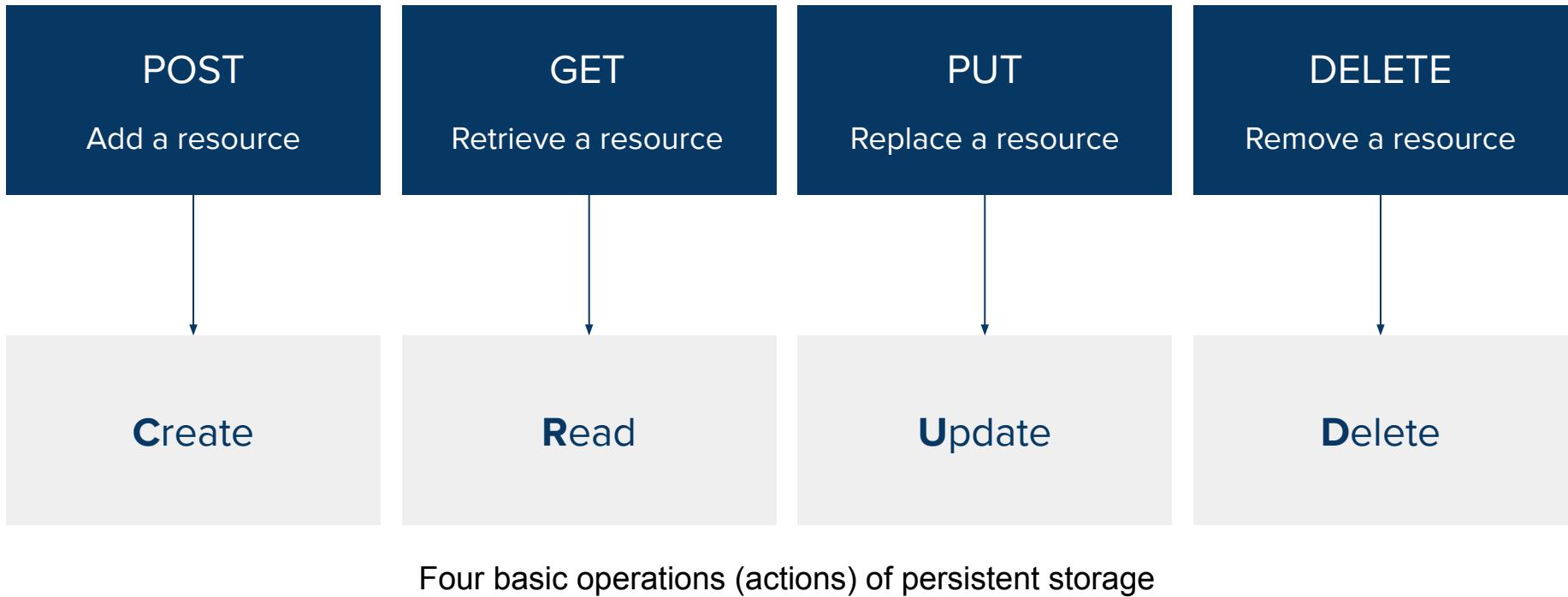
Request body: (optional) send data to server

E.g., JSON data for student object

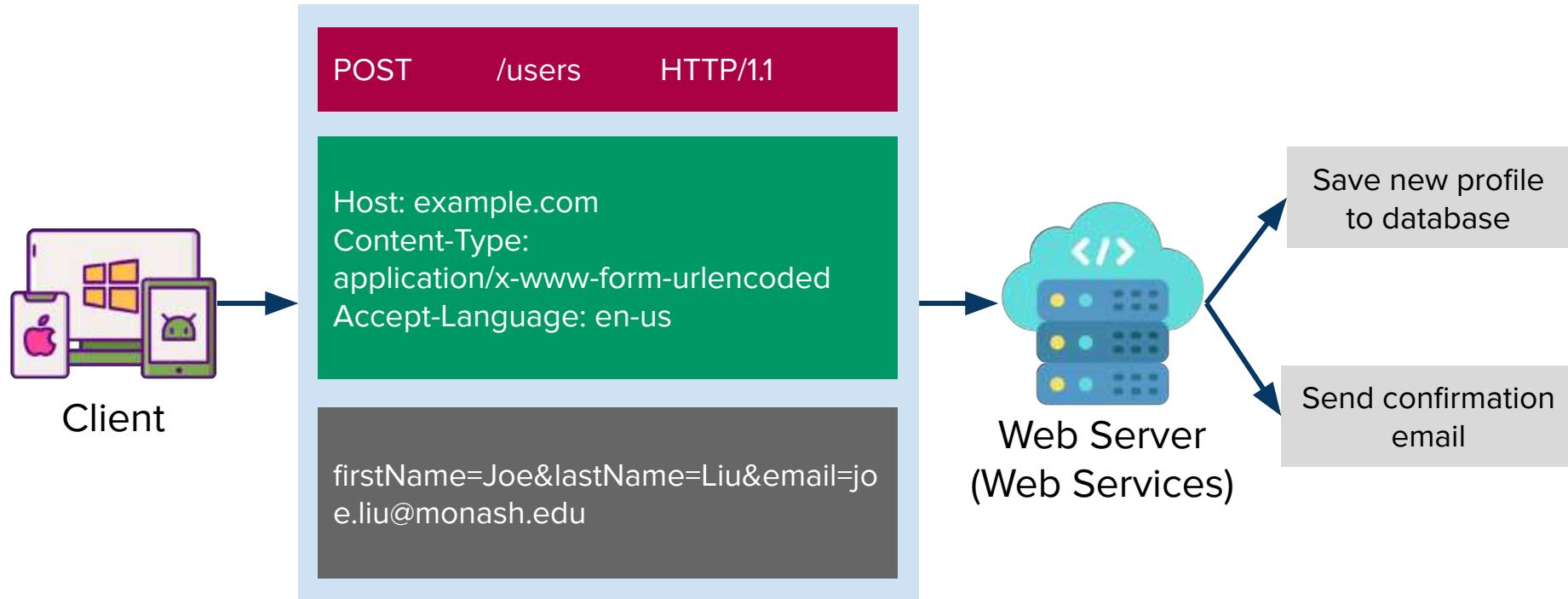
```
{  
  "name": "Joe",  
  "age": 18  
}
```



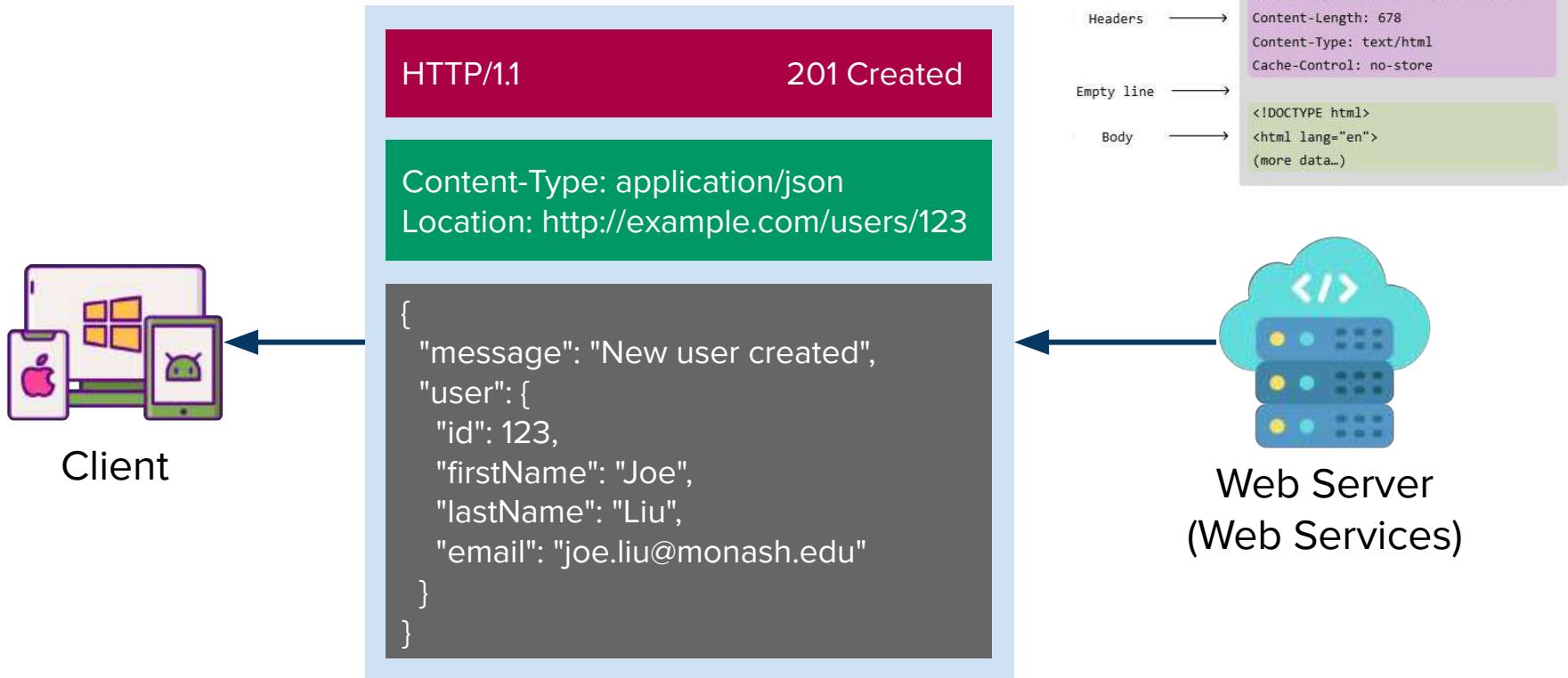
# Main HTTP Methods



# HTTP Request Example

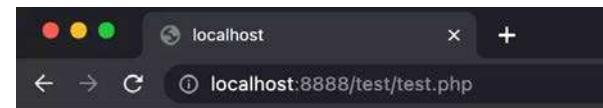
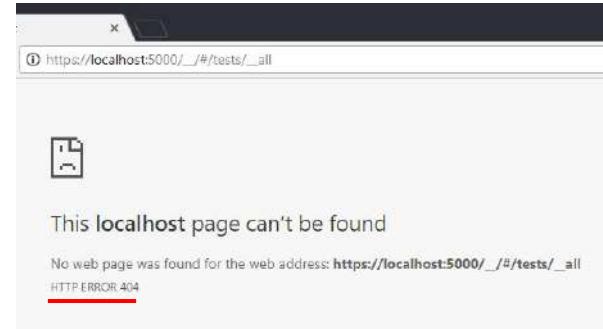


# HTTP Response



# Status Code

Status Code Range	Meaning
100 – 199	<b>Informational</b> responses
200 – 299	<b>Successful</b> responses
300 – 399	<b>Redirection</b> messages
400 – 499	<b>Client error</b> responses
500 – 599	<b>Server error</b> responses



This page isn't working

localhost is currently unable to handle this request.

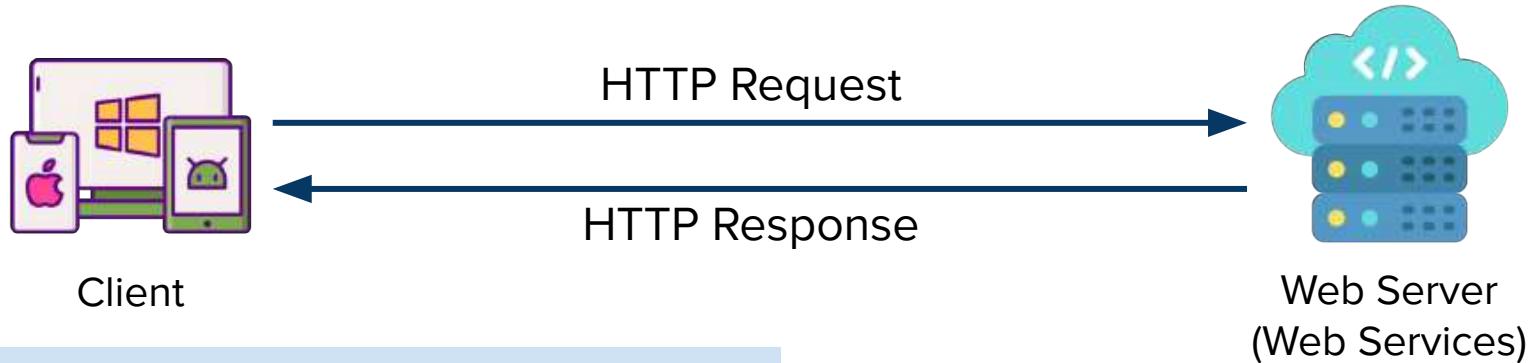
HTTP ERROR 500



MONASH  
University

# RESTful Web Service

# Web Service Patterns and Protocols



**SOAP** and **WSDL** web services are  
**XML-based** protocol/language

## URL Patterns

example.com/getUser  
example.com/addUser  
example.com/updateUser  
example.com/deleteUser

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns:m="http://www.example.org">
  <soap:Header>
    </soap:Header>
  <soap:Body>
    <m:GetStockPrice>
      <m:StockName>T</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Example of SOAP message



MONASH  
University

# RESTful Web Service

RESTful web services, or **REST APIs**, are a type of web service that follows the principles of **Representational State Transfer (REST)**

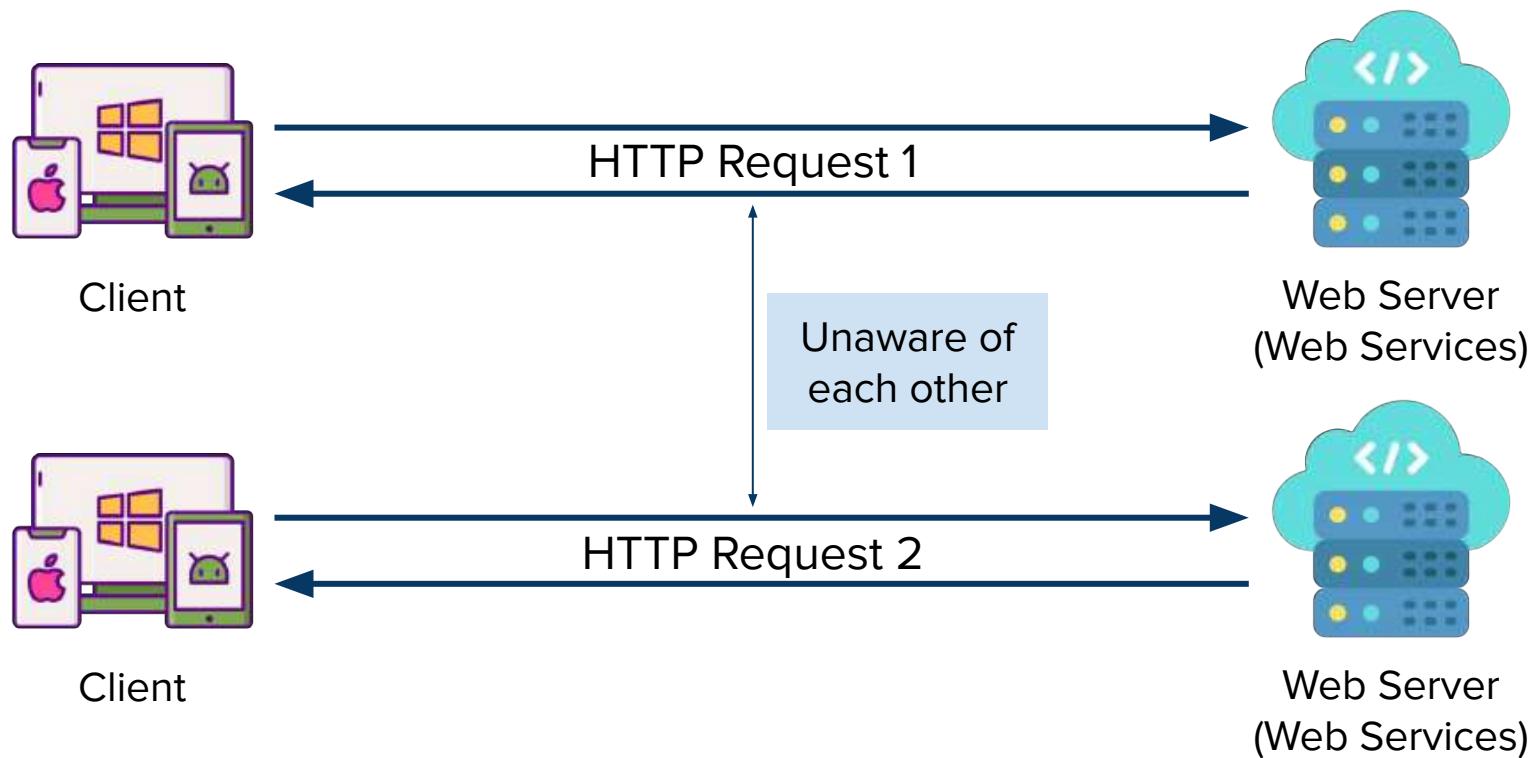
Stateless Client-Server  
Relationship

Utilise HTTP Methods  
(POST, GET, PUT, DELETE)

Structured and Consistent URLs

Consistent Data Type Transfer

# Stateless Client-Server Relationship



# HTTP Methods and Structured URLs

HTTP Method	Consistent URL	Web Service Operation
GET	example.com/users	Fetch User
POST	example.com/users	Add User
PUT	example.com/users	Update User
DELETE	example.com/users	Delete User

HTTP Method	Consistent URL	Web Service Operation
GET	example.com/users/123	Get user with ID of 123
DELETE	example.com/users/123/comments	Delete comments of user whose ID is 123
GET	example.com/users/123/email	Get email of the user whose ID is 123

# Consistent Data Type

JSON

XML

A web service is RESTful when it provides **stateless operations** to manage data using different **HTTP methods** and **structured URLs**

# **JSON and JSON parsing**

# JSON

- JSON stands for JavaScript Object Notation
- JSON is lightweight text-data interchange format
- JSON is "self-describing" and easy to understand
- JSON supports two structures:
  - Objects: a collection of name/value pairs {"firstName": "John"}
  - Arrays: an ordered list of values

```
{"phoneNumber":  
  [  
    {  
      "type": "home", "number": "212 555-1234"  
    },  
    {  
      "type": "fax", "number": "646 555-4567"  
    }  
  ]  
}
```

[https://www.w3schools.com/js/js\\_json\\_intro.asp](https://www.w3schools.com/js/js_json_intro.asp)



MONASH  
University

# JSON (cont'd)

- Objects in name/value pairs (properties) separated by a colon
- A value can be a string, a number, true/false or null, an object or an array
- Data is separated by commas
- Curly braces hold objects and square brackets hold arrays

```
{ "firstName": "John", "lastName": "Smith", "age": 25, "address": {  
    "streetAddress": "21 2nd Street",  
    "city": "New York",  
    "state": "NY",  
    "postalCode": 10021  
},  
"phoneNumber": [  
    {  
        "type": "home", "number": "212 555-1234"  
    },  
    {  
        "type": "fax", "number": "646 555-4567"  
    }]  
}
```

# Parsing JSON

- JSON parsing online
- <https://jsonformatter.org/json-parser>
- <https://jsoneditoronline.org/>

```
{ "firstName": "John", "lastName": "Smith", "age": 25,  
"address": { "streetAddress": "21 2nd Street", "city": "New  
York", "state": "NY", "postalCode": 10021  
}, "phoneNumbers": [ {"type": "home", "number": "212  
555-1234"}, {"type": "fax", "number": "646 555-4567"} ]}
```



```
▼ object {5}  
  firstName : John  
  lastName : Smith  
  age : 25  
  ▼ address {4}  
    streetAddress : 21 2nd Street  
    city : New York  
    state : NY  
    postalCode : 10021  
  ▼ phoneNumbers [2]  
    ▼ 0 {2}  
      type : home  
      number : 212 555-1234  
    ▼ 1 {2}  
      type : fax  
      number : 646 555-4567
```



# Popular HTTP Libraries

## *Pros & Cons*

# A Professional App should have ...



Authentication



Async Requests



Map JSON to  
usable objects

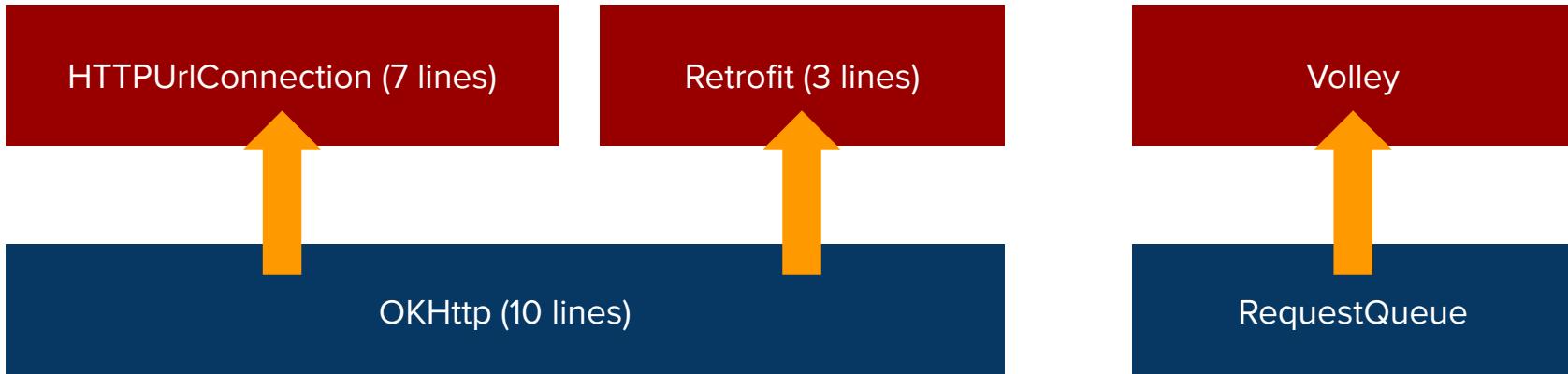


Load Images



MONASH  
University

# Popular HTTP libraries



They all use Background Threads. Asynchronous in nature.

# Disadvantages of HttpURLConnection

- Poor readability and less expressive
- Lots of boilerplate
  - Byte arrays, stream readers
- No built-in support for parsing JSON response
- Manage background threads manually
  - Poor resource management

<https://developer.android.com/reference/java/net/HttpURLConnection>

# Disadvantages of Volley

- Limited REST specific features
- Poor authentication layer
- Meagre documentation
- Smaller community

<https://github.com/google/volley>

# Introduction to Retrofit

# What is Retrofit

- Retrofit developed by Square <https://square.github.io/retrofit/>
- Retrofit facilitates **interactions with public APIs** in Android (http calls)
- Retrofit is built on top of OkHttp
- Retrofit supports adding and using converters such as **Gson libraries** to convert Java objects into their JSON representation or vice versa



# Why use Retrofit

- Active Community
  - Easier troubleshooting
- Expressive code with more abstraction
- Manage resources efficiently
  - Background threads
  - Async calls and queries
  - Automatic JSON parsing using GSON library
  - Automatic error handling callbacks
  - Built-in user authentication support

# Internet access permission & Retrofit Dependencies

- To use the internet, the manifest file must include the internet permission

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

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

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

    <application
        ...


```

- Using Retrofit, we need to add the required dependencies to the module level gradle file

```
implementation("com.squareup.retrofit2:retrofit:2.11.0")
implementation("com.squareup.retrofit2:converter-gson:2.11.0") // Gson Converter
```

# Retrofit Model Class

- To provide a mapping from the structure of the JSON's response body to Kotlin objects, we use **data classes** (other options are also possible)
- Using data classes, we can obtain the objects we want from the long body of the JSON response
- *E.g. in the Posts response (JSON) we want to access key-value pairs under the ‘Posts’ so we create a Retrofit Model class to map them*

```
data class Post(  
    val userId: Int,  
    val id: Int,  
    val title: String,  
    val body: String  
)
```

# Retrofit Model Class (cont'd)

The screenshot shows a code editor with a green header bar. The menu items are: New document 2, New, Open, Save, Copy. Below the menu is a toolbar with icons for text, tree, table, and various file operations. The main area displays a JSON array of posts:

```
[{"id": 1, "userId": 1, "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit", "body": "quia et suscipit\n    suscipit recusandae consequuntur expedita et cum\n    reprehenderit molestiae ut ut quas totam\n    nostrum rerum est autem sunt rem eveniet architecto"}, {"id": 2, "userId": 1, "title": "qui est esse", "body": "est rerum tempore vitae\n    sequi sint nihil reprehenderit dolor beatae ea dolores neque\n    fugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis\n    qui aperiam non debitis possimus qui neque nisi nulla"}, {"id": 3, "userId": 1, "title": "ea molestias quasi exercitationem repellat qui ipsa sit aut", "body": "et iusto sed quo iure\n    voluptatem occaecati omnis eligendi aut ad\n    voluptatem doloribus vel accusantium quis pariatur\n    molestiae porro eius odio et labore et velit aut"}, {"id": 4, "userId": 1, "title": "et harum quidem id est laborum", "body": "et harum quidem id est laborum"}, {"id": 5, "userId": 1, "title": "et harum quidem id est laborum", "body": "et harum quidem id est laborum"}, {"id": 6, "userId": 1, "title": "et harum quidem id est laborum", "body": "et harum quidem id est laborum"}, {"id": 7, "userId": 1, "title": "et harum quidem id est laborum", "body": "et harum quidem id est laborum"}, {"id": 8, "userId": 1, "title": "et harum quidem id est laborum", "body": "et harum quidem id est laborum"}, {"id": 9, "userId": 1, "title": "et harum quidem id est laborum", "body": "et harum quidem id est laborum"}, {"id": 10, "userId": 1, "title": "et harum quidem id est laborum", "body": "et harum quidem id est laborum"}]
```

```
data class Post(\n    val userId: Int,\n    val id: Int,\n    val title: String,\n    val body: String\n)
```

# Retrofit Interface

- Retrofit interface handles the HTTP API
- An interface defines http methods (**GET**) and the HTTP API (@Path or @Query)  
*[suspend fun will be covered next week]*
  - @Query is used to define query parameters for HTTP requests
  - @Path is used to define path parameters that are included in a URL path
- The Model class should **match the returned type** in the Retrofit Interface

```
interface MyAPI {  
    @GET("posts")  
    fun getPosts(): Call<List<Post>>  
}
```

<https://square.github.io/retrofit/>

```
interface MyAPI {  
    @GET("Posts/{id}")  
    suspend fun getPostsByID(  
        @Path("id") id: Int  
    ): Call<List<Post>>  
}
```

[Understand Query and Path in HTTP Requests](#)

# Retrofit Builder

- **Retrofit.Builder** is used to create **an instance of Retrofit** by calling the **build()**
- The build() uses the **baseUrl** and the **Gson converter factory** provided to
- create the Retrofit instance
- We then call the **create()** on the Retrofit instance and pass the Retrofit interface

```
val BASE_URL = "https://jsonplaceholder.typicode.com/"

val api = Retrofit.Builder()
    .baseUrl(BASE_URL)
    .addConverterFactory(GsonConverterFactory.create())
    .build()
    .create(MyAPI::class.java)
```

# Retrofit Response

- The **enqueue()** function asynchronously send the “**GET**” request defined in our API via **getPosts()** and notify Callback of its responses. We need to override the default behaviours for functions **onResponse()** and **onFailure()**.

```
val TAG: String = "CHECK_RESPONSE"

api.getPosts().enqueue(object : Callback<List<Post>> {
    override fun onResponse(p0: Call<List<Post>>, p1: Response<List<Post>>) {
        if(p1.isSuccessful) {
            p1.body()?.let {
                for (post in it) {
                    Log.i(TAG, "onResponse: ${post.body}")
                }
            }
        }
    }

    override fun onFailure(p0: Call<List<Post>>, p1: Throwable) {
        Log.i(TAG, "onFailure: ${p1.message}")
    }
})
```

# Reminders and Announcements

## Assignments:

- **App Critiques (10%) - Deadline (Thursday 11:55 PM)**
- **Peer Engagement Weekly Task (2%)**

## Lab activities this week

- Develop three apps that connect to WebAPIs
  - Use Coil3 Library and Retrofit Library
  - Generate and use Google Gemini API Key

## Week 8

- *Introduction to coroutines and async tasks in Kotlin*
- *Structured concurrency basics*



# Reference

- Pari Delir Haghghi (S1 2024) Network Connection and Retrofit [PowerPoint slides], FIT5046: Mobile and Distributed Computing Systems, Monash University.
- Flaticon: <https://www.flaticon.com/>
- <https://medium.com/ibtech/activity-vs-fragment-703c749c1bbd>
- ChatGPT Image Generation - one week off

# Lab: API

## Overview

Welcome to Week 7 of FIT2081! This week, we'll explore how mobile applications communicate with external services through APIs (Application Programming Interfaces). This is a fundamental skill in modern app development, as very few apps exist in isolation – most need to fetch data from servers, update remote databases, or interact with third-party services.

The ability to connect to external resources transforms your mobile applications from isolated tools into connected platforms that deliver real-time information and services. In today's lab, you'll learn how to:

1. Connect to remote APIs
2. Process and display data from external sources
3. Implement different networking libraries
4. Handle asynchronous operations
5. Manage network errors gracefully

 These skills directly relate to your NutriTrack assignment, where you'll need to implement both the FruityVice API and Generative AI integration for the motivational messaging feature.

## Lab Learning Objectives

By the end of this lab, you will be able to:

1. **Configure** networking permissions and dependencies in Android applications
2. **Implement** API requests using different networking libraries (Coil, Retrofit, Google AI SDK)
3. **Process** API responses and display data in Jetpack Compose UI
4. **Handle** network errors and edge cases gracefully
5. **Apply** modern Android architecture patterns for network operations
6. **Integrate** third-party APIs to extend application functionality

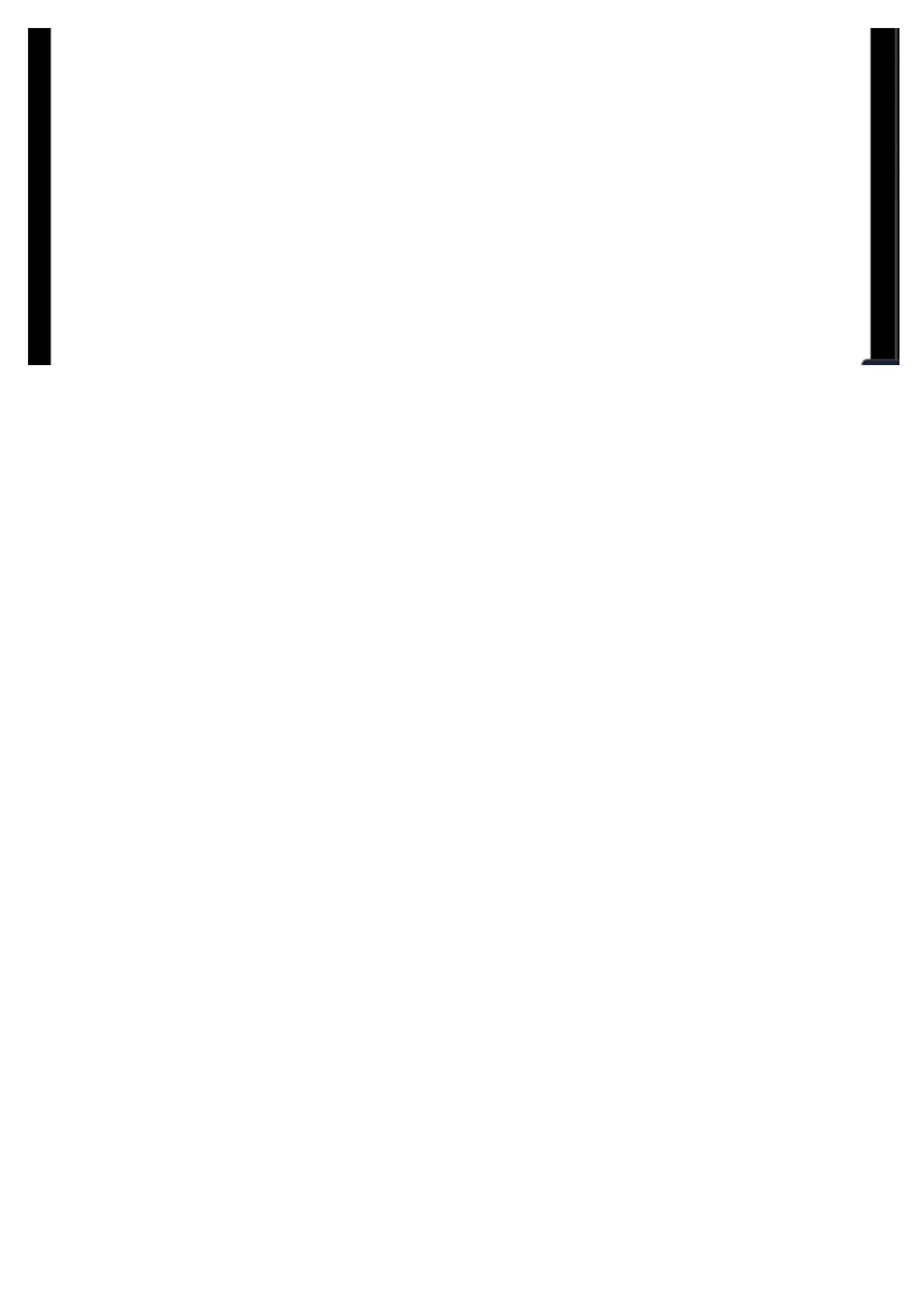
First, we are going to build our first internet-connected app! Take a sneak peak below at what we will build in Part 1.



Country Code

Get Flag





# Part 1a: Country Flag

## Learning Objectives

- Implement a simple API request to fetch images from a remote source
- Integrate the Coil3 image loading library
- Configure proper network permissions
- Handle basic error states in network requests
- Create efficient image loading with caching

## Introduction

Most mobile applications need to download and display images from remote servers. While you could implement this manually, image loading libraries like Coil provide efficient solutions with features like caching, placeholders, transformations, and error handling.

The Country Flag app demonstrates how to retrieve and display national flags using the country code entered by the user. This application:

1. Takes a two-letter country code as input (e.g., "us", "ca", "au")
2. Constructs a URL to the [flagcdn.com](https://flagcdn.com) API
3. Uses Coil3 to efficiently download and display the flag

## Real-World Application

This pattern is used in many applications:

- Travel apps displaying destination country information
- Language learning apps showing country flags for language selection
- International news apps indicating news sources
- Currency converters showing country flags beside currency codes
- E-commerce platforms indicating product origin

This app allows users to view national flags by entering a country code. It features a simple user interface with a text input field where users can enter a two-letter country code, a button to submit the request, and an image display area where the corresponding flag appears. The application uses the Coil3 library for efficient image loading and displays flags retrieved from the [flagcdn.com](https://flagcdn.com) API. The project follows modern Android development practices with a Material3 design system implementation, edge-to-edge UI support, and a clean separation of UI components. The architecture is relatively straightforward, focusing on a single screen experience built with Compose's declarative UI paradigm. The app requires internet permission to fetch flag images from the remote API. It's structured using Gradle with Kotlin DSL for build configuration and appears to be part of a learning or

demonstration project, possibly for educational purposes.

The objectives of this app are:

- Use a third-party library in an Android Application.
  - Generate a dynamic URL based on user input.
  - Access the internet from an Android Application.
- =====

## Country Flags Repository (<https://flagcdn.com/>)

[Flagcdn.com](https://flagcdn.com) is a free service created and maintained by [Flagpedia.net](https://Flagpedia.net) and hosted on Cloudflare that provides an API and content delivery network (CDN) specifically for national and regional flags. The service offers an easy solution for developers and website creators to embed high-quality flag images without hosting them locally. The platform includes all 254 country flags, 50 flags of U.S. states, and EU and UN flags, all sourced from vector files on Wikipedia Commons. Users can access these flags in multiple formats, including PNG, WebP, SVG (with optimized compression), and high-quality JPEG, making it suitable for various implementation needs. This is the service used in the country flag app, which constructs URLs like "[https://flagcdn.com/w640/\\$countryCode.jpg](https://flagcdn.com/w640/$countryCode.jpg)" to retrieve flag images based on the country code entered by the user. The service ensures all flags remain current and up-to-date, simplifying maintenance for developers who would otherwise need to manage their flag image collections.

## Coil3 Library

Coil3 is an advanced image loading library for Android applications and Compose Multiplatform, designed as a modern solution for efficient image handling. The acronym stands for "Coroutine Image Loader," highlighting its seamless integration with Kotlin Coroutines for asynchronous operations. Renowned for its performance optimization, Coil3 incorporates sophisticated memory and disk caching mechanisms, intelligent downsampling, and automatic request management to enhance application responsiveness while minimizing resource consumption. The library is remarkably lightweight, requiring only Kotlin, Coroutines, and Okio dependencies, while maintaining full compatibility with Google's R8 code shrinker for optimized APK sizes. In Jetpack Compose applications, Coil3's `AsyncImage` composable offers a clean, declarative approach to image loading with minimal boilerplate, as demonstrated in the country flag application which loads national flags from remote URLs. The library's Kotlin-first architecture prioritizes idiomatic code while maintaining excellent interoperability with contemporary Android development libraries including OkHttp, Ktor, and Material components. This makes Coil3 particularly valuable for modern Android development workflows focused on performance, maintainability, and sleek user experiences.

References: <https://coil-kt.github.io/coil/>

## Configure the App

## Step 1: Configure the App

1. Create a new Android project with an Empty Activity template
2. Add internet permission to `AndroidManifest.xml`:

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

Add Coil3 dependencies to your module-level `build.gradle.kts` file:

```
implementation("io.coil-kt.coil3:coil-compose:3.1.0")
implementation("io.coil-kt.coil3:coil-network-okhttp:3.1.0")
```

Click "Sync Now" to download the dependencies

**Why this matters:** The Internet permission is required for any app that needs to communicate with servers. Without it, network requests will fail with a security exception.

## Step 2: Create the UI Structure

Let's build a simple UI with a text field for country code input, a button to trigger the request, and an area to display the flag image.

In your `MainActivity.kt` update with the following content:

Let's add a text field and a button to fetch the required flag.

```
Column(
    modifier = modifier.fillMaxSize(),
    horizontalAlignment = Alignment.CenterHorizontally,
    verticalArrangement = Arrangement.Center
) {
    // Input field for the country code (e.g., "us", "ca", "au")
    TextField(
        value = countryCode,
        onValueChange = { countryCode = it },
        label = { Text("Country Code") },
        modifier = Modifier.padding(16.dp)
    )

    // Button to generate the flag URL and trigger image loading
    Button(
        onClick = { //generate the flag URL here
        },
        modifier = Modifier.padding(16.dp)
    ) {
        Text("Get Flag")
    }
}
```

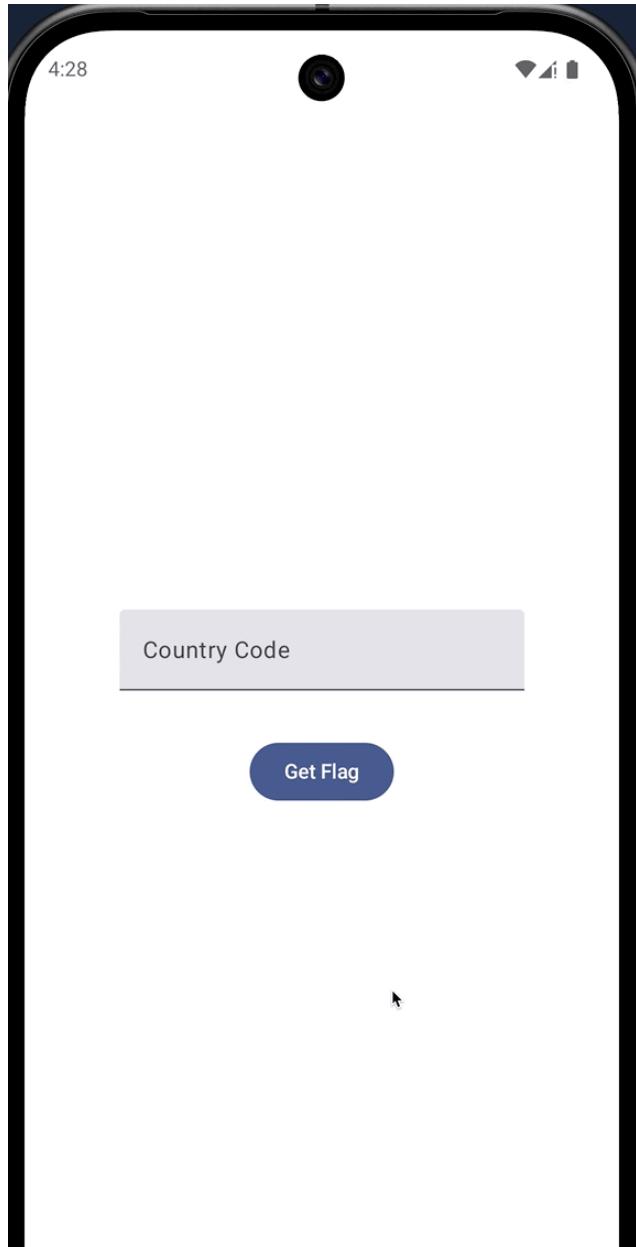
The button should take the country code provided in the text field and update a local mutable state.

```
Button(  
    onClick = {  
        // Construct URL for the flag image using the flagcdn.com API  
        imageUrl = "https://flagcdn.com/w640/$countryCode.jpg"  
    },  
    modifier = Modifier.padding(16.dp)  
) {  
    Text("Get Flag")  
}
```

Now, add the AsyncImage from the Coil3 library to fetch and show the required flag.

```
// Only show the image if a URL has been generated  
if(imageUrl.isNotEmpty()) {  
    // AsyncImage from Coil3 library handles image loading, caching and display  
    AsyncImage(  
        model = imageUrl,  
        contentDescription = "Country Flag",  
        modifier = Modifier.size(200.dp)  
    )  
}
```

Expected Output



## Final Code

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            Week7_country_flagTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    CountryFlagScreen(modifier = Modifier.padding(innerPadding))
                }
            }
        }
    }
}
```

```

@OptIn(ExperimentalMaterialsApi::class)
@Composable
fun CountryFlagScreen(modifier: Modifier = Modifier) {
    var countryCode by remember { mutableStateOf("") }
    var imageUrl by remember { mutableStateOf("") }

    Column(
        modifier = modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        TextField(
            value = countryCode,
            onValueChange = { countryCode = it },
            label = { Text("Country Code") },
            modifier = Modifier.padding(16.dp)
        )
        Button(
            onClick = { imageUrl = "https://flagcdn.com/w640/$countryCode.jpg" },
            modifier = Modifier.padding(16.dp)
        ) {
            Text("Get Flag")
        }
        Spacer(modifier = Modifier.height(16.dp))
        if(imageUrl.isNotEmpty()) {
            AsyncImage(
                model = imageUrl,
                contentDescription = "Country Flag",
                modifier = Modifier.size(200.dp)
            )
        }
    }
}

```

## Nots: How Coil3 Works Behind the Scenes

Coil3 (Coroutine Image Loader) handles several complex tasks for you:

- 1. Asynchronous Loading:** Uses Kotlin coroutines to load images without blocking the UI thread
- 2. Memory Caching:** Stores recently loaded images in memory to avoid repeated downloads
- 3. Disk Caching:** Saves images to disk for offline access
- 4. Request Deduplication:** Combines identical requests to minimize network usage
- 5. Lifecycle Awareness:** Cancels requests when components are destroyed
- 6. Error Handling:** Manages failed requests gracefully
- 7. Transformations:** Resize, crop, or otherwise modify images before display

This simplifies what would otherwise be hundreds of lines of custom code to handle these scenarios.

# Part 1b: Troubleshooting & Extensions

## Troubleshooting Common Issues

### 1. Image Not Loading:

- Check you have the Internet permission in your manifest
- Verify the country code is valid (2 letters)
- Check your device has internet connectivity
- Verify the URL format is correct

### 2. Slow Image Loading:

- Large images might take time to download
- Network speed can affect loading time
- First-time requests won't benefit from caching

### 3. Coil Library Not Found:

- Ensure you've added both Coil dependencies
- Make sure you've run Gradle sync after adding dependencies
- Check for typos in the import statements

## Extension Activity

### 1. Enhanced Input Validation:

- Add country code validation to ensure only valid ISO country codes are accepted
- Implement auto-completion for country codes as the user types
- Show a dropdown of country names to select from instead of typing codes

### 2. Advanced Error Handling:

- Implement retry functionality for failed image loads
- Add a placeholder image to display while loading
- Implement a fade-in animation when the image loads

### 3. Flag Information:

- Add a second API call to retrieve information about the country
- Display country name, population, and capital alongside the flag
- Implement a "Share" button to share the flag and country information

## Notes: Add Error Handling and Loading States

*Try doing this step by yourself to push your skills!*

Let's enhance our implementation to handle loading states and errors:

```
// Add these state variables at the top of the composable
var isLoading by remember { mutableStateOf(false) }
```

```
var loadError by remember { mutableStateOf(false) }
```

Update the `AsyncImage` element to handle loading and errors. Try looking at the documentation to do this yourself before looking at the hints here:

```
Box(  
    modifier = Modifier  
        .size(280.dp)  
        .padding(16.dp),  
    contentAlignment = Alignment.Center  
) {  
    AsyncImage(  
        model = imageUrl,  
        contentDescription = "Flag of $countryCode",  
        modifier = Modifier.fillMaxSize(),  
        onLoading = { isLoading = true },  
        onSuccess = {  
            isLoading = false  
            loadError = false  
        },  
        onError = {  
            isLoading = false  
            loadError = true  
        }  
    )  
  
    // Show loading indicator  
    if (isLoading) {  
        CircularProgressIndicator()  
    }  
  
    // Show error message  
    if (loadError) {  
        Column(  
            horizontalAlignment = Alignment.CenterHorizontally  
) {  
            Icon(  
                imageVector = androidx.compose.material.icons.Icons.Filled.Error,  
                contentDescription = "Error",  
                tint = MaterialTheme.colorScheme.error  
            )  
            Text(  
                text = "Could not load flag for '$countryCode'",  
                color = MaterialTheme.colorScheme.error  
            )  
        }  
    }  
}
```

Reflection

Take a moment to consider:

1. How does the Country Flag app demonstrate the separation of UI and data concerns?
2. What advantages does asynchronous image loading provide for user experience?
3. How could you apply similar techniques to load other types of media in your apps?
4. In what ways could the error handling be improved to provide a better user experience?

# Part 2a: Currency Exchange (Overview)

## Learning Objectives

- Implement a RESTful API client using Retrofit
- Configure API services and data models
- Apply the repository pattern for data operations
- Process JSON responses with GSON
- Implement coroutines for asynchronous network operations
- Create a clean separation between UI and data layers

## Introduction

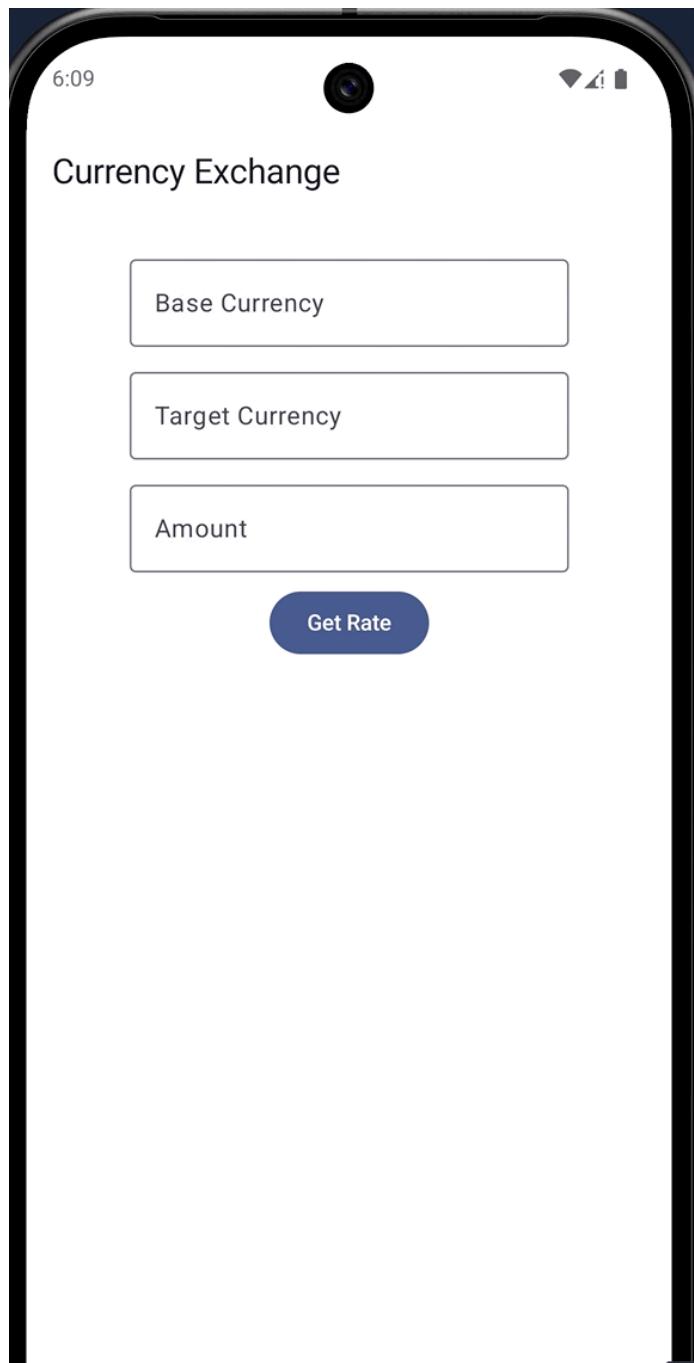
RESTful APIs are a standard way for applications to communicate with remote servers. Retrofit is a type-safe HTTP client for Android that simplifies API integration by converting HTTP API calls into Kotlin interface methods. In this section, we'll create a Currency Exchange app that fetches real-time exchange rates from the Frankfurter API.

## Real-World Application

This pattern is used in numerous applications:

- Banking apps for currency conversion
- Travel apps for displaying local prices
- Stock market and investment apps
- E-commerce platforms for international pricing
- Financial planning tools

This is the app we will be building:

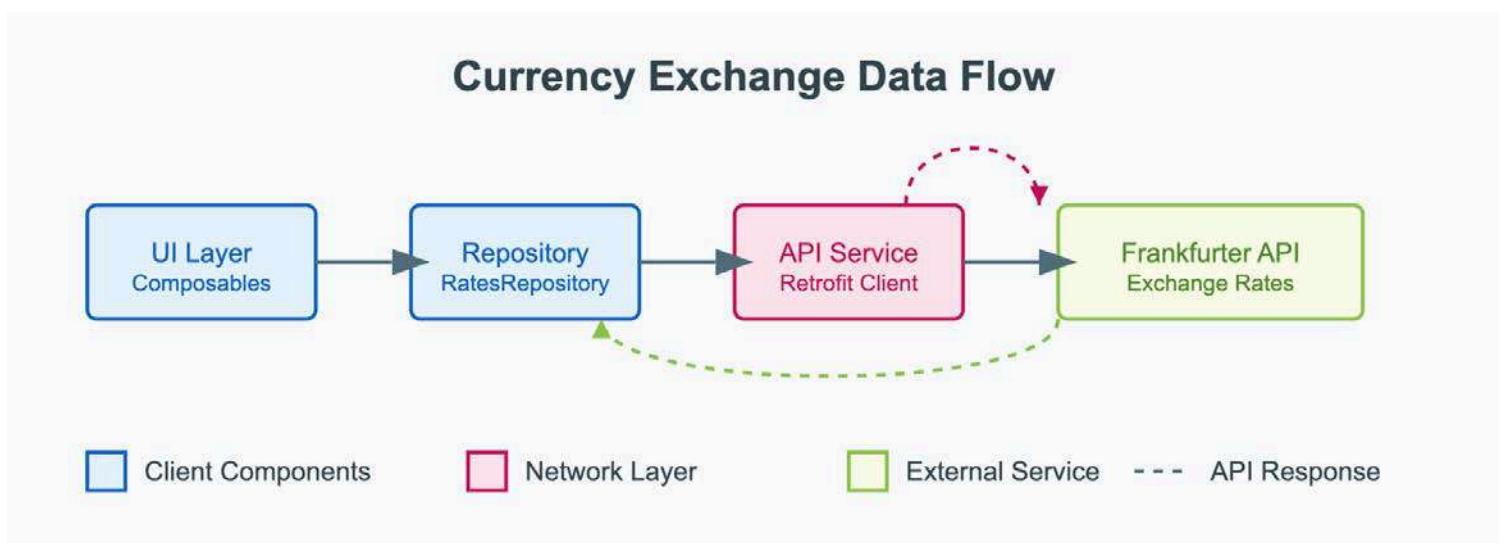


## Part 2b: Coding

**Note:** This application is a currency exchange converter that allows users to convert monetary values between different currencies using real-time exchange rates. The app features a simple user interface built with Jetpack Compose, where users can input a base currency (USD), a target currency (EUR), and the amount they want to convert.

When the user taps the "Get Rate" button, the app makes an asynchronous API call to the Frankfurter currency exchange rate service using Retrofit, retrieves the current exchange rate for the specified currencies, performs the conversion calculation, and displays the result on the screen with proper formatting. The application follows modern Android development practices with a clean architecture that separates network calls, data handling, and UI components.

## App Architecture



How could we represent this in pseudocode?

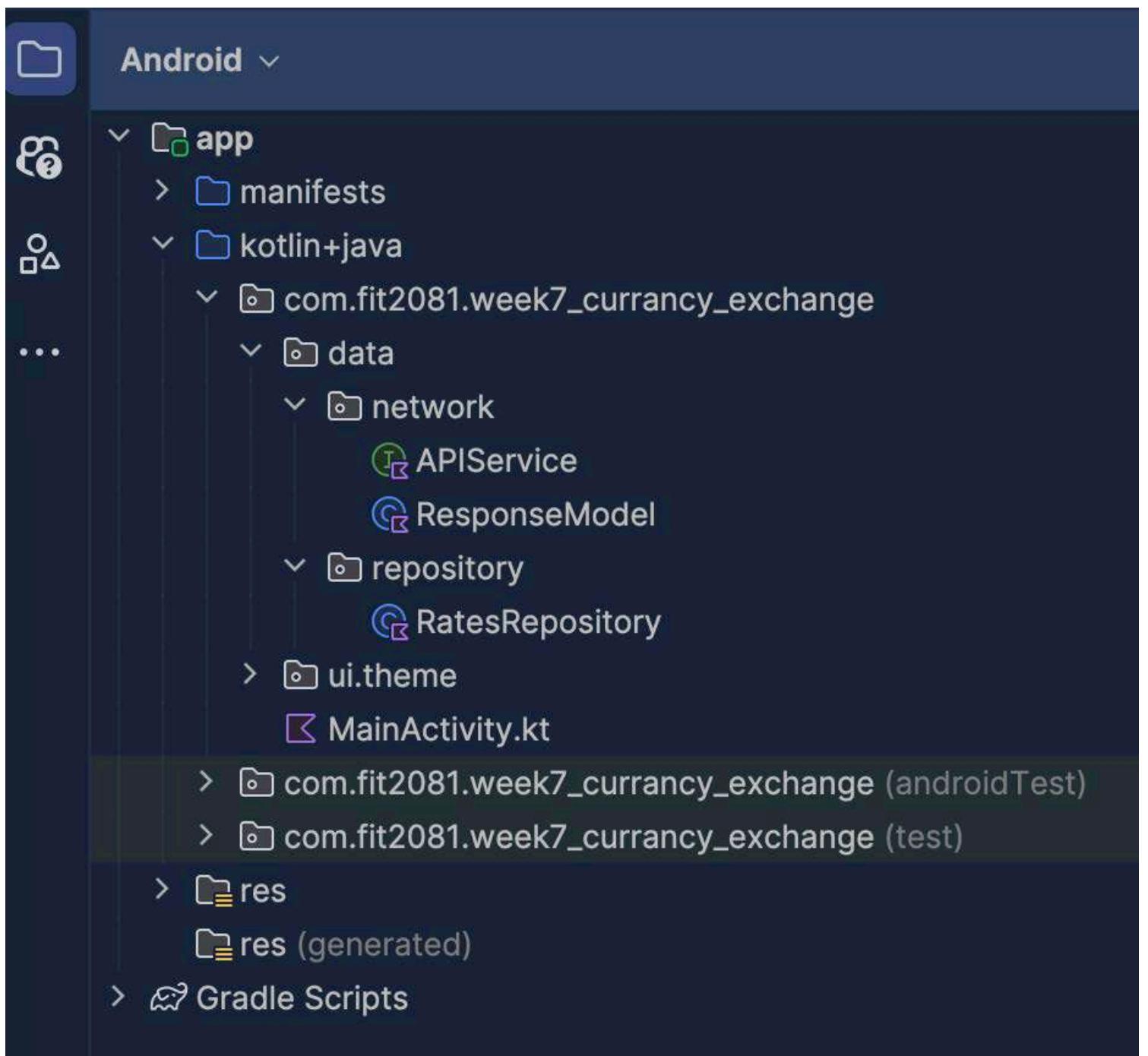
```
User Input → Button Click → CoroutineScope.launch →  
    Repository.getRate() → APIService.getRate() →  
        Retrofit converts to HTTP → Server processes request →  
            Server sends JSON response → Retrofit parses JSON →  
                ResponseModel object → Display result
```

The application follows a clean architecture design with three distinct layers. The UI layer is built using Jetpack Compose, featuring a simple interface with input fields for base currency, target currency, and amount, along with a "Get Rate" button and result display. The data layer consists of a repository pattern implemented in RatesRepository, which acts as an intermediary between the UI

and network operations. The network layer is handled by APIService, which defines API endpoints using Retrofit to communicate with the Frankfurter currency exchange API. When a user enters currency codes (like USD, EUR) and an amount, then taps the "Get Rate" button, the app launches a Kotlin coroutine that calls the repository, making an asynchronous API request. The response data, structured according to the ResponseModel class, is then used to calculate the converted amount, which is formatted and displayed to the user. This separation of concerns allows for maintainable code and efficient data flow throughout the application.

## App Objectives

- Fetch current exchange rates from the Frankfurter API service
- Calculate and display the converted amount with proper formatting
- Demonstrate modern Android development practices, including:
  - Clean architecture with separation of concerns
  - Jetpack Compose for UI implementation
  - Retrofit for network operations
  - Kotlin coroutines for asynchronous processing
  - Repository pattern for data handling



## Package Design Structure

The **network** package in this application is responsible for managing all API communication with the Frankfurter currency exchange service. It consists of two key components:

- `APIService.kt` - Defines the interface for API interactions using Retrofit:
  - Declares the endpoint for fetching exchange rates (`@GET("v1/latest")`).
  - Configures the Retrofit instance with base URL and JSON converter.
  - Provides a clean API for making network requests with typed parameters and responses.
- `ResponseModel.kt` - Contains the data class that models the API response:
  - Defines the structure with base (currency code) and rates (map of currency codes to exchange rates)

- Enables automatic JSON deserialization through Retrofit's converter.

This package implements a separation of concerns by isolating all network-related functionality from the rest of the application. It creates a clean abstraction that the repository layer can use without knowing implementation details about HTTP requests, headers, or response parsing. The network layer serves as the app's data source, providing the exchange rate information needed to perform currency conversions.

## Step 1: Configure the App

1. Create a new Android project or continue with the existing one
2. Add internet permission to `AndroidManifest.xml` (if not already added):

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

1. Add Retrofit and GSON dependencies to your module-level `build.gradle.kts` file:

```
// Retrofit
implementation("com.squareup.retrofit2:retrofit:2.9.0")
// Retrofit with Scalar Converter
implementation("com.squareup.retrofit2:converter-scalars:2.9.0")

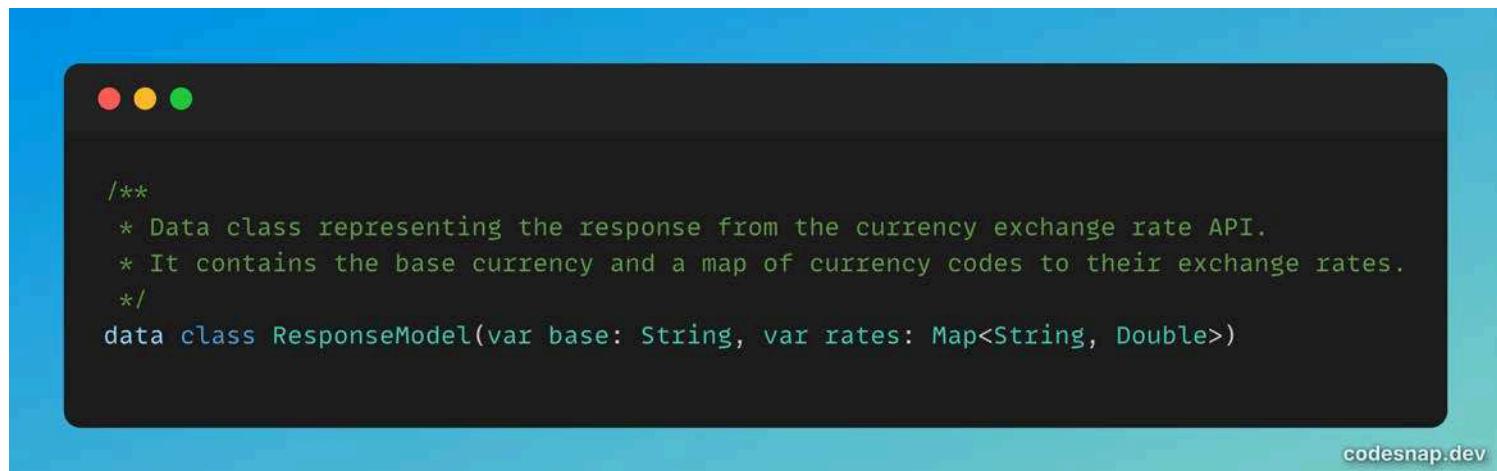
implementation ("com.squareup.retrofit2:converter-gson:2.9.0")
```

1. Click "Sync Now" to download the dependencies

## Step 2: Create the Model Package

First, let's create a data class to represent the API response. Under the **network** package, add the following file:

`ResponseModel.kt`



```
/**  
 * Data class representing the response from the currency exchange rate API.  
 * It contains the base currency and a map of currency codes to their exchange rates.  
 */  
data class ResponseModel(var base: String, var rates: Map<String, Double>)
```

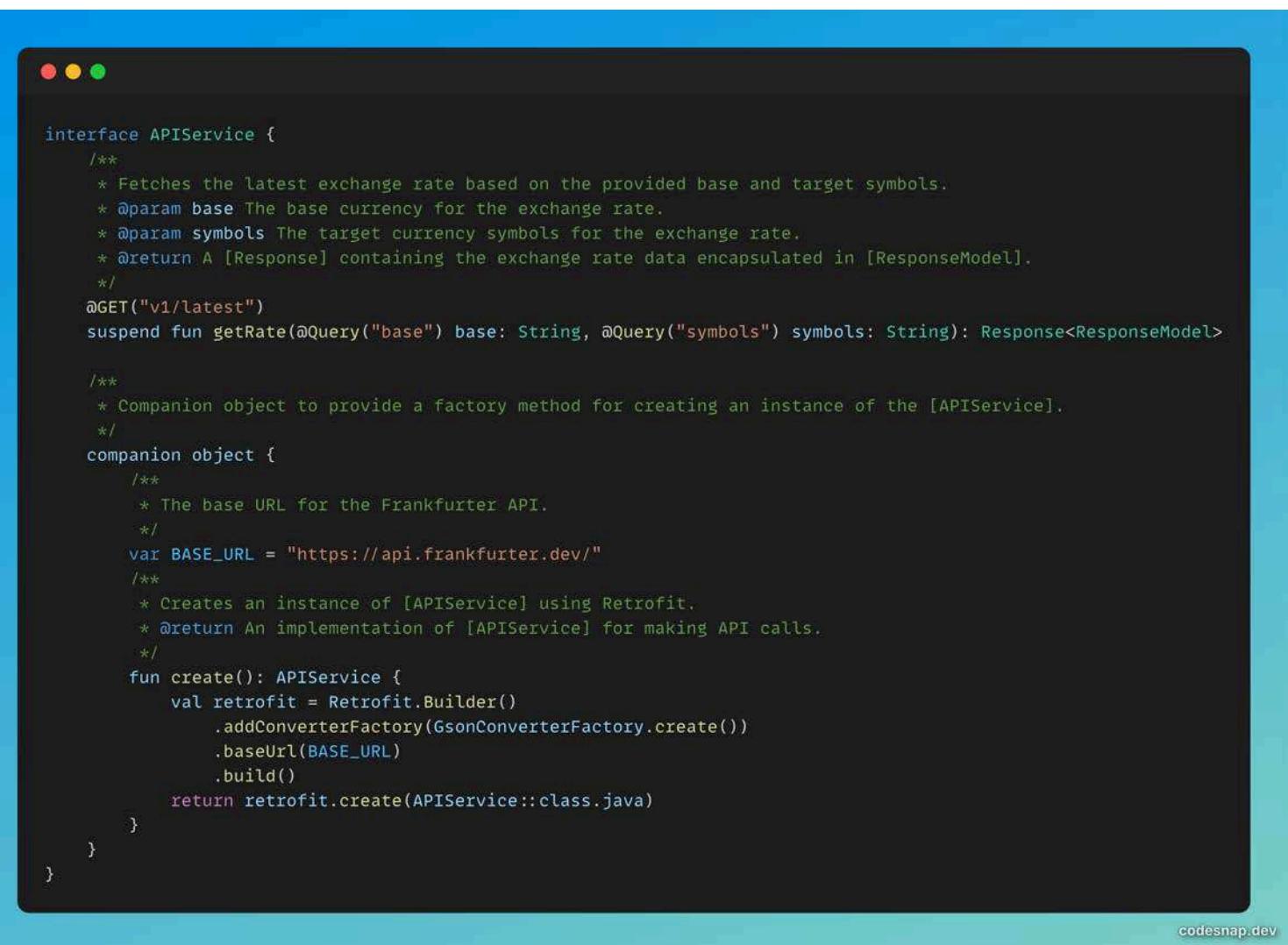
codesnap.dev

`ResponseModel.kt` serves as a data transfer object (DTO) that maps the JSON response from the Frankfurter API into a structured Kotlin object. It encapsulates the currency exchange data with two properties: the base currency and a map of currency codes to their exchange rates.

## Step 3: Create the Network Package

In the same package, add this next file:

`APIService.kt`:



```
interface APIService {
    /**
     * Fetches the latest exchange rate based on the provided base and target symbols.
     * @param base The base currency for the exchange rate.
     * @param symbols The target currency symbols for the exchange rate.
     * @return A [Response] containing the exchange rate data encapsulated in [ResponseModel].
     */
    @GET("v1/latest")
    suspend fun getRate(@Query("base") base: String, @Query("symbols") symbols: String): Response<ResponseModel>

    /**
     * Companion object to provide a factory method for creating an instance of the [APIService].
     */
    companion object {
        /**
         * The base URL for the Frankfurter API.
         */
        var BASE_URL = "https://api.frankfurter.dev/"

        /**
         * Creates an instance of [APIService] using Retrofit.
         * @return An implementation of [APIService] for making API calls.
         */
        fun create(): APIService {
            val retrofit = Retrofit.Builder()
                .addConverterFactory(GsonConverterFactory.create())
                .baseUrl(BASE_URL)
                .build()
            return retrofit.create(APIService::class.java)
        }
    }
}
```

codesnap.dev

This interface abstracts away the low-level HTTP communication details, providing a clean API that the repository layer can use to fetch currency exchange data.

Let's break down this interface:

- `@GET("v1/latest")` specifies the endpoint relative to the base URL
- The `getRate` method takes two parameters marked with `@Query`, which will be added as

query parameters to the URL

- The `suspend` keyword indicates this function should be called from a coroutine context
- `Response<ResponseModel>` is the return type, using Retrofit's `Response` wrapper
- The companion object provides a factory method to create the API service

## Step 4: Create the Repository Package

Now, let's create the repository that will handle data operations:

The repository package in this application is responsible for implementing the repository pattern, which serves as an abstraction layer between the data sources (network API) and the UI layer.

Specifically:

- Data Abstraction: RatesRepository hides the complexity of data operations from the UI layer, making the code more maintainable.
- Single Source of Truth: It provides a centralized point for obtaining exchange rate data.
- Separation of Concerns: The repository cleanly separates the business logic from network implementation details.
- API Communication: It handles interactions with the APIService to fetch currency exchange rates.
- Data Processing: Though minimal in this implementation, the repository could be expanded to handle caching, data transformations, or combining data from multiple sources.

The repository's structure allows the app to follow clean architecture principles by creating boundaries between different layers, making the codebase more testable and maintainable.

```
/*
 * Repository class for handling currency exchange rate data operations.
 * This class acts as an intermediary between the ViewModel and the API service.
 */
class RatesRepository() {
    // Create an instance of the API service for making network requests.
    private val apiService = APIService.create()

    /**
     * Fetches the currency exchange rate for a given base currency and target currency symbols.
     *
     * @param base The base currency code (e.g., "USD").
     * @param symbols The comma-separated list of target currency symbols (e.g., "EUR,GBP").
     * @return A ResponseModel containing the exchange rate data, or null if the request fails.
     */
    suspend fun getRate(base:String, symbols:String): ResponseModel? {
        return apiService.getRate(base,symbols).body();
    }
}
```

codesnap.dev

Psst: How would you add error handling here for BEST PRACTICE? Have a think and then see the spoiler:

▶ Expand

## Class MainActivity.kt

Let's create a screen for our currency exchange app: `CurrencyExchangeScreen.kt`

Firstly, let's create several variables (mutable states) to serve our app.



```
// Creates a repository instance for fetching exchange rates.  
var repository: RatesRepository = RatesRepository()  
// Remembers the user's input for the base currency.  
var baseCurrency by remember { mutableStateOf("") }  
// Remembers the user's input for the target currency.  
var targetCurrency by remember { mutableStateOf("") }  
// Remembers the amount to be converted.  
var amount by remember { mutableStateOf("") }  
// Remembers the result of the currency conversion.  
var result by remember { mutableStateOf("") }  
// Creates a coroutine scope tied to the composable's lifecycle.  
val coroutineScope = rememberCoroutineScope()
```

codesnap.dev

Now, create the UI elements:

```
Column(  
    modifier = Modifier  
        .padding(innerPadding)  
        .padding(16.dp) // Adds padding inside the column.  
        .fillMaxWidth(), // Makes the column take the full width of the screen.  
    horizontalAlignment = Alignment.CenterHorizontally, // Centers content horizontally.  
    verticalArrangement = Arrangement.spacedBy(8.dp) // Adds vertical spacing between items.  
) {  
    OutlinedTextField(  
        value = baseCurrency, // Displays the current base currency.  
        onValueChange = { baseCurrency = it }, // Updates the base currency when the user types.  
        label = { Text("Base Currency") } // Label for the text field.  
)  
    OutlinedTextField(  
        value = targetCurrency, // Displays the current target currency.  
        // Updates the target currency when the user types.  
        onValueChange = { targetCurrency = it },  
        label = { Text("Target Currency") } // Label for the text field.  
)  
    OutlinedTextField(  
        value = amount, // Displays the current amount to be converted.  
        onValueChange = { amount = it }, // Updates the amount when the user types.  
        label = { Text("Amount") } // Label for the text field.  
)  
    Button(  
        onClick = {  
            //  
        })  
    {  
        Text("Get Rate") // Button text.  
    }  
    if (result.isNotEmpty()) {  
        // Displays the conversion result if it's not empty.  
        Text(  
            text = "Result: $result $targetCurrency",  
            fontSize = 24.sp,  
            fontWeight = FontWeight.Bold  
        )  
    }  
}
```

codesnap.dev

Inside the onClick method, call the function getRate from the repository class.

```
onClick = {
    coroutineScope.launch {
        // Launches a coroutine to fetch the exchange rate.
        val rate = repository.getRate(baseCurrency, targetCurrency)?.rates?.get(targetCurrency)
        // Fetches the exchange rate from the repository.
        if (rate != null) {
            // Calculates the converted amount and updates the result.
            result = String.format("%.2f", (amount.toDouble() * rate))
        }
    }
}
```

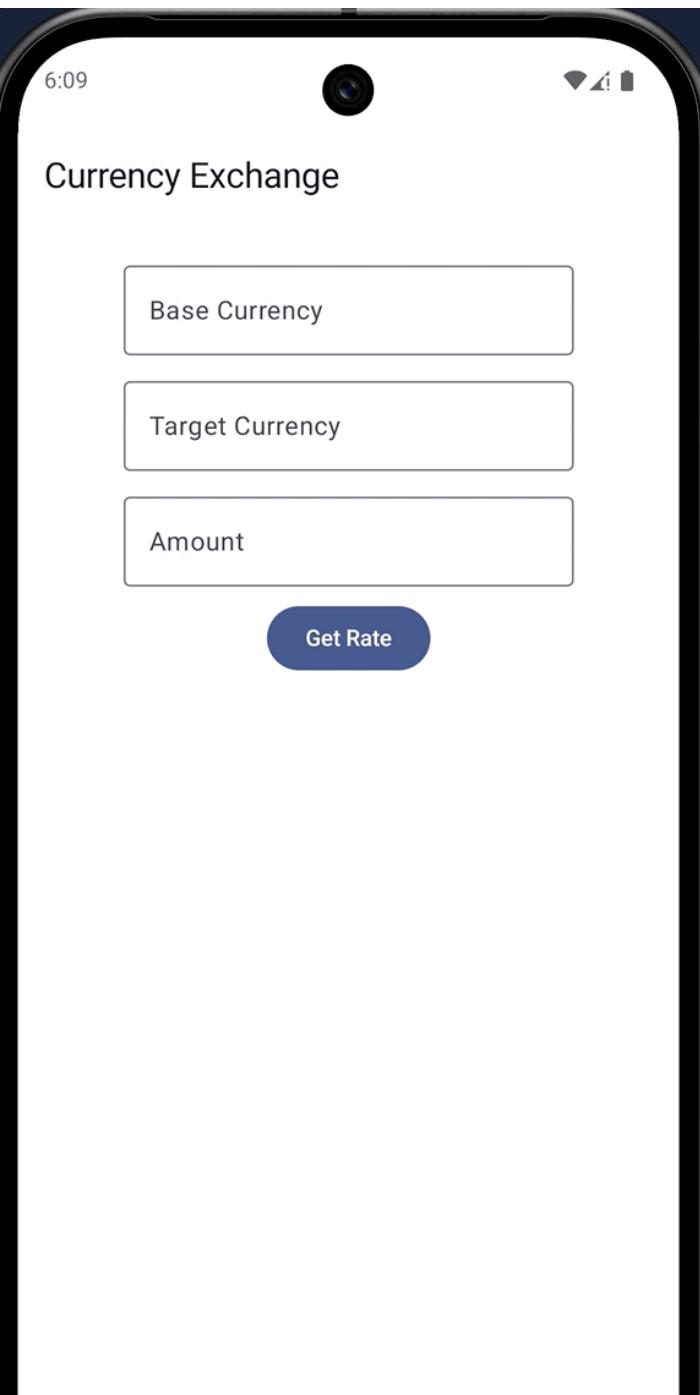
codesnap.dev

## How Retrofit Works Behind the Scenes

Retrofit simplifies API integration through several key mechanisms:

1. **Interface-Based API Definition:** Converts HTTP API endpoints into interface methods
2. **Annotation Processing:** Uses annotations like `@GET`, `@POST`, `@Query` to build HTTP requests
3. **Type-Safe Converters:** Automatically converts between JSON and Kotlin objects using GSON
4. **Dynamic Proxy Implementation:** Creates implementations of your interface at runtime
5. **OkHttp Integration:** Leverages OkHttp for efficient HTTP requests with features like connection pooling
6. **Coroutine Support:** Native support for suspension functions for asynchronous operations

## Expected Output



# Part 2c: Troubleshooting and Extensions

## Troubleshooting Common Issues

### 1. API Request Failing:

- Check internet permission in manifest
- Verify base URL is correct (<https://api.frankfurter.dev>)
- Ensure currency codes are valid (3-letter ISO codes)
- Add logging interceptor to see detailed request/response

### 2. JSON Parsing Errors:

- Ensure your data class properties match the JSON field names
- Check that property types match the JSON types
- Consider adding @SerializedName annotations for non-matching fields

### 3. Coroutine Context Issues:

- Ensure API calls are made within a coroutine scope
- Use appropriate dispatchers (IO for network operations)
- Handle exceptions properly to prevent coroutine crashes

## Extension Activity

### 1. Enhanced Currency Converter:

- Add a dropdown to select from common currencies
- Implement historical rate lookup with date selection
- Show conversion trends in a chart

### 2. Caching Exchange Rates:

- Implement a local database to cache exchange rates
- Add timestamp to track when rates were last updated
- Provide offline functionality with cached rates

### 3. Multi-Currency Conversion:

- Allow users to convert to multiple target currencies simultaneously
- Display a list of popular conversion results
- Implement a favorite currencies feature

---

## Part 3a: ADVANCED - GenAI (Overview)

### Learning Objectives

- Integrate a modern AI API into a mobile application
- Implement API key security practices
- Create a responsive UI for AI interactions
- Manage state during asynchronous AI operations
- Apply the MVVM architecture pattern

### Introduction

Generative AI is transforming mobile applications by enabling natural language understanding, content generation, and intelligent interactions. In this section, we'll integrate Google's Gemini API to create a simple AI chat interface.

### Real-World Application

This pattern is used in numerous applications:

- Virtual assistants in productivity apps
- Writing assistants in note-taking apps
- Smart replies in messaging platforms
- Content generation in creative tools
- Personalized recommendations in lifestyle apps
- Chatbots in customer service applications

## Part 3b: Coding

Generate Google Gemini API Key (using personal email)

- Visit this site: <https://ai.google.dev/gemini-api/docs/api-key>
- Log in with your personal (not Monash) credentials
- Click on "Get a Gemini API key in Google AI Studio"
- Click on "Create API Key"
- From projects, select Gemini API, then click on "Create API Key"



Monash has turned off the access to Google AI Studio.



If you encounter difficulties, use the following API Key temporarily:

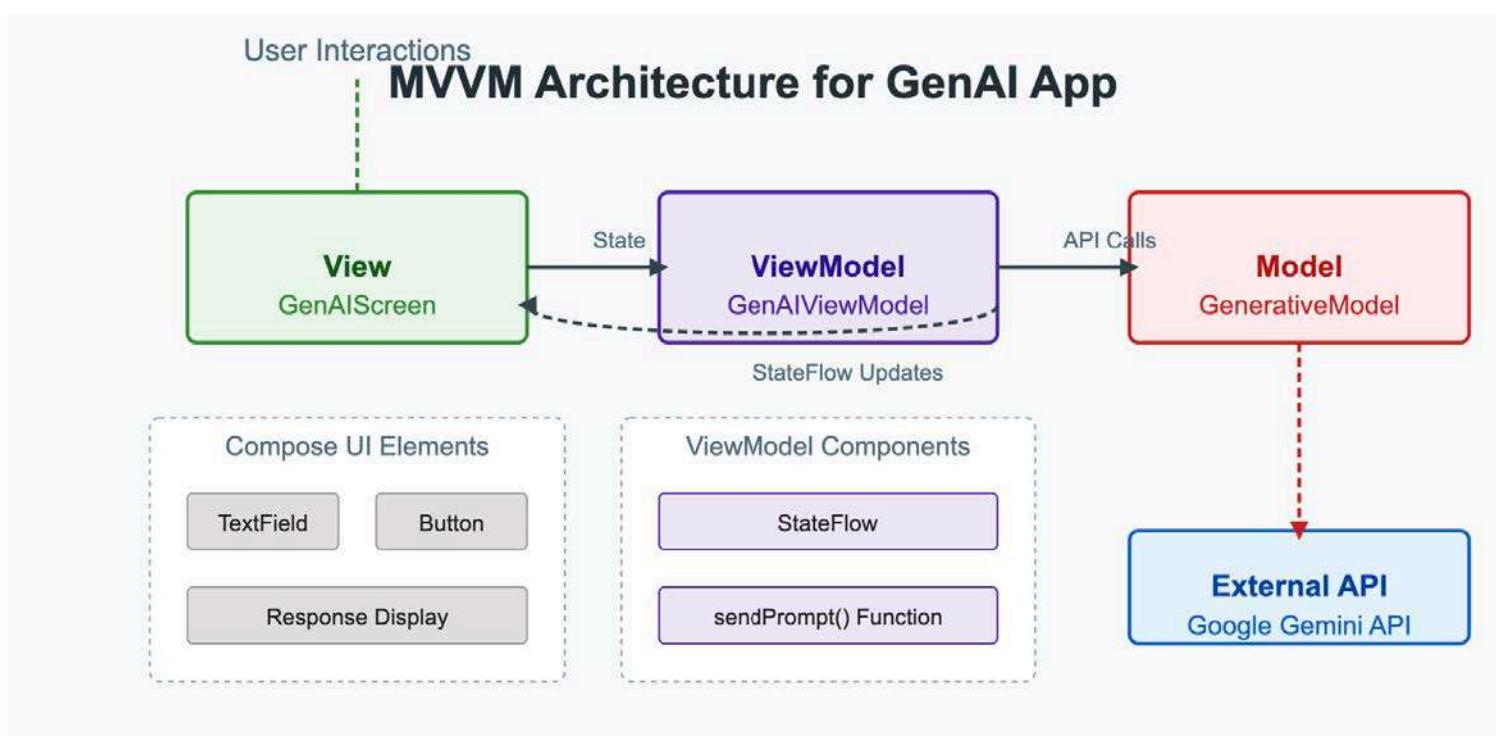
AlzaSyDrzfHdeeoZK9qG1DCB07Bb6qsqOw469Ps

## Configure the App

Open the local.properties file and add the API key you created earlier in this format

```
apiKey=YOUR_API_KEY_Goes_Here
```

## App Structure



This application demonstrates a simple implementation of a generative AI interface using the Gemini 1.5 Flash model. The project follows the MVVM (Model-View-ViewModel) architecture pattern, clearly separating concerns across its components. The codebase is structured with MainActivity as the entry point, which hosts the GenAISScreen Composable UI element. UI interactions are handled by the GenAIViewModel, which manages state through a StateFlow and communicates with the Google Generative AI API. The application uses Jetpack Compose for its modern declarative UI, with state management implemented through MutableStateFlow and collectAsState. Depending on the interaction status, the UI displays different states (Initial, Loading, Success, Error). The project includes proper error handling and follows Android development best practices with coroutines for asynchronous operations, keeping UI interactions responsive while network operations happen in the background.



## UiState Interface

Using a sealed interface hierarchy, the `UiState.kt` file implements a core state management pattern in this app. It defines four distinct states that represent the complete lifecycle of text generation:

- Initial for the app's starting state,
- Loading during content generation,
- Success containing the generated output text,
- and Error with appropriate error messages when generation fails.

This structured approach to state management follows modern Android development practices by providing type-safe state transitions and enabling the UI layer to react appropriately to each state. The GenAIViewModel uses this sealed interface to emit different states through a StateFlow, which the Compose UI observes and renders accordingly. By centralizing state definitions in this file, the application achieves a clear separation between state representation and the components that produce or consume these states, making the codebase more maintainable and testable.

```
sealed interface UiState {  
  
    /**  
     * Empty state when the screen is first shown  
     */  
    object Initial : UiState  
  
    /**  
     * Still loading  
     */  
    object Loading : UiState  
  
    /**  
     * Text has been generated  
     */  
    data class Success(val outputText: String) : UiState  
  
    /**  
     * There was an error generating text  
     */  
    data class Error(val errorMessage: String) : UiState  
}
```

codesnap.dev

## GenAIViewModel ViewModel Class

The GenAIViewModel.kt file represents a critical component in this app's MVVM architecture, serving as the intermediary between the UI and the Gemini AI service. This ViewModel manages the entire state lifecycle of AI text generation through a StateFlow that emits different UiState instances (Initial, Loading, Success, Error), keeping the UI layer informed about the current operation status.

It encapsulates the GenerativeModel configuration, setting up the "gemini-1.5-flash" model with the appropriate API key. The class handles all asynchronous operations through Kotlin coroutines launched within the viewModelScope, ensuring that network requests occur on the IO dispatcher

while maintaining proper lifecycle awareness. The main functionality is exposed through the `sendPrompt()` method, which processes user input, manages state transitions, performs error handling, and delivers results back to the UI. This implementation demonstrates good separation of concerns, as the `ViewModel` contains no direct UI references while providing a clean API for the composable UI elements to consume.

```
/*
 * ViewModel class for handling interactions with a generative AI model.
 * This class manages the UI state and communicates with the GenerativeModel
 * to generate content based on user prompts.
 */
class GenAIVViewModel : ViewModel() {

    /**
     * Mutable state flow to hold the current UI state.
     * Initially set to `UiState.Initial`.
     */
    private val _uiState: MutableStateFlow<UiState> =
        MutableStateFlow(UiState.Initial)

    /**
     * Publicly exposed immutable state flow for observing the UI state.
     */
    val uiState: StateFlow<UiState> =
        _uiState.asStateFlow()

    /**
     * Instance of the GenerativeModel used to generate content.
     * The model is initialized with a specific model name and API key.
     */
    private val generativeModel = GenerativeModel(
        modelName = "gemini-1.5-flash",
        apiKey = BuildConfig.apiKey
    )

    /**
     * Sends a prompt to the generative AI model and updates the UI state
     * based on the response.
     *
     * @param prompt The input text prompt to be sent to the generative model.
     */
    fun sendPrompt(
        prompt: String
    ) {
        // Set the UI state to Loading before making the API call.
        _uiState.value = UiState.Loading

        // Launch a coroutine in the IO dispatcher to perform the API call.
        viewModelScope.launch(Dispatchers.IO) {
            try {
                // Generate content using the generative model.
                val response = generativeModel.generateContent(
                    prompt
                )
                // Update the UI state with the generated content.
                _uiState.value = UiState.Content(response)
            } catch (e: Exception) {
                // Handle the error, e.g., show an error message or fallback.
                _uiState.value = UiState.Error(e.message)
            }
        }
    }
}
```

```
    val response = generateContentOrGenerateContent()
    content {
        text(prompt) // Set the input text for the model.
    }
    // Update the UI state with the generated content if successful.
    response.text?.let { outputContent ->
        _uiState.value = UiState.Success(outputContent)
    }
} catch (e: Exception) {
    // Update the UI state with an error message if an exception occurs.
    _uiState.value = UiState.Error(e.localizedMessage ?: "")
}
}
}
}
```

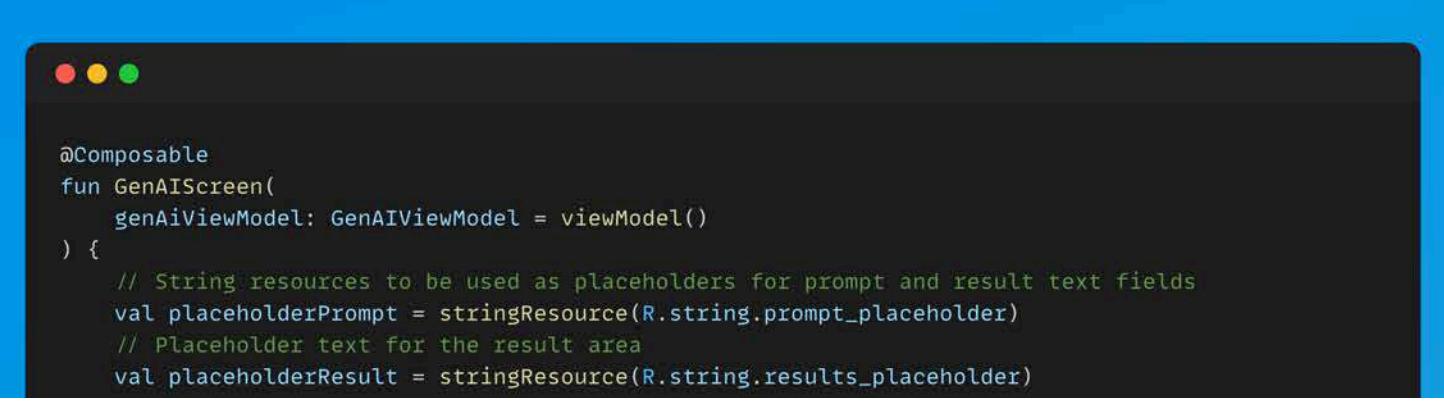
codesnap.dev



If you get the error "Unresolved Reference apiKey", try to replace the BuildConfig statement with:  
apiKey="AlzaSyDrzfHdeeoZK9qG1DCB07Bb6qsqOw469Ps"

## GenAISScreen.kt class

The GenAISScreen.kt file implements the main user interface component. It contains a single composable function that serves as the central UI hub for user interactions, utilizing modern declarative UI patterns. The screen presents a simple yet functional interface with a text input field for entering prompts, a submission button, and a response area that adapts based on the current UI state. It leverages the GenAIViewModel to manage business logic and state, observing state changes through Flow collection and responding accordingly by showing a loading indicator, error messages, or successful AI-generated content. The component demonstrates effective state management using rememberSaveable to preserve user input across configuration changes, proper resource management through string references, and responsive layout design with Compose modifiers. The implementation also follows accessibility best practices with appropriate text styling and color contrast for different UI states, creating a cohesive user experience that effectively bridges the UI layer with the underlying AI functionality.



The screenshot shows the GenAISScreen.kt code in an Android Studio editor. The code is a Composable function named GenAISScreen that takes a GenAIViewModel parameter. It uses string resources for placeholders and initializes the placeholderPrompt and placeholderResult variables. The code is displayed in a light theme with syntax highlighting for keywords and comments.

```
@Composable
fun GenAISScreen(
    genAiViewModel: GenAIViewModel = viewModel()
) {
    // String resources to be used as placeholders for prompt and result text fields
    val placeholderPrompt = stringResource(R.string.prompt_placeholder)
    // Placeholder text for the result area
    val placeholderResult = stringResource(R.string.results_placeholder)
```

```
// State to hold user input prompt, persisted across configuration changes
var prompt by rememberSaveable { mutableStateOf(placeholderPrompt) }

// State to hold the result text to display, persisted across configuration changes
var result by rememberSaveable { mutableStateOf(placeholderResult) }

// Observe the UI state from the ViewModel as a state that triggers recomposition when changed
val uiState by genAiViewModel.uiState.collectAsState()

// Get the current Android context
val context = LocalContext.current

/**
 * Root column that contains all UI elements arranged vertically
 */
Column(
    modifier = Modifier.fillMaxSize()
) {
    /**
     * App title text component
     */
    Text(
        text = stringResource(R.string.app_title),
        style = MaterialTheme.typography.titleLarge,
        modifier = Modifier.padding(16.dp)
    )

    /**
     * Row containing the prompt input field and submit button
     */
    Row(
        modifier = Modifier.padding(all = 16.dp)
    ) {
        /**
         * Text input field for entering the AI prompt
         */
        TextField(
            value = prompt,
            label = { Text(stringResource(R.string.label_prompt)) },
            onValueChange = { prompt = it },
            modifier = Modifier
                .weight(0.8f)
                .padding(end = 16.dp)
                .align(Alignment.CenterVertically)
        )

        /**
         * Button to submit the prompt to the AI model
         * Disabled when prompt is empty
         */
        Button(
            onClick = {
                // Send the current prompt text to the ViewModel for processing
                genAiViewModel.sendPrompt(prompt)
            },
            enabled = prompt.isNotEmpty(),
            modifier = Modifier
                .align(Alignment.CenterVertically)
        ) {
            Text(text = stringResource(R.string.action_go))
        }
    }

    /**
     * Conditional UI rendering based on the current UI state
     */
}
```

```
/*
if (uiState is UiState.Loading) {
    // Display loading indicator when content is being generated
    CircularProgressIndicator(modifier = Modifier.align(Alignment.CenterHorizontally))
} else {
    // Initialize text color with default value
    var textColor = MaterialTheme.colorScheme.onSurface

    // Handle error state
    if (uiState is UiState.Error) {
        // Use error color for error messages
        textColor = MaterialTheme.colorScheme.error
        // Update result with the error message
        result = (uiState as UiState.Error).errorMessage
    }
    // Handle success state
    else if (uiState is UiState.Success) {
        // Use default text color for success
        textColor = MaterialTheme.colorScheme.onSurface
        // Update result with the generated content
        result = (uiState as UiState.Success).outputText
    }
}

// Create a scrollable state for the text display area
val scrollState = rememberScrollState()

/**
 * Text component to display either the AI-generated content or error messages
 * Supports scrolling for long content
 */
Text(
    text = result,
    textAlign = TextAlign.Start,
    color = textColor,
    modifier = Modifier
        .align(Alignment.CenterHorizontally)
        .padding(16.dp)
        .fillMaxSize()
        .verticalScroll(scrollState)
)
}
}
}
```

## Expected Output

FIT2081 GenAI



Prompt

Go

(Results will appear here)

## Part 3c: Troubleshooting

### Troubleshooting: unresolved reference ai/GenerativeModel

Check these:

```
// In build.gradle.kts
```

```
android {  
    //some code  
    buildFeatures {  
        compose = true  
        buildConfig = true  
    }  
}  
  
dependencies {  
    //some code  
    implementation(libs.generativeai)  
    //some code  
}
```

```
// In lib.versions.toml
```

```
[versions]  
generativeai = "0.9.0"  
  
[libraries]  
generativeai = { group = "com.google.ai.client.generativeai", name = "generativeai", version.ref =
```

At last, rebuild it.

### Troubleshooting: API Key Not Showing Up from local.properties

If you've added your API key to `local.properties` but it still isn't showing up in your app, here's a step-by-step guide to help you debug it.

#### Step 1: Confirm `local.properties` is Correct

Make sure you've added your key to the bottom of the `local.properties` file like this:

```
GENAI_API_KEY="your-actual-api-key"
```

Make sure there are **no spaces** before or after the `=`.

#### ✗ Step 2: You're Probably Importing the WRONG `BuildConfig`

This is one of the most common mistakes:

#### ✗ Wrong Import:

```
import com.google.ai.client.generativeai.BuildConfig
```

#### Correct Import:

```
import com.yourpackagename.BuildConfig
```

If you use the wrong `BuildConfig`, you're importing a library's config — **not your app's**, which is why your key won't show up.

## Step 4: Clean and Rebuild the Project

Sometimes Android Studio gets grumpy. Try:

- **Build > Clean Project**
- **Build > Rebuild Project**
- Then try running again.

## Still Not Working?

- Double-check the package name and import.
- Make sure you **re-synced Gradle** after changing anything in `build.gradle.kts`.
- If using multiple modules, ensure you're referencing the correct `BuildConfig` from the **app module**.

# FIT2081 Week 8

## Coroutines & Async

## Programming

# Week 8 - Lecture Reading Materials

---

## Introduction

Welcome to our lecture on Coroutines & Async Programming in Android development. Last week, we explored the Networking & APIs. Today, we'll learn how to efficiently make data operations (CRUD) in different sources (Room Database and APIs) by using Kotlin Coroutines.

Consider our previous fitness app example: when the system wants to search for a user profile saved in the Room Database, it might take seconds to iterate through the database. The same thing happens if users access online information, such as downloading food recipes. They may have a slow internet connection, which causes some delays in receiving the information. As a developer, you want to have the data input/output (IO) operations in parallel with normal user interaction, such as showing a loading bar/circle when waiting. Then, the UI interactions need to be performed **asynchronously** with the IO operations.

**Kotlin Coroutines** are a feature in Kotlin that provides a **lightweight** way of managing **concurrency** and **asynchronous programming**. Unlike traditional threads, coroutines are more efficient, consume fewer resources, and are easier to manage. They allow you to write non-blocking code and perform background tasks **without blocking the main thread** (UI thread), ensuring your app remains responsive.

## Learning Objectives

By the end of this lecture, you will be able to:

- understand concurrency basics for mobile app development;
- understand coroutines and async tasks in Kotlin;
- learn the best practices for network calls;
- and practice network calls in Compose projects.

# Concurrency vs Parallelism

## Concurrency

- Refers to the ability of a system to handle multiple tasks at the same time, but not necessarily executing them simultaneously.
- It's about dealing with multiple tasks in an overlapping period, potentially switching between them. The tasks might be executed on a single processor or core.
- Concurrency is useful for systems that need to manage lots of I/O-bound tasks, like reading from files or waiting for network responses.

## Parallelism

- Refers to the simultaneous execution of multiple tasks, usually on multiple processors or cores.
- It's about splitting a task into smaller sub-tasks that can be run simultaneously, improving performance for compute-heavy operations.
- Parallelism is useful for CPU-bound tasks, where a problem can be divided into independent sub-problems that can be processed at the same time.

## Key Difference

- **Concurrency** is about handling multiple tasks at once but not necessarily at the same time.
  - multitasking (tasks run "in parallel" in an interleaved way but not simultaneously).
- **Parallelism** is about performing multiple tasks at the same time, usually to speed up computation.
  - simultaneous execution (tasks run at the same time on multiple cores).

# Multithreading vs Multiprocessing

## Multithreading

- Running multiple threads within the same process. All threads share the same memory space. Imagine you are handling multiple tasks on the same computer. Each task runs in the same environment, sharing resources (like memory, files, etc.).
- Suitable for I/O-bound tasks like network requests or file reading.
- If multiple threads need access to shared resources, conflicts or inconsistencies may arise (requiring locking mechanisms).

## Multiprocessing

- Running multiple independent processes, each with its own memory space. Imagine you are performing tasks on multiple computers. Each computer has its own resources, and they don't interfere with each other.
- Suitable for CPU-bound tasks like complex computations. Each process can run on a separate core, fully utilizing multi-core processors.
- Inter-process communication is more complex, and creating processes can be more expensive.

## Key Difference

- **Multithreading:** Runs in the same process with shared memory space, ideal for I/O-bound tasks.
- **Multiprocessing:** Each process has its own memory space, suitable for CPU-bound tasks, and can take full advantage of multi-core CPUs.

# Android Main (UI) Thread

## What is a Thread?

A thread is a sequence of instructions that can be executed independently. In Android, all components (e.g., Activities, Services, Broadcast Receivers) run on a thread.

## Threads in Android

- Android uses a **single-threaded model**, where all interactions with the UI are done on the Main (UI) Thread. Other tasks like background processing, network calls, or I/O operations are offloaded to separate threads to ensure the UI remains responsive.
- By default, Android applications have a Main Thread (UI Thread), which is responsible for interacting with the user interface and handling events like button clicks and gestures.

## Why Use Multiple Threads?

Using multiple threads allows Android apps to continue responding to user input while performing heavy tasks in the background.

## What is the Main (UI) Thread?

The **Main Thread** (UI Thread) is the primary thread responsible for rendering UI elements, processing user input, and interacting with Android components like Activities and Services.

- The UI Thread is responsible for:
  - Drawing UI components (e.g., TextViews, Buttons, etc.).
  - Handling UI interactions (e.g., click events, gestures).
  - Updating the screen in response to user interactions....
- The Main Thread runs an **event loop**, constantly checking for new messages or events (e.g., touch events, button clicks) and handling them.
- **Why the Main Thread is Critical**
  - Smooth UI Performance
  - Application Not Responding Error
  - Thread Synchronization

# How to Offload Work from the Main Thread

- **Using Background Threads**
  - **AsyncTask** (deprecated in API level 30, but still useful for understanding background tasks):
    - Executes background tasks asynchronously and provides a simple way to update the UI thread after the task is complete.
    - Typically used for short background operations that update the UI.
  - **Thread**:
    - The most basic way to run a task in the background.
    - You create a new `Thread` and run the task inside the `run()` method.
  - **HandlerThread**:
    - A specialized subclass of `Thread` that allows you to create a thread with a **message queue** for handling events and messages.
- **ExecutorService**:
  - A more efficient and flexible way to manage background tasks.
  - `ExecutorService` can manage multiple threads and is particularly useful for managing a pool of threads for long-running tasks.

# Communicating Back to the UI Thread

- **Updating UI from a Background Thread**
  - You cannot directly update UI elements from a background thread. To communicate with the Main Thread, use:
    - **Handler**: You can post messages from a background thread to the Main Thread using a `Handler`.
    - **runOnUiThread()**: A method in `Activity` to execute code on the Main Thread.
  - **LiveData** and **ViewModel**:
    - Modern Android development encourages the use of `LiveData` and `ViewModel` for managing UI updates safely.
    - `LiveData` ensures that data changes are observed and reflected on the UI.
    - `ViewModel` helps persist UI-related data across configuration changes.
- **Coroutines (Kotlin)**:
  - With Kotlin, **Coroutines** provide a simpler way to manage background tasks. You can use `launch` and `async` to run background tasks and then update the UI using the `Dispatchers.Main` context.

# Kotlin Coroutines

**Kotlin Coroutines** are a feature in Kotlin that provides a lightweight way of managing concurrency and asynchronous programming. Unlike traditional threads, coroutines are more efficient, consume fewer resources, and are easier to manage. They allow you to write non-blocking code and perform background tasks without blocking the main thread (UI thread), ensuring your app remains responsive.

## Why do we need Kotlin Coroutines?

- **Avoid Blocking the UI Thread:** In Android, UI updates must happen on the main thread. If you perform long-running tasks, like network requests or database queries, on the main thread, it will block the UI and make the app unresponsive. Coroutines let you perform such tasks asynchronously without blocking the main thread.
- **Simplified Asynchronous Operations:** Traditional approaches for handling asynchronous tasks (e.g., using callbacks or `AsyncTask`) can lead to messy code and "callback hell." Coroutines allow you to write asynchronous code in a sequential manner, making it much easier to read and maintain.
- **Better Performance:** Coroutines are lightweight compared to threads. You can launch thousands of coroutines without overloading system resources, unlike threads, which are more resource-heavy.
- **Structured Concurrency:** Coroutines help manage tasks within specific scopes, ensuring that tasks are safely cancelled when no longer needed or when the scope (like an Activity or ViewModel) is destroyed.

## Key Concepts in Kotlin Coroutines

- **Coroutine Scope:** This defines the lifecycle of the coroutines. You need to specify the scope under which the coroutine operates, such as `GlobalScope` for global coroutines or `lifecycleScope` tied to the Activity/Fragment lifecycle.
- **Suspending Functions:** These are functions that can be paused and resumed without blocking the thread. They are used for tasks like network requests or long-running operations.
- **Dispatchers:** Dispatchers determine the thread or thread pool that the coroutine will run on. Common dispatchers include:
  - `Dispatchers.Main` (runs on the main UI thread)
  - `Dispatchers.IO` (for input/output operations like network calls)
  - `Dispatchers.Default` (for CPU-intensive tasks)
- **launch and async:** These are the coroutine builders.

- `launch` is used to start a coroutine when you don't need a result.
- `async` is used when you need to get a result back, and it returns a `Deferred` object.

## Examples of Kotlin Coroutines with Jetpack Compose

```

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.*
import androidx.compose.material3.Button
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

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

@Composable
fun MyComposeApp() {
    var text by remember { mutableStateOf("Click to Load Data") }
    val coroutineScope = rememberCoroutineScope()

    Column(
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text(text = text, style = MaterialTheme.typography.titleSmall)
        Spacer(modifier = Modifier.height(20.dp))
        Button(onClick = {
            coroutineScope.launch {
                text = "Loading..."
                val result = fetchData()
                text = result
            }
        }) {
            Text("Load Data")
        }
    }
}

```

```
}

suspend fun fetchData(): String {
    delay(2000) // Simulating network delay
    return "Data Loaded Successfully"
}
```

# Jetpack Compose + Coroutines + Retrofit (Example)

## 1. Add Dependencies

Add these to `build.gradle.kts` (Kotlin DSL):

```
// Retrofit for networking
// Check latest version: https://mvnrepository.com/artifact/com.squareup.retrofit2/retrofit
implementation ("com.squareup.retrofit2:retrofit:2.11.0")
implementation ("com.squareup.retrofit2:converter-gson:2.11.0")

// Coroutine support for Retrofit
// Check latest version: https://mvnrepository.com/artifact/org.jetbrains.kotlinx/kotlinx-coroutines
implementation ("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.10.1")
```

## 2. Add Internet Permission

Add these to `AndroidManifest.xml`:

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

## 3. Define Post Model - Data Class

Create `Post.kt`:

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

## 4. Create Retrofit API Service

Create `ApiService.kt`:

```
import retrofit2.http.GET

interface ApiService {
    @GET("posts") // Fetching a single post for simplicity
    suspend fun getPosts(): List<Post> // suspending function that can be paused and resumed without
}
```

## 5. Create Retrofit Instance

Create `RetrofitClient.kt`:

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

object RetrofitClient {
    private const val BASE_URL = "https://jsonplaceholder.typicode.com/"

    val apiService: ApiService by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(ApiService::class.java)
    }
}
```

## 6. Use Repository Pattern to separate network logic into a repository

Create `PostRepository.kt`:

```
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext

class PostRepository {
    suspend fun getPosts(): Result<List<Post>> {
        return try {
            val response = withContext(Dispatchers.IO) { RetrofitClient.apiService.getPosts() }
            Result.success(response)
        } catch (e: Exception) {
            Result.failure(e)
        }
    }
}
```

- line 7: `Dispatchers.IO` is optimised for **network/database operations**.
  - `withContext(Dispatchers.IO)` switches execution to a background thread.

## 7. Use ViewModelScope in ViewModel

Create `APIViewModel.kt`:

```
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
```

```

import kotlinx.coroutines.launch

class APIViewModel(private val repository: PostRepository): ViewModel() {
    private val _posts = MutableStateFlow<List<Post>>(emptyList()) // Using StateFlow (better than
    val posts: StateFlow<List<Post>> = _posts

    private val _error = MutableStateFlow<String?>(null) // StateFlow for errors
    val error: StateFlow<String?> = _error

    fun fetchPosts() {
        viewModelScope.launch {
            repository.getPosts().fold(
                onSuccess = { _posts.value = it },
                onFailure = { _error.value = it.message }
            )
        }
    }
}

```

- line 15: `viewModelScope.launch{}` ensures the coroutine is **cancelled when ViewModel is cleared.**
  - prevent memory leaks.
- line 8 & 11: **Using StateFlow** makes it easy to **observe UI state** in Jetpack Compose.
- line 18: Observe app crashes due to network failures.
  - Return failure message if onFailure

## 8. Use Jetpack Compose's collectAsState()

Modify `MainActivity.kt`:

```

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.Button
import androidx.compose.material3.Card
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {

```

```

    val repository = PostRepository()
    val viewModel = APIViewModel(repository) // Inject repository manually

    super.onCreate(savedInstanceState)
    setContent {
        Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
            RetrofitScreen(
                modifier = Modifier.padding(innerPadding).fillMaxSize(),
                viewModel
            )
        }
    }
}

@Composable
fun RetrofitScreen(modifier: Modifier = Modifier, viewModel: APIViewModel){
    val posts by viewModel.posts.collectAsState()
    val error by viewModel.error.collectAsState()

    Column(
        modifier = modifier,
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Top
    ) {
        Button(onClick = {
            viewModel.fetchPosts()
        }) {
            Text("Fetch API Data")
        }

        Spacer(modifier = Modifier.height(10.dp))

        // Refresh and display posts list after recomposition
        error?.let {
            Text(text = "Error: $it", color = Color.Red)
        }
        posts.isNotEmpty().let {
            PostList(posts)
        }
    }
}

// Composable function to display the list of posts
@Composable
fun PostList(posts: List<Post>){
    LazyColumn(
        modifier = Modifier
            .fillMaxSize()
            .padding(8.dp)
    ) {
        // Iterate over the list of posts
        items(posts, key = {post -> post.id}){ post ->
            // Display each post using the PostItem composable
        }
    }
}

```

```

        PostItem(post)
    }
}

// Composable function to display individual user items
@Composable
fun PostItem(post: Post){
    Card(
        modifier = Modifier
            .fillMaxWidth() // make them occupy the whole width
            .padding(10.dp) // space between post items
    ) {
        Column(
            modifier = Modifier.padding(8.dp)
        ) {
            // Display the post title
            Text(
                text = post.title,
                style = MaterialTheme.typography.titleMedium,
                modifier = Modifier.padding(bottom = 4.dp)
            )
            // Display the post body
            Text(
                text = post.body,
                style = MaterialTheme.typography.bodyMedium
            )
        }
    }
}

```

- line 21: Manually Provide Repository in Activity
- line 22: instantiate **ViewModel** in an Activity and pass it down:
- line 38&39: Observe ViewModel in Jetpack Compose
  - **Efficient recomposition** with `collectAsState()`

## Summary of Best Practices

- **Use `Dispatchers.IO` for network calls.**
  - `Dispatchers.IO` is optimized for network/database operations.
- **Use `viewModelScope.launch{}`** inside ViewModel to launch coroutines
  - Avoids memory leaks.
- **Handle API errors** with `try-catch` or `Result<T>`.
  - No more crashes
- **Implement a Repository layer** for cleaner architecture.
  - Follows Separation of Concerns.
- **Observe StateFlow in Compose** using `collectAsState()`.

- Use `StateFlow` instead of `LiveData`, which is Jetpack Compose-friendly.



MONASH  
University

**FIT2081**  
**Mobile Application**  
**Development**



# Coroutines & Async Programming

**Week 8**

Jiazhou 'Joe' Liu

# Learning Objectives

- **Fundamental concepts**
  - Concurrency basics for mobile app development
  - Introduction to coroutines and async tasks in Kotlin
  - Best practices for using coroutines
- **Practice Network calls in Compose projects (video tutorial)**

# Concurrency in Mobile App Development

# Concurrency in Mobile App Development

**Concurrency** is the ability of your mobile app to **execute multiple tasks at the same time**, such as fetching data, updating the UI, and responding to user input.

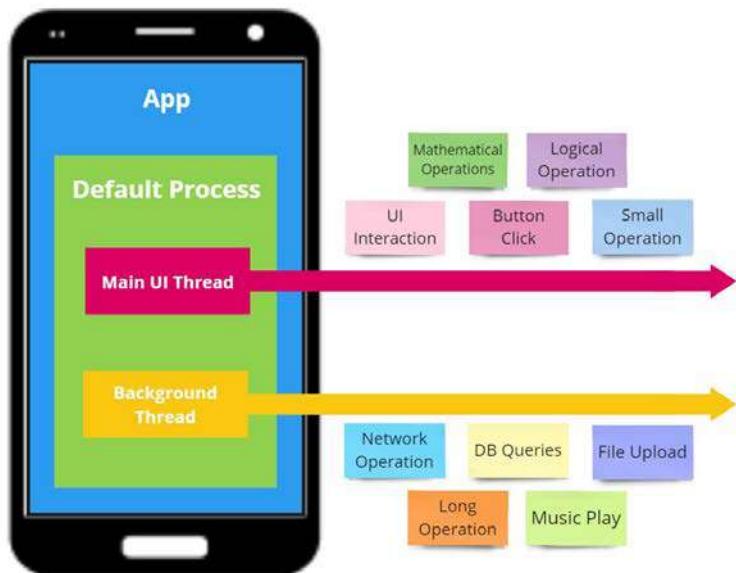
## Key Concepts of Concurrency in Mobile App Development

1. **Multithreading** – Running multiple threads to perform tasks concurrently.
2. **Parallelism** – Executing multiple tasks at the same time on multi-core processors.
3. **Asynchronous Programming** – Performing tasks without blocking the main thread (UI thread).
4. **Thread Pooling** – Managing multiple threads efficiently to avoid performance overhead.
5. **Task Queues** – Scheduling and managing background tasks in an orderly manner.

# Single-Threaded vs. Multi-Threaded

Threads allow a program to operate more efficiently by doing **multiple** things at the same time.

Threads can be used to perform complicated tasks in the background **without interrupting the main program**.



## Single-Threaded Use Cases

Single-threaded programs are suitable for applications with low computational complexity or minimal I/O operations, such as command-line utilities and **simple data processing tasks**.

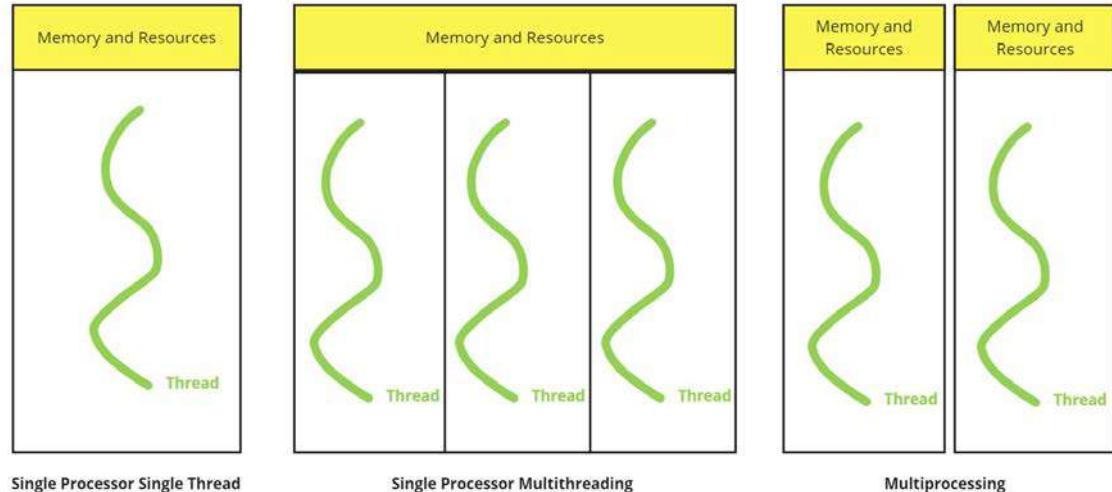
## Multi-Threaded Use Cases

Multi-threaded programs shine in scenarios with computationally intensive tasks, parallel processing, and applications with significant I/O operations, like **web services and database CRUD**.



# Parallelism

Parallelism involves breaking down tasks into smaller parts and executing them **at the same time** across multiple CPU cores or threads. This differs from **concurrency**, where tasks only appear to run simultaneously but may actually be interleaved.



## Multithreading: Sharing the Stage

Each thread represents a separate flow of control within a program. They **share the same memory space and resources** (like the CPU cache) but can execute concurrently.

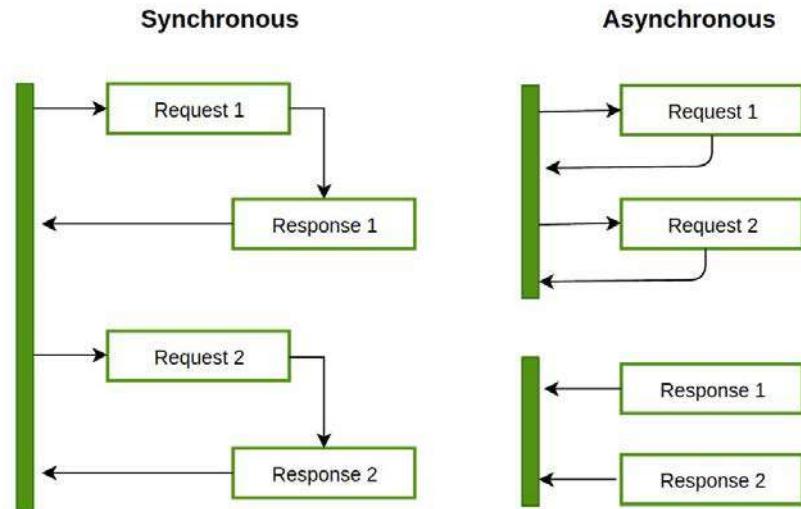
## Multiprocessing: Dividing and Conquering

Each CPU can run its own process (an independent program instance) with **its own memory space and resources**.

# Asynchronous Programming

A programming paradigm that allows a system to handle **multiple tasks** without waiting for one to finish before starting another. It improves efficiency, especially in applications that involve **I/O operations** like fetching data from a database, making network requests, or reading/writing files.

Feature	Concurrency	Asynchronous Programming
Definition	Running multiple tasks in an overlapping manner	Running tasks efficiently without blocking
Execution	Uses multiple threads/processes	Uses event loops (single-threaded but non-blocking)
Blocking	Tasks may block threads	Non-blocking (uses <code>async/await</code> )
Example	Running two calculations at the same time	Fetching data while keeping UI responsive



# Thread Pooling

A collection (or "pool") of **pre-created** threads is managed and **reused** for executing multiple tasks.

## How Thread Pooling Works

1. A pool of worker threads is created and maintained.
2. Tasks are submitted to the thread pool instead of creating new threads for each task.
3. The pool assigns tasks to available threads.
4. Once a thread completes a task, it returns to the pool for reuse.

```
import java.util.concurrent.Executors

val executor = Executors.newFixedThreadPool(4) // Pool of 4 threads

fun runTask() {
    executor.execute {
        println("Task running on thread: ${Thread.currentThread().name}")
    }
}
```

# Task Queues

A data structure used to manage and execute tasks **sequentially** or **concurrently** in Android applications. It holds units of work (“tasks”) until a thread is ready to execute them.

Whenever you see `.post()`, `.execute()`, `enqueue()` or `launch()`, you’re likely putting a runnable, message or coroutine onto a task queue. The runtime then pulls tasks off that queue and executes them on the appropriate thread.

```
import kotlinx.coroutines.*  
  
fun main() = runBlocking {  
    val taskQueue = mutableListOf<Deferred<Unit>>()  
  
    repeat(3) { i ->  
        val task = async(Dispatchers.Default) {  
            delay(1000L)  
            println("Task $i completed")  
        }  
        taskQueue.add(task)  
    }  
  
    taskQueue.awaitAll()  
}
```

# Concurrency in Android

## Threads & Handlers

Basic way to run tasks in the background.

## AsyncTask (Deprecated in API level 30)

Used for simple background tasks.

## Coroutines (Kotlin)

Recommended for structured concurrency.

## Executors & WorkManager

Used for scheduled and long-running tasks.



# Coroutines

## *Structured Concurrency*

# Structured Concurrency

**Structured Concurrency** is a programming paradigm that ensures tasks are started, managed, and completed in a structured and predictable manner. It helps developers avoid issues like **leaking background tasks**, **resource mismanagement**, and **callback hell** by **scoping tasks to a specific context** (e.g., a function or a lifecycle).

# Kotlin Coroutines

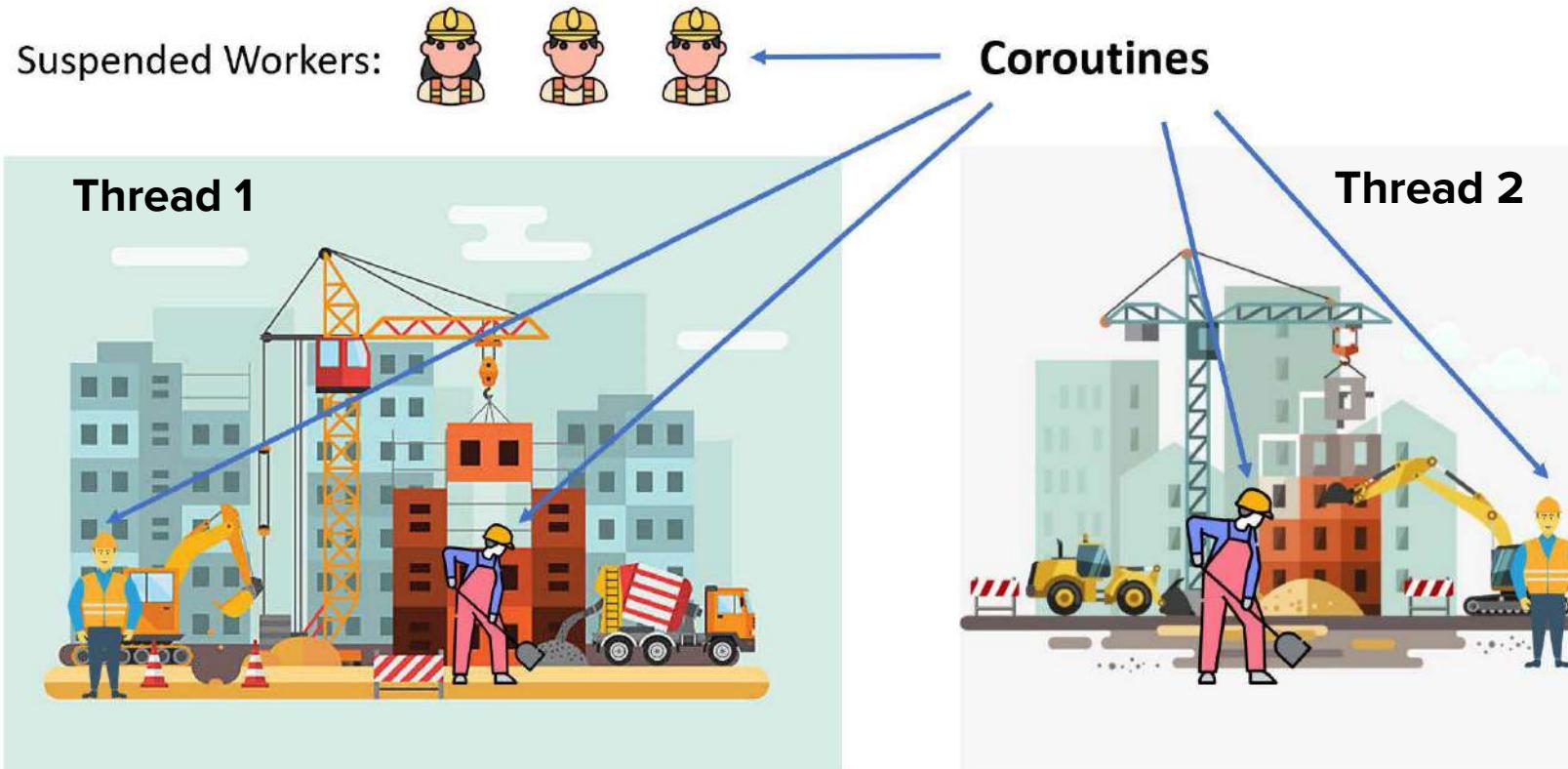
Coroutines were introduced to simplify asynchronous programming and support running concurrent operations

- They help with managing long-running jobs without blocking the **main thread**
- Coroutines use a '**suspension**' technique
- A suspend function has the **suspend keyword** to enforce the function to be called in a coroutine

A coroutine is similar to the concept of **thread** but is not bound to any particular thread

- Its method is 'lightweight' because **multiple coroutines can run on a single thread.**
- It could **suspend** (pause) its execution in one thread and **resume** in another thread
- When a coroutine is suspended, its local variables will be saved. When the execution of a suspended coroutine is resumed, it will be from the place where it was suspended
- This method improves resource consumption and concurrent operations

# Kotlin Coroutines Illustration



# The Main (UI) Thread

When an Android application is launched, the system creates a single ‘main’ thread, aka the ‘UI’ thread

- **Rule 1: Do not block the main/UI thread**
  - Performing long operations, such as accessing the network or database operations, block the UI thread so we must run them on background (worker) threads
- **Rule 2: Do not access the Android UI toolkit from outside the UI thread**
  - We shouldn’t update or access the UI elements from any thread other than the UI (main) thread

## Examples of background operations an app is doing outside its main workflow

### Asynchronous work

- A time-consuming work can run as an asynchronous work on other threads
- *Kotlin coroutines* and *Java threads* (Executors and Future objects)

### Foreground services

- They perform operations noticeable to the user (not to be interrupted), such as playing music, and can create a heavy load so their use is limited
- They can be implemented using a *Service* or a *WorkManager*

### Background work

- When a job should keep running even if the user leaves the app
- Use *CoroutineWorker* and *WorkManager*



# Coroutine - Suspending Function

A **suspending function** in Kotlin is a function that can be paused and resumed without blocking the thread. It is defined using the **suspend** keyword and can only be called from another suspending function or within a coroutine.

## Key Features of a Suspending Function:

1. **Non-Blocking:** It does not block the thread while waiting for the result.
2. **Can Call Other Suspending Functions:** A suspending function can call other suspending functions, making it easier to compose asynchronous operations.
3. When a coroutine is suspended, its **local variables will be saved**. When the execution of a suspended coroutine is resumed, it will be from the place where it was suspended

```
@Dao
interface PatientDao {
    //suspend is a coroutine function that can be
    paused and resumed at a later time.
    //suspend is used to indicate that the function
    will be called from a coroutine.
    @Insert
    suspend fun insert(patient: Patient)

    @Update
    suspend fun update(patient: Patient)

    @Delete
    suspend fun delete(patient: Patient)

    @Query("DELETE FROM patients")
    suspend fun deleteAllPatients()

    @Query("SELECT * FROM patients ORDER BY id ASC ")
    fun getAllPatients(): Flow<List<Patient>>
}
```

# Coroutine - Lifecycle-aware coroutine scopes

A **CoroutineScope** defines a scope for coroutines and keeps track of any coroutine it creates using launch or async

- **ViewModelScope** is defined for each ViewModel in an Android app
- Any coroutine launched in the ViewModel scope is **automatically canceled** if the ViewModel is destroyed
- Another option, [LifeCycleScope](#)

```
fun fetchPosts () {
    viewModelScope.launch {
        repository.getPosts().fold(
            onSuccess = { _posts.value = it },
            onFailure = { _error.value = it.message }
        )
    }
}
```

[https://developer.android.com/topic/libraries/architecture/coroutines#view\\_modelscope](https://developer.android.com/topic/libraries/architecture/coroutines#view_modelscope)

# Coroutine - Start a Coroutine

After creating a suspend function, we must start/execute it in one of two ways:

1. Using the **launch** starts a coroutine but does not return the result (fire and forget)
2. Using the **async** starts a coroutine and return a result calling **await()**

```
//inserts a patient into the repo
fun insert(patient: Patient) = viewModelScope.launch {
    patientRepo.insert(patient)
}
//deletes a patient from the repo
fun delete(patient: Patient) = viewModelScope.launch {
    patientRepo.delete(patient)
}
//updates a patient in the repo
fun update(patient: Patient) = viewModelScope.launch {
    patientRepo.update(patient)
}
```

```
suspend fun fetchTwoDocs () =
    coroutineScope {
        val deferredOne = async { fetchDoc(1) }
        val deferredTwo = async { fetchDoc(2) }
        deferredOne.await()
        deferredTwo.await()
    }
```

<https://developer.android.com/kotlin/coroutines.coroutines-adv#start>

# Coroutine - Dispatchers

Kotlin coroutines use **dispatchers** to determine which threads are used for coroutine execution

1. **Dispatchers.Default** - This dispatcher is optimized to perform CPU-intensive work e.g. sorting a list
2. **Dispatchers.IO** – it is used to perform the IO related work (disk or network) outside the main thread, e.g. working with Room or running any network operations
3. **Dispatchers.Main** – it runs a coroutine on the main Android thread, e.g. when interacting with the UI

```
class PostRepository {  
    suspend fun getPosts(): Result<List<Post>> {  
        return try {  
            val response = withContext(Dispatchers.IO)  
            RetrofitClient.apiService.getPosts()  
        } catch (e: Exception) {  
            Result.failure(e)  
        }  
    }  
}
```

<https://developer.android.com/kotlin/coroutines/coroutines-adv#main-safety>

# Coroutine - Flow and StateFlow

**Flow<T>** is a cold (lazy) sequential stream of values. Nothing happens until you call collect {...}. Each collector gets its own fresh execution of the producer. **Use Flow for streams you drive on-demand.**

**StateFlow<T>** is a hot (eager), shared state-holder that always has a current value. It starts emitting immediately and keeps the latest value in memory. So all collectors see the same , ongoing sequence and receive the most recent value on subscription. **Use StateFlow when you have mutable state that multiple parties observe.**

```
@Dao
interface WorkoutDao {
    //The return type is a Flow, which is a data stream
    //that can be observed for changes.
    @Query("SELECT * FROM workouts ORDER BY date DESC")
    fun getAllWorkouts(): Flow<List<Workout>>
}
```

```
class WorkoutViewModel(application: Application) :
    AndroidViewModel(application) {
    val allWorkouts: StateFlow<List<Workout>>
}
```

# Best Practices to Make Network Calls

- **Implement a Repository layer** for cleaner architecture.
  - Follows Separation of Concerns.
- **Use Dispatchers.IO for network calls.**
  - Dispatchers.IO is optimized for network/database operations.
- **Handle API errors** with try-catch or Result<T>.
  - No more crashes

```
class PostRepository {  
    suspend fun getPosts(): Result<List<Post>> {  
        return try {  
            val response = withContext(Dispatchers.IO) {  
                RetrofitClient.apiService.getPosts()  
            }  
            Result.success(response)  
        } catch (e: Exception) {  
            Result.failure(e)  
        }  
    }  
}
```

# Best Practices to Make Network Calls (cont.)

- Use `viewModelScope.launch{}` inside ViewModel to launch coroutines
  - Avoids memory leaks.

```
class APIViewModel(private val repository: PostRepository): ViewModel() {  
    private val _posts = MutableStateFlow<List<Post>>(emptyList())  
    val posts: StateFlow<List<Post>> = _posts  
  
    fun fetchPosts() {  
        viewModelScope.launch {  
            ...  
        } } }
```

- Observe **StateFlow** in Compose using `collectAsState()`.
  - Use StateFlow instead of LiveData, which is Jetpack Compose-friendly.

```
@Composable  
fun RetrofitScreen(modifier: Modifier = Modifier, viewModel: APIViewModel) {  
    val posts by viewModel.posts.collectAsState()  
}
```

# Reminders and Announcements

## Lab activities this week

- Develop an app covering multiple topics
  - Database (multiple tables and DAOs + complex queries)
  - Global Auth Provider (manage user authentication status)
  - Navigation with Routing (passing parameters)

## Week 9

- *Look into the MVVM architecture*
- *Recommended Android App Architecture*

# Reference

- Pari Delir Haghghi (S1 2024) Kotlin Coroutines [PowerPoint slides], FIT5046: Mobile and Distributed Computing Systems, Monash University.
- Worker icons: <https://www.flaticon.com/>
- Construction site clipping art: <https://au.pinterest.com/pin/634655772464432888/>

# Lab: Database, Navigation with Routing, MVVM

---

## Introduction

## Overview

In this lab, we will build an Android app that combines several advanced mobile development concepts. These topics are:

- Database
  - Multiple tables database
  - Complex Queries
  - Multiple DAOs
  - Authenticate the users from the database
- Global Auth Provider
- Navigation with Routing
  - Passing parameters through routes
- MVVM Architecture

The app's core functionality is to **record and display students' quiz attempts**. Users can log in as either a **student** or a **teacher (admin)**. Students can submit quiz results, and teachers can view **average marks** and **individual attempt histories**.

## Learning Objectives

By the end of this lab, you will be able to:

- Understand and implement Room databases with multiple related tables.
- Write complex SQL queries using Room and multiple DAO interfaces.
- Implement user authentication by validating login credentials stored in a database.
- Use a Global Auth Provider to maintain user state across the app.
- Navigate between screens using Jetpack Navigation, including passing parameters through routes.
- Apply the MVVM architecture to separate concerns, with ViewModels handling business logic.
- Understand how these concepts are used in real-world apps like learning management systems (e.g., Moodle, Canvas).

## Lab Tasks Overview

- Design the Database Schema

- Create tables for Users, Quizzes, and Attempts.
- Define relationships (e.g., one user can have many attempts).
- Implement Room with Multiple DAOs
  - Write DAOs to support operations like inserting attempts, fetching user details, and calculating averages.
- Set Up User Authentication
  - Build a login screen that validates credentials from the database.
  - Store the logged-in user using a Global Auth Provider (e.g., Singleton or Hilt).
- Navigation with Parameters
  - Navigate from the login screen to either the student dashboard or teacher dashboard, passing user roles and IDs as parameters.
- MVVM Structure
  - Create ViewModels for managing UI state and business logic, especially for login and data retrieval.
- Display Data with Complex Queries
  - Teachers should see average scores per student.
  - Students should see their own attempt history.

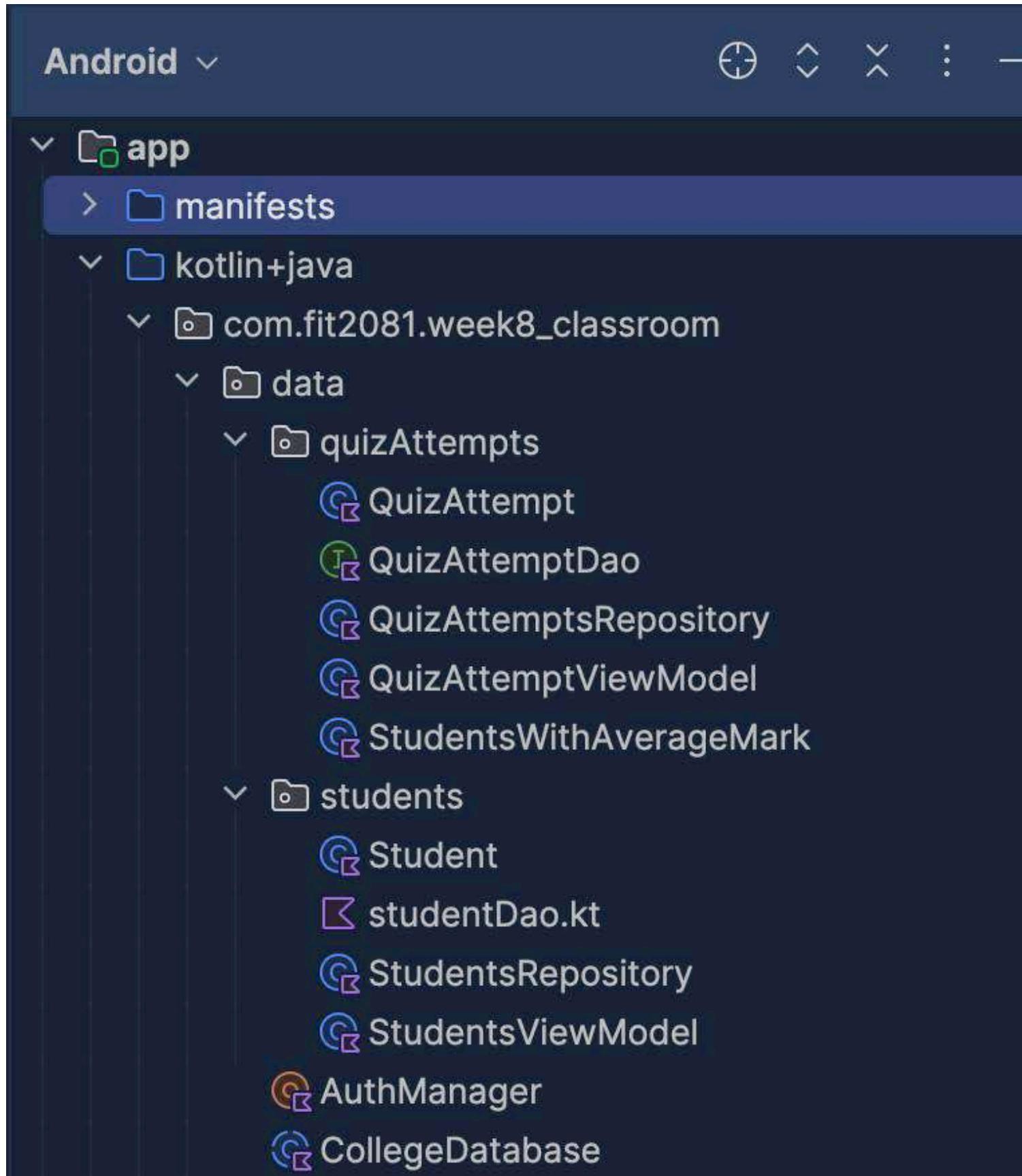
## Connection to Assignment 3

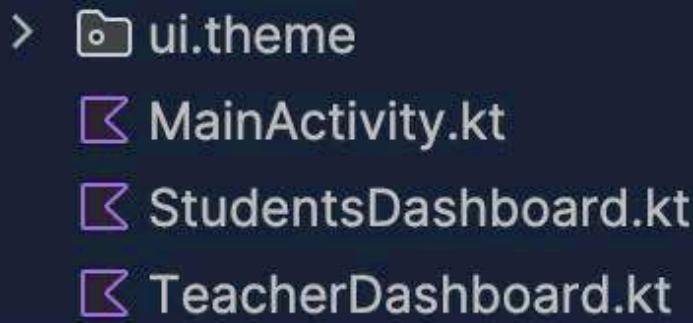
The skills in this lab directly support **Assignment 3**, particularly in:

- **Storing and retrieving structured data** using Room.
- **Implementing MVVM** to separate business logic from UI.
- **Handling user sessions** for different roles (student vs. teacher).

## App Structure – Detailed Version

**i** We will organise our database files within a 'data' sub-package. Since this app involves two tables, we will create a separate sub-package for each table, as depicted below.





# 1. Package Organisation

The app follows a **feature-based structure**, with clear separation of concerns between data handling, business logic (ViewModels), and UI.

## Package Breakdown:

- **data.quizAttempts** : Handles all data operations related to quiz attempts.
  - QuizAttempt : Entity class representing a single quiz attempt.
  - QuizAttemptDao : Interface defining SQL operations for quiz attempts.
  - QuizAttemptsRepository : Manages data access for quiz attempts, provides a clean API to ViewModels.
  - QuizAttemptViewModel : ViewModel handling logic for displaying quiz attempt data in the UI.
  - StudentsWithAverageMark : Likely a data class for complex queries, such as fetching students along with their average marks.
- **data.students** : Handles student-related data and logic.
  - Student : Entity class representing student information.
  - studentDao.kt : Interface defining SQL operations for student data (e.g., authentication).
  - StudentsRepository : Provides an abstraction over the student data source for the ViewModel.
  - StudentsViewModel : ViewModel that manages student-specific logic, likely including login/authentication.
- **AuthManager** :
  - A global singleton object responsible for holding the current logged-in user's state (e.g., name, student ID, role).
  - This helps manage **role-based navigation** and user-specific data access across the app.
- **CollegeDatabase** :
  - The main Room database class where all DAOs (for students and quiz attempts) are registered.
  - Responsible for providing database access throughout the app.

## 2. UI Components:

- **MainActivity.kt** : The entry point of the app. Likely responsible for setting up navigation and initializing the UI.
- **StudentsDashboard.kt** : UI screen shown to regular users (students) after successful login.
- **TeacherDashboard.kt** : UI screen shown to admin users (teachers) after successful login.

## 3. Navigation Flow:

- After login, based on the role stored in `AuthManager`, users are navigated:
  - **Students** → `StudentsDashboard`
  - **Teachers** → `TeacherDashboard`

## 4. Key Architectural Patterns:

- **MVVM (Model-View-ViewModel):**
  - **Model**: Entity classes (`Student`, `QuizAttempt`) and DAOs handle data persistence.
  - **ViewModel**: `QuizAttemptViewModel`, `StudentsViewModel` provide logic and data to the UI.
  - **Repository**: Acts as an intermediary between ViewModel and DAOs.
- **Global Authentication:**
  - `AuthManager` is used across the app to keep track of the current user session.

## Part 1: The Students Table

In this table, we will save:

- student name
- student password
- student ID

## Entity (Students Table Model)

The **Student** class represents a student record in the Room database. Each instance of this class corresponds to a row in the students table.

```
1  /**
2   * Represents a student entity within the Room database.
3
4  */
5  @Entity(tableName = "students")
6  data class Student(
7      // Primary key with auto-generation
8      @PrimaryKey(autoGenerate = true) val id: Int = 0,
9      val studentName: String, // Student's name
10     val studentPassword: String, // Student's password
11     // Student ID (optional, defaults to empty string)
12     val studentId: String = ""
13 )
```

codesnap.

## DAO

The **StudentDao** interface defines database operations for managing students in the Room database. It allows you to insert, retrieve individual students, list all students, and perform advanced queries like calculating average quiz marks per student.

```

1  @Dao
2  interface StudentDao {
3      /**
4      * Inserts a new student into the database.
5      *
6      * @param student The [Student] object to be inserted.
7      */
8      @Insert
9      suspend fun insert(student: Student)
10
11     /**
12     * Retrieves a student from the database based on their ID.
13     *
14     * @param studentId The ID of the student to retrieve,
15     * @return The [Student] object if found,
16     * null otherwise (handled by Room as per suspend function).
17     */
18     @Query("SELECT * FROM students WHERE studentId = :studentId")
19     suspend fun getStudentById(studentId: String): Student
20
21     /**
22     * Retrieves all students from the database.
23     *
24     * @return A [Flow] emitting a list of all [Student]
25     * objects in the database. The Flow will emit
26     * a new list whenever the underlying data changes.
27     */
28     @Query("SELECT * FROM students")
29     fun getAll(): Flow<List<Student>>
30
31     /**
32     * Retrieves a list of students along with their
33     * average quiz marks.
34     *
35     * This query performs an inner join between the `students`
36     * and `quiz_attempts` tables,
37     * calculates the average `finalMark` for each student,
38     * and groups the results by student ID.
39     *
40     * @return A [Flow] emitting a list of [StudentsWithAverageMark] objects.
41     * Each object contains the
42     * student's ID, name, and average quiz mark.
43     * The Flow will emit a new list whenever the underlying data changes.
44     */
45     @Query("SELECT students.studentId, students.studentName, AVG(quiz_attempts.finalMark) " +
46             "as averageMark FROM students Inner Join quiz_attempts " +
47             "ON students.studentId = quiz_attempts.studentId Group By students.studentId")
48     fun getStudentsWithAverageMarks(): Flow<List<StudentsWithAverageMark>>
49 }

```

codesnap.dev

## Complex Query – getStudentsWithAverageMarks()

### *SQL Explanation:*

- Performs an **INNER JOIN** between:
  - **students table:** provides student IDs and names.

- **quiz\_attempts table:** contains quiz scores.
- **Calculates the average** of each student's **finalMark** using the SQL **AVG()** function.
- **GROUP BY** `students.studentId` ensures that:
  - Each row in the result corresponds to **one student**.
  - The **averageMark** is calculated **per student**.

## Student Table

student_id	student_name	student_password
1	Alice	pass123
2	Bob	pass456
3	Charlie	pass789

Table after INNER JOIN + GROUP BY

## Quiz attempts Table

student_id	quiz_id	quiz_date	final_mark
1	101	2025-04-01	1
1	102	2025-04-02	2
2	101	2025-04-01	3
3	102	2025-04-02	0

Table after INNER JOIN + GROUP BY

student_id	student_name	average_mark
1	Alice	1.5
2	Bob	3
3	Charlie	0

## Returned Data:

- The method returns a **Flow** of **List<StudentsWithAverageMark>**.
- Each object in this list contains:
  - **studentId**: The unique ID of the student.
  - **studentName**: The name of the student.
  - **averageMark**: The calculated average quiz score for that student.

## Reactive Behaviour with Flow:

- Since the return type is **Flow**, any change in the **students** or **quiz\_attempts** tables will automatically update the data observed in the UI. Ideal for real-time dashboards showing average performance.

# Repository

```
1 class StudentsRepository(context: Context) {
2
3     // Get the StudentDao instance from the database
4     private val studentsDao = CollegeDatabase.getDatabase(context).studentDao()
5
6     /**
7      * Inserts a new student into the database.
8      * @param student The student object to be inserted.
9      */
10    suspend fun insertStudent(student: Student) {
11        studentsDao.insert(student)
12    }
13
14    /**
15     * Retrieves a student from the database by their ID.
16     * @param studentId The ID of the student to retrieve.
17     * @return The student object with the given ID.
18     */
19    suspend fun getStudentById(studentId: String): Student {
20        return studentsDao.getStudentById(studentId)
21    }
22
23    /**
24     * Retrieves all students from the database as a Flow.
25     * This allows observing changes to the student list over time.
26     * @return A Flow emitting a list of all students.
27     */
28    fun getAllStudents(): Flow<List<Student>> = studentsDao.getAll()
29
30    /**
31     * Retrieves students along with their average marks as a Flow.
32     * This allows observing changes to the student list and their average marks over time.
33     * @return A Flow emitting a list of StudentsWithAverageMark objects.
34     *         Each StudentsWithAverageMark object contains student information and their average mark.
35     */
36    fun getStudentsWithAverageMarks(): Flow<List<StudentsWithAverageMark>> =
37        studentsDao.getStudentsWithAverageMarks()
38
39 }
40
```

codesnap.dev

## Students View Model

```

1 class StudentsViewModel(context: Context) : ViewModel() {
2
3     val repository = StudentsRepository(context = context)
4
5     /**
6      * Retrieves a Flow of all students.
7      *
8      * This Flow emits a list of [Student] objects, representing all students in the database.
9      * It updates whenever the student data changes in the database.
10     *
11     * @return A Flow of a list of [Student] objects.
12     */
13    val allStudents: Flow<List<Student>> = repository.getAllStudents()
14
15    /**
16     * Inserts a new student into the database.
17     *
18     * This is a suspend function and should be called within
19     * a coroutine or another suspend function.
20     *
21     * @param student The [Student] object to be inserted into the database.
22     */
23    suspend fun insert(student: Student) = repository.insertStudent(student)
24
25    /**
26     * Retrieves a student from the database by their ID.
27     *
28     * This is a suspend function and should be called within
29     * a coroutine or another suspend function.
30     *
31     * @param studentId The ID of the student to retrieve.
32     * @return The [Student] object with the specified ID, or null if no such student exists.
33     */
34    suspend fun getStudentById(studentId: String): Student {
35        return repository.getStudentById(studentId)
36    }
37
38    /** Retrieves a Flow of students with their average marks.
39     * @return A flow of list of [StudentsWithAverageMark] objects*/
40    fun getStudentsWithAverageMarks(): Flow<List<StudentsWithAverageMark>>
41        = repository.getStudentsWithAverageMarks()
42
43
44    // Factory class for creating instances of StudentsViewModel
45    class StudentsViewModelFactory(context: Context) : ViewModelProvider.Factory {
46        private val context = context.applicationContext
47        override fun <T : ViewModel> create(modelClass: Class<T>): T =
48            StudentsViewModel(context) as T
49    }
50 }
51

```

## StudentsWithAverageMark data class

This class is used as a data model for the results of the DAO queries.

```
1 class StudentsWithAverageMark {  
2     var studentId: String = ""  
3     var studentName: String = ""  
4     var averageMark: Double = 0.0  
5 }
```

## Part 2: The Quiz Attempts Table

In this table, we will save:

- student ID (who did the attempt)
- quiz ID
- quiz date
- quiz final mark

## Entity (Quiz Attempt Model)

The **QuizAttempt** class represents a single quiz attempt made by a student. It is part of the Room database and corresponds to a table called "quiz\_attempts".

```
1  /**
2   * Represents a quiz attempt made by a student.
3   *
4   * This data class is annotated with @Entity to mark it as a Room database table.
5   * The table name is "quiz_attempts".
6   */
7  @Entity(tableName = "quiz_attempts")
8  data class QuizAttempt(
9      /**
10       * Unique identifier for the quiz attempt.
11       * It is automatically generated by Room.
12       */
13      @PrimaryKey(autoGenerate = true)
14      val id: Int = 0,
15      /**
16       * Identifier of the student who attempted the quiz.
17       */
18      val studentId: String,
19      /**
20       * Identifier of the quiz that was attempted.
21       */
22      val quizId: String,
23      val quizDate: String,
24      val finalMark: Double
25  )
```

codesnap.dev

# DAO

The **QuizAttemptDao** interface defines data access methods for interacting with the `quiz_attempts` table in the Room database. It provides a clean abstraction over SQL queries, allowing other parts of your app (such as Repositories and ViewModels) to access data without dealing with raw SQL.

- **@Dao Annotation:**
  - Declares this interface as a **Data Access Object** for Room.
  - Room will automatically generate the implementation at compile time.
- **@Insert suspend fun insert(quizAttempt: QuizAttempt):**
  - Inserts a new `QuizAttempt` object into the `quiz_attempts` table.
  - It's a **suspend function**, meaning it should be called from a coroutine, ensuring the database operation runs off the main thread.
- **@Query("SELECT \* FROM quiz\_attempts")**
- **fun getAllQuizAttempt(): Flow<List<QuizAttempt>>:**
  - Retrieves **all quiz attempts** from the table.
  - Returns a **Flow** that emits updates whenever the data in the table changes.
  - Useful for real-time UI updates, as Flow works well with Jetpack Compose and LiveData.
- **@Query("SELECT \* FROM quiz\_attempts WHERE studentId = :studentId")**
- **fun getQuizAttemptByStudentId(studentId: String): Flow<List<QuizAttempt>>:**
  - Retrieves all quiz attempts **for a specific student**.
  - Takes `studentId` as a parameter and filters the results.
  - Also returns a **Flow**, making it reactive and auto-updating.

```
1  @Dao
2  /**
3   * Data Access Object (DAO) for interacting with the
4   * quiz_attempts table in the database.
5   */
6  interface QuizAttemptDao {
7      /**
8       * Inserts a new [QuizAttempt] into the database.
9       *
10      * @param quizAttempt The [QuizAttempt] object to be inserted.
11     */
12    @Insert
13    suspend fun insert(quizAttempt: QuizAttempt)
14
15    /**
16     * Retrieves all [QuizAttempt]s from the database as a [Flow] of lists.
17     *
18     * @return A [Flow] emitting a list of all [QuizAttempt]s in the table.
19     */
20    @Query("SELECT * FROM quiz_attempts")
21    fun getAllQuizAttempt(): Flow<List<QuizAttempt>>
22
23    /**
24     * Retrieves all [QuizAttempt]s for a specific student ID.
25     */
26    @Query("Select * FROM quiz_attempts WHERE studentId = :studentId")
27    fun getQuizAttemptByStudentId(studentId: String): Flow<List<QuizAttempt>>
28 }
```

codesnap.dev

## Data Repository

- This repository acts as an intermediary between the ViewModel and the local database
- providing methods to perform CRUD (Create, Read) operations on quiz attempts
- It encapsulates the data access logic, allowing the ViewModel to interact with the data layer
- without needing to know the specifics of how the database operations are implemented.

```
1 class QuizAttemptsRepository(context: Context) {
2     // Create an instance of the QuizAttempt DAO
3     private val quizAttemptDao =
4         CollegeDatabase.getDatabase(context).quizAttemptDao()
5
6     /**
7      * Retrieve all quiz attempts from the database.
8      * @return A Flow emitting a list of all quiz attempts.
9      */
10    val allAttempts: Flow<List<QuizAttempt>> = quizAttemptDao.getAllQuizAttempt()
11
12    /**
13     * Insert a new quiz attempt into the database.
14     * @param attempt The QuizAttempt object to be inserted.
15     */
16    suspend fun insert(attempt: QuizAttempt) {
17        quizAttemptDao.insert(attempt)
18    }
19
20    /**
21     * Retrieve quiz attempts for a specific student from the database.
22     * @param studentId The ID of the student.
23     * @return A Flow emitting a list of quiz attempts for the specified student.
24     */
25    fun getQuizAttemptByStudentId(studentId: String): Flow<List<QuizAttempt>> =
26        quizAttemptDao.getQuizAttemptByStudentId(studentId)
27 }
```

codesnap.dev

## View Model

- The QuizAttemptViewModel class plays a crucial role in the application's architecture by acting as an intermediary between the UI and data layers. It manages and provides quiz attempt data to the UI, handles user interactions related to quiz attempts (such as inserting new attempts or retrieving attempts by student ID), and ensures data consistency by utilising the QuizAttemptsRepository for all database operations.

```
1  /**
2  * ViewModel class for managing quiz attempts data.
3  * This class interacts with the QuizAttemptsRepository to perform database operations
4  * and provides data to the UI.
5  *
6  * @param context The application context.
7  */
8 class QuizAttemptViewModel(context: Context) : ViewModel() {
9
10    private val quizAttemptRepository: QuizAttemptsRepository =
11        QuizAttemptsRepository(context)
12
13    /**
14     * Flow of all quiz attempts.
15     */
16    val allAttempts: Flow<List<QuizAttempt>> = quizAttemptRepository.allAttempts;
17
18    /**
19     * Inserts a new quiz attempt into the database.
20     * @param quizAttempt The quiz attempt to be inserted.
21     */
22    fun insertQuizAttempt(quizAttempt: QuizAttempt) {
23        viewModelScope.launch { quizAttemptRepository.insert(quizAttempt) }
24    }
25
26    /**
27     * Retrieves quiz attempts for a specific student ID.
28     * @param studentId The ID of the student.
29     * @return A Flow emitting a list of quiz attempts for the given student ID.
30     */
31    fun getQuizAttemptByStudentId(studentId: String):
32        Flow<List<QuizAttempt>> = quizAttemptRepository.getQuizAttemptByStudentId(studentId)
33
34    class QuizAttemptViewModelFactory(context: Context) : ViewModelProvider.Factory {
35        private val context = context.applicationContext
36        // Use application context to avoid memory leaks
37        override fun <T : ViewModel> create(modelClass: Class<T>): T =
38            QuizAttemptViewModel(context) as T
39    }
40 }
```

## Part 3: Authentication Manager

**AuthManager** is a singleton object that manages user authentication status in the app. It keeps track of the currently logged-in user's ID and provides methods to log in, log out, and retrieve the current user ID.

### **Field and Method Explanation:**

- **object AuthManager :**
  - Declares **AuthManager** as a Kotlin **singleton**.
  - Only **one instance** exists for the entire app lifecycle.
  - Acts as a **central authority** for authentication state.
- **val \_userId: MutableState<String?> :**
  - A **MutableState** that holds the current user's ID.
  - Initially **null** (when no user is logged in).
  - Used in **Jetpack Compose** for reactive UI updates: any Composables observing this will automatically recompose when the value changes.
- **fun login(userId: String) :**
  - Sets **\_userId** to the **provided userId**, marking the user as logged in.
  - Triggers UI updates where **\_userId** is observed.
- **fun logout() :**
  - Resets **\_userId** to **null**, logging the user out.
  - Also triggers UI updates.
- **fun getStudentId(): String? :**
  - Returns the current **logged-in user's ID**, or **null** if no user is logged in.
  - Useful for checking login status or fetching user-specific data.

```
1 object AuthManager {
2     val _userId: MutableState<String?> = mutableStateOf(null)
3
4
5     fun login(userId: String) {
6         _userId.value = userId
7     }
8
9     fun logout() {
10        _userId.value = null
11    }
12
13    fun getStudentId(): String? {
14        return _userId.value
15    }
16 }
```

codesnap.dev

## How It Fits in Your App:

- After login, `AuthManager.login(userId)` is called.
- For role-based navigation or data access, `AuthManager.getStudentId()` is used.
- When the user logs out, `AuthManager.logout()` is called to clear their session.

## Part 4: Database Class

The **CollegeDatabase** class is the central access point for managing the app's local data. It uses the Room Library, providing a type-safe abstraction over SQLite, allowing efficient and reliable database interactions.

### Key Features:

- Defines the database schema with **Student** and **QuizAttempt** entities.
- Provides abstract methods to access **DAOs** for CRUD operations.
- Implements the **Singleton design pattern** to ensure only **one instance** of the database exists during the app's lifecycle.

### Annotation and Entity Definition:

- `@Database(entities = [...]):`
  - Declares this class as a Room **database**.
  - Lists **Student** and **QuizAttempt** as entities (tables).
  - **version = 1**: The current version of the database schema.
  - **exportSchema = false**: Disables schema export (useful for simple apps or avoiding schema files).

### Abstract DAO Access Methods:

- `abstract fun studentDao(): StudentDao`:
  - Provides access to all **student-related** database operations.
- `abstract fun quizAttemptDao(): QuizAttemptDao`:
  - Provides access to all **quiz attempt-related** operations.

### Singleton Pattern (companion object):

- Ensures **only one instance** of `CollegeDatabase` exists.
- `@Volatile private var Instance`:
  - Guarantees thread-safe access.
- `getDatabase(context: Context)`:
  - Checks if the instance exists.
  - If not, builds a new Room database with the name "**item\_database**".
  - Ensures the instance is created **only once** using **synchronized** block.

```
1  @Database(entities = [Student::class, QuizAttempt::class], version = 1, exportSchema = false)
2  /**
3   * Abstract class representing the college database.
4   * It extends RoomDatabase and provides access to the DAO interfaces for the entities.
5   */
6  abstract class CollegeDatabase : RoomDatabase() {
7      /**
8       * Provides access to the StudentDao interface for performing
9       * database operations on Student entities.
10      * @return StudentDao instance.
11      */
12      abstract fun studentDao(): StudentDao
13
14      /**
15       * Provides access to the QuizAttemptDao interface for
16       * performing database operations on QuizAttempt entities.
17       * @return QuizAttemptDao instance.
18      */
19      abstract fun quizAttemptDao(): QuizAttemptDao
20
21      companion object {
22          // Singleton instance of the database
23          @Volatile
24          private var Instance: CollegeDatabase? = null
25
26          /**
27           * Retrieves the singleton instance of the database.
28           * If an instance already exists, it returns the existing
29           * instance. Otherwise, it creates a new instance of the database.
30           * @param context The context of the application.
31           * @return The singleton instance of CollegeDatabase.
32          */
33          fun getDatabase(context: Context): CollegeDatabase {
34              return Instance ?: synchronized(this) {
35                  Room.databaseBuilder(context, CollegeDatabase::class.java, "item_database")
36                      .build()
37                      .also { Instance = it }
38              }
39          }
40      }
41 }
```

## Part 5: Student Dashboard

This screen allows **students** to:

- Answer a **3-question quiz** using checkboxes.
- Submit their responses.
- Have their attempt **saved** in the database with:
  - Student ID
  - Quiz ID
  - Date
  - Final mark (1 point per correct answer)

### First,

Let's create three boolean mutable states to store the answers to the questions.

- Each variable (`q1`, `q2`, `q3`) tracks whether the student marked **true/false** for a question.
- **remember { mutableStateOf(...)** } ensures that state persists during recompositions (Jetpack Compose).
- The state resets after submission.

```
var q1 by remember { mutableStateOf(false) }
var q2 by remember { mutableStateOf(false) }
var q3 by remember { mutableStateOf(false) }
```

### Second,

Let's develop the UI, consisting of a column with three questions and a button to submit the response.

A **Column** layout with:

- Title
- Instructions
- **3 Rows**, each containing:
  - A **Checkbox** (bound to `q1/q2/q3`).
    - Checkbox is **controlled** by `q1`.
    - When checked/unchecked, `q1` updates.

- The **same structure** is repeated for `q2` and `q3`.
  - A **Text** label with the question.
- A **Submit Button**.

```
1  Column(modifier = Modifier.padding(innerPadding)) {  
2      // Mutable state variables to track checkbox states for each question  
3      var q1 by remember { mutableStateOf(false) }  
4      var q2 by remember { mutableStateOf(false) }  
5      var q3 by remember { mutableStateOf(false) }  
6  
7      // Get the current context  
8      val _context = LocalContext.current  
9  
10     // Display the quiz title with styling  
11     Text(  
12         text = "FIT2081 Mobile Quiz",  
13         style = TextStyle(  
14             fontSize = 30.sp,  
15             fontWeight = androidx.compose.ui.text.font.FontWeight.Bold  
16         )  
17     )  
18     // Add vertical space  
19     Spacer(modifier = Modifier.padding(10.dp))  
20     // Display instructions  
21     Text(text = "Select the correct answers to the following statements")  
22     // Add vertical space  
23     Spacer(modifier = Modifier.padding(20.dp))  
24     // Row layout for the first question with checkbox and text  
25     Row(  
26         // Align content vertically in the center  
27         verticalAlignment = Alignment.CenterVertically,  
28         horizontalArrangement = Arrangement.spacedBy(8.dp)  
29     ) {  
30         Checkbox(  
31             checked = q1,  
32             onCheckedChange = { q1 = it }  
33         )  
34         // Display the first question text  
35         Text(text = "Paris is the capital of France.")  
36     }  
37     // Row layout for the second question with checkbox and text  
38     Row(  
39         // Align content vertically in the center  
40         verticalAlignment = Alignment.CenterVertically,  
41         horizontalArrangement = Arrangement.spacedBy(8.dp)  
42     ) {  
43         Checkbox(  
44             checked = q2,  
45             onCheckedChange = { q2 = it }  
46         )  
47         // Display the second question text  
48         Text(text = "Vincent van Gogh painted the Mona Lisa")  
49     }  
50     // Row layout for the third question with checkbox and text
```

```
51     Row(
52         // Align content vertically in the center
53         verticalAlignment = Alignment.CenterVertically,
54         horizontalArrangement = Arrangement.spacedBy(8.dp)
55     ) {
56         Checkbox(checked = q3, onCheckedChange = { q3 = it })
57         // Display the third question text
58         Text(text = "Mount Kosciuszko is the highest mountain in Australia")
59     }
60     Spacer(modifier = Modifier.padding(20.dp))
61     // Submit button
```

codesnap.dev

## After that,

I should add the button. The onclick method collects the states of the three boolean variables and calculates the total mark by assigning one mark for each correct answer. After that, it creates an instance of the QuizAttempt model class and sends it to the insertQuizAttempt method, which is located in the quizAttemptViewModel.

```
1  Button(
2      onClick = {
3          // Calculate the total mark based on selected answers
4          var totalMark = 0.0
5          if (q1) totalMark += 1
6          if (!q2) totalMark += 1
7          if (q3) totalMark += 1
8
9          // Generate a random 4-digit number for the quiz ID
10         var random4digits = Random.nextInt(1000, 9999)
11         // Create the quiz ID string
12         var quizId = "Qz$random4digits"
13         // Get the current date in dd/MM/yyyy format
14         val dateFormat = SimpleDateFormat("dd/MM/yyyy")
15         val currentDate = dateFormat.format(System.currentTimeMillis())
16         // Create a QuizAttempt object with student ID, quiz ID, date, and final mark
17         var attempt: QuizAttempt = QuizAttempt(
18             studentId = AuthManager.getStudentId().toString(),
19             quizId = quizId,
20             quizDate = currentDate,
21             finalMark = totalMark
22         )
23         // Launch a coroutine to insert the quiz attempt into the database
24         CoroutineScope(Dispatchers.IO).launch {
25             quizAttemptViewModel.insertQuizAttempt(attempt)
26         }
27         // Reset the checkbox states after submission
28         q1 = false
29         q2 = false
30         q3 = false
31         Toast.makeText(_context, "Quiz Attempt Submitted", Toast.LENGTH_SHORT).show()
32
33     }, modifier = Modifier.align(Alignment.End)
34 )
35     Text("Submit Quiz Attemp")
36
37     // Add a horizontal divider line
38     HorizontalDivider()
39 }
```

## Part 6: Teacher Dashboard

The teacher's role in our app has two main functions: adding new students to the database and listing their quiz attempts.

### First,

let's create three different routes for our navigation.

```
1 sealed class TeacherDashboardScreen(val route: String) {  
2     object AddStudent : TeacherDashboardScreen("add_student")  
3     object ListStudents : TeacherDashboardScreen("list_students")  
4     object StudentAttempts : TeacherDashboardScreen("student_attempts/{id}")  
5 }  
6
```

### Then,

get access to the students and quizzes view models:

```
1 // Initialize the StudentsViewModel using ViewModelProvider with a factory  
2 val studentViewModel: StudentsViewModel = ViewModelProvider(  
3     this, StudentsViewModel.StudentsViewModelFactory(this@TeacherDashboard)  
4 )[StudentsViewModel::class.java]  
5  
6  
7 // Initialize the QuizAttemptViewModel using ViewModelProvider with a factory  
8 val quizAttemptViewModel: QuizAttemptViewModel = ViewModelProvider(  
9     this, QuizAttemptViewModel.QuizAttemptViewModelFactory(this@TeacherDashboard)  
10 )[QuizAttemptViewModel::class.java]
```

### Next step,

let's add the top and bottom navigation bars.

```
1  val navController = rememberNavController()
2  Week8_classroomTheme {
3      Scaffold(
4          modifier = Modifier.fillMaxSize(),
5          topBar = {
6              TopAppBar(
7                  colors = TopAppBarDefaults.topAppBarColors(
8                      containerColor = MaterialTheme.colorScheme.primaryContainer,
9                      titleContentColor = MaterialTheme.colorScheme.primary,
10                 ),
11                 title = {
12                     Text("Teacher Dashboard")
13                 }
14             )
15         },
16         bottomBar = {
17             BottomAppBar(
18                 containerColor = MaterialTheme.colorScheme.primaryContainer,
19                 contentColor = MaterialTheme.colorScheme.primary,
20             ) {
21                 Row(
22                     Modifier.fillMaxWidth(),
23                     horizontalArrangement = Arrangement.SpaceAround
24                 ) {
25                     IconButton(onClick = { navController.navigate(TeacherDashboardScreen.AddStudent.route) }) {
26                         Icon(Icons.Filled.Add, contentDescription = "Add Student")
27                     }
28                     IconButton(onClick = { navController.navigate(TeacherDashboardScreen.ListStudents.route) }) {
29                         Icon(Icons.Filled.List, contentDescription = "List Students")
30                     }
31                 }
32             }
33         },
34     ) { innerPadding →
35         TeacherDashboardContent(
36             modifier = Modifier.padding(innerPadding),
37             studentViewModel = studentViewModel,
38             navController = navController,
39             quizAttemptViewModel = quizAttemptViewModel
40         )
41     }
42 }
43 }
44 }
```

codesnap.dev

The bottom bar has two icons: the first one navigates to the add student screen, while the second icon takes you to the list of students.

The TeacherDashboardContent is a simple function that wraps the NavHostTeacher by a column

```
1 @Composable
2 fun TeacherDashboardContent(
3     modifier: Modifier = Modifier,
4     studentViewModel: StudentsViewModel,
5     navController: NavHostController,
6     quizAttemptViewModel: QuizAttemptViewModel
7 ) {
8
9     Column(
10         modifier = modifier.fillMaxSize(),
11         verticalArrangement = Arrangement.Top,
12         horizontalAlignment = Alignment.CenterHorizontally
13     ) {
14         NavHostTeacher(
15             navController = navController,
16             studentViewModel = studentViewModel,
17             quizAttemptViewModel = quizAttemptViewModel
18         )
19     }
20 }
```

The routing of the screen navigation is defined in the following function:

```
@Composable
fun NavHostTeacher(
    navController: NavController, // Navigation controller to navigate between screens
    studentViewModel: StudentsViewModel, // ViewModel to manage student data
    quizAttemptViewModel: QuizAttemptViewModel, // ViewModel to manage quiz attempt data
    modifier: Modifier = Modifier // Optional modifier for customizing the layout
) {
    // Create a navigation host to manage navigation between different teacher dashboard screens
    NavHost(
        navController = navController, // Controller that handles navigation events
        startDestination = TeacherDashboardScreen.AddStudent.route, // Initial screen to display is AddStudent
        modifier = modifier // Apply any layout modifiers passed to this composable
    ) {
        // Define navigation route for the Add Student screen
        composable(TeacherDashboardScreen.AddStudent.route) {
            AddStudent(studentViewModel = studentViewModel) // Display AddStudent composable with required ViewModel
        }

        // Define navigation route for the List Students screen
        composable(TeacherDashboardScreen.ListStudents.route) {
            ListStudents(studentsViewModel = studentViewModel, navController) // Display ListStudents with ViewModel and navController
        }

        // Define navigation route for the Student Attempts screen with dynamic ID parameter
        composable(TeacherDashboardScreen.StudentAttempts.route) {
            val studentId = it.arguments?.getString("id") // Extract the student ID from the navigation arguments
            // Display the StudentsAttempts composable with quiz attempt data for the specific student
            StudentsAttempts(quizAttemptViewModel = quizAttemptViewModel, studentId.toString())
        }
    }
}
```

codesnap.dev

## Adding a New Student

The AddStudent function creates a form-based interface with three input fields (ID, name, and password) that maintain their state across recompositions using remember and mutableStateOf. When a user enters information and clicks the "Add Student" button, the function first validates that essential fields are not empty. Once validated, it creates a new Student object with the provided information, then leverages Kotlin coroutines to perform the database insertion asynchronously on the IO dispatcher, preventing UI thread blocking. This approach ensures the application remains responsive during database operations. After successfully adding the student to the database (via the StudentsViewModel), the function logs the action for debugging purposes and resets the input fields to prepare for another entry. The entire interface is arranged in a Column with centred alignment, providing a clean, user-friendly experience for teachers managing their student roster.

```

1  @Composable
2  fun AddStudent(studentViewModel: StudentsViewModel) {
3      // State variables to track input field values, using remember to preserve state across recompositions
4      var studentId by remember { mutableStateOf("") }          // Tracks student ID input
5      var studentName by remember { mutableStateOf("") }        // Tracks student name input
6      var studentPassword by remember { mutableStateOf("") }    // Tracks student password input
7
8      // Form layout with centered alignment for input fields and submit button
9      Column(horizontalAlignment = Alignment.CenterHorizontally) {
10         // Input field for student ID with label
11         OutlinedTextField(
12             value = studentId,
13             onValueChange = { studentId = it }, // Update state when text changes
14             label = { Text("Student ID") }
15         )
16
17         // Input field for student name with label
18         OutlinedTextField(
19             value = studentName,
20             onValueChange = { studentName = it }, // Update state when text changes
21             label = { Text("Student Name") }
22         )
23
24         // Input field for student password with label
25         OutlinedTextField(
26             value = studentPassword,
27             onValueChange = { studentPassword = it }, // Update state when text changes
28             label = { Text("Student Password") }
29         )
30
31         // Add vertical space between input fields and button
32         Spacer(modifier = Modifier.height(16.dp))
33
34         // Button to submit the form and create a new student
35         Button(onClick = {
36             // Validate input fields are not empty before processing
37             if (studentId.isNotBlank() && studentName.isNotBlank()) {
38                 // Get current date and time to track when the student was added
39                 val currentTime = LocalDateTime.now()
40                 val formatter =
41                     DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss", Locale.getDefault())
42                 val formattedDateTime = currentTime.format(formatter)
43
44                 // Create new Student object with input values
45                 val newStudent = Student(
46                     studentId = studentId,
47                     studentName = studentName,
48                     studentPassword = studentPassword
49                 )
50
51                 // Launch coroutine to insert student in background thread to avoid blocking UI
52                 CoroutineScope(Dispatchers.IO).launch {
53                     studentViewModel.insert(newStudent) // Insert student into database
54                     Log.d(TAG, "Added Student: $newStudent") // Log success for debugging
55
56                     // Reset input fields after successful addition
57                     studentId = ""
58                     studentName = ""
59                 }
60             }
61         }) {
62             Text("Add Student") // Button label
63         }
64     }
65 }
```

# List Students

The `ListStudents` and `StudentItem` functions work in tandem to create an interactive student roster display in this Android application. `ListStudents` serves as the container function that first retrieves student data, along with their calculated average marks, from the database using Flow. This data is collected as a state using `collectAsStateWithLifecycle` to ensure the list automatically refreshes when the underlying data changes. It then implements a `LazyColumn` - Android's efficient recycling list component - to display potentially large numbers of students without performance degradation. For each student in the retrieved list, `ListStudents` delegates the rendering to the `StudentItem` function, passing both the student data and navigation controller. The `StudentItem` function creates a Material Design card for each student, displaying their name, ID, and precisely formatted average mark (to two decimal places) in an evenly spaced horizontal row. Each card is made interactive through its `onClick` property, which triggers navigation to the `StudentAttempts` screen when tapped. It cleverly replaces the route placeholder `{id}` with the actual student ID to load the appropriate quiz history. This pattern of container-item relationship exemplifies modern Android UI architecture, separating concerns related to data retrieval, list management, and item presentation.

```

1  @Composable
2  fun ListStudents(studentsViewModel: StudentsViewModel, navController: NavHostController) {
3      // Retrieve list of students with their average marks using Flow and collect as state
4      // The list automatically updates when the underlying database changes
5      // An empty list is provided as initial value to avoid null states
6      val students by studentsViewModel.getStudentsWithAverageMarks()
7          .collectAsStateWithLifecycle(emptyList())
8
9      // Create a scrollable lazy column to display the student list
10     // with appropriate padding around the content
11     LazyColumn(
12         contentPadding = PaddingValues(horizontal = 16.dp, vertical = 8.dp)
13     ) {
14         // Generate list items dynamically based on the number of students
15         // Each item in the list is a StudentItem composable representing a single student
16         items(students.size) { index ->
17             StudentItem(student = students[index], navController)
18         }
19     }
20 }
21
22 @Composable
23 fun StudentItem(student: StudentsWithAverageMark, navController: NavHostController) {
24     // Create a clickable Material Design card for displaying student information
25     // The card acts as a container that visually groups related content
26     Card(
27         modifier = Modifier
28             .fillMaxWidth() // Make the card fill the available width
29             .padding(vertical = 4.dp), // Add vertical spacing between cards
30         elevation = CardDefaults.cardElevation(defaultElevation = 2.dp), // Add subtle shadow for depth
31         onClick = {
32             // Navigate to the StudentAttempts screen when card is clicked
33             // Replace the placeholder {id} with the actual studentId in the route
34             navController.navigate(TeacherDashboardScreen.StudentAttempts.route.replace("{id}", student.studentId))
35         }
36     ) {
37         // Arrange student details in a horizontal row with equal spacing
38         Row(
39             modifier = Modifier
40                 .fillMaxWidth() // Make row fill the card width
41                 .padding(16.dp), // Add internal padding for content
42             horizontalArrangement = Arrangement.SpaceBetween, // Distribute items evenly across the row
43             verticalAlignment = Alignment.CenterVertically // Center items vertically
44         ) {
45             Text(text = student.studentName) // Display student name
46             Text(text = student.studentId) // Display student ID
47             Text(text = String.format("%.2f", student.averageMark), fontSize = 16.sp) // Display formatted average mark with 2 decimal places
48         }
49     }
50 }
51

```

codesnap.dev

## List Student's Attempts

Each time a user clicks on a student, the app should navigate to another screen that shows all the student's attempts (see line 34 above).

Now, let's iterate through all the students retrieved from the view model and place them in a `LazyColumn`.

```
@Composable
fun StudentsAttempts(quizAttemptViewModel: QuizAttemptViewModel, studentId: String) {
    // Fetch all quiz attempts for the specified student ID and collect them as a state
    // that updates when the underlying flow emits new data.
    // The initial value is an empty list to avoid null states.
    val quizAttempts by quizAttemptViewModel.getQuizAttemptByStudentId(studentId).collectAsStateWithLifecycle(emptyList())

    // Create a scrollable lazy column to display the list of quiz attempts
    // with horizontal and vertical padding for better visual spacing
    LazyColumn(
        contentPadding = PaddingValues(horizontal = 16.dp, vertical = 8.dp)
    ) {
        // Generate list items dynamically based on the number of quiz attempts
        // Each item in the list is a QuizItem composable that displays a single quiz attempt
        items(quizAttempts.size) { index ->
            QuizItem(quizAttempt = quizAttempts[index])
        }
    }
}
```

codesnap.dev

For each quiz attempt, use the following function to present it.

```
1 @Composable
2 fun QuizItem(quizAttempt: QuizAttempt){
3     // Create a Material Design card to display quiz attempt information
4     // The card has a consistent appearance and provides visual separation between list items
5     Card(
6         modifier = Modifier
7             .fillMaxWidth()      // Make the card fill the available width
8             .padding(vertical = 4.dp), // Add vertical spacing between cards
9             elevation = CardDefaults.cardElevation(defaultElevation = 2.dp), // Add subtle shadow for depth
10    ){
11        // Arrange quiz attempt details in a horizontal row with equal spacing
12        Row(
13            modifier = Modifier
14                .fillMaxWidth() // Make row fill the card width
15                .padding(16.dp), // Add internal padding for content
16                horizontalArrangement = Arrangement.SpaceBetween, // Distribute items evenly across the row
17                verticalAlignment = Alignment.CenterVertically // Center items vertically
18        ) {
19            Text(text = quizAttempt.quizId) // Display quiz identifier
20            Text(text = quizAttempt.quizDate) // Display quiz date
21            Text(text = String.format("%.2f", quizAttempt.finalMark), fontSize = 16.sp) // Display formatted mark with 2 decimal places
22        }
23    }
24 }
```

codesnap.dev

## Part 7: MainActivity class

The MainActivity.kt file serves as the entry point for this Android classroom application, implementing a dual-authentication system for both staff (Teachers) and students. It establishes a declarative UI that creates an intuitive login flow. It initialises the necessary ViewModels that manage student data persistence and state, while applying the application's theme through Week8\_classroomTheme.

At its core, the application presents users with two distinct login paths, displayed prominently via buttons. When a user selects either the staff or student login option, the NavHostController navigates to the appropriate authentication screen, maintaining state through the navigation process. For student authentication, the application connects to a database through the StudentsViewModel to validate credentials, while staff authentication uses hardcoded credentials (admin/admin) for simplicity. Upon successful authentication, the application launches the appropriate dashboard activity (TeacherDashboard or StudentsDashboard) using Android's Intent system, effectively transitioning the user to their role-specific interface.

```
1 sealed class MainActivityScreen(val route: String) {  
2     object StaffLogin : MainActivityScreen("staff_login")  
3     object StudentLogin : MainActivityScreen("student_login")  
4 }  
5
```

codesnap.dev

```
1 class MainActivity : ComponentActivity() {
2     val _context = this
3
4     override fun onCreate(savedInstanceState: Bundle?) {
5         super.onCreate(savedInstanceState)
6         enableEdgeToEdge()
7         setContent {
8             // Initialize the StudentsViewModel using ViewModelProvider with a factory pattern
9             // This allows the ViewModel to survive configuration changes and maintain state
10            val studentViewModel: StudentsViewModel = ViewModelProvider(
11                this, StudentsViewModel.StudentsViewModelFactory(this@MainActivity)
12            )[StudentsViewModel::class.java]
13
14            // Apply the app's theme to ensure consistent styling across the UI
15            Week8_classroomTheme {
16                // Scaffold provides the basic material design visual layout structure
17                // It fills the entire screen and handles the app's main content area
18                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding -
19                    // Display the MainScreen composable, passing the necessary padding from the Scaffold
20                    // and the studentViewModel for data access
21                    MainScreen(
22                        modifier = Modifier.padding(innerPadding),
23                        studentViewModel = studentViewModel
24                    )
25                }
26            }
27        }
28    }
29 }
```

codesnap.dev

The MainScreen composable function serves as the central user interface orchestrator for the application's authentication system, elegantly dividing the screen into two vertically stacked sections with a weighted distribution. In the top half, it presents users with a clear choice between two prominent login paths—Staff and Student—implemented as Material Design buttons that occupy 80% of the screen width and are horizontally centred for optimal visual balance. When a user interacts with either button, the function leverages a navigation controller (created via rememberNavController) to transition to the appropriate authentication screen seamlessly. The bottom half of the display is dedicated to the NavHostSection composable, which dynamically renders either the StaffLoginScreen or StudentLoginScreen based on the user's selection, maintaining state throughout the process. This dual-section approach creates a consistent visual hierarchy that separates navigation choices from form input, enhancing usability while efficiently managing screen real estate. The function receives and passes along the studentViewModel to child composables, ensuring that authentication processes have access to the necessary data operations without requiring redundant initialisation of database connections.

```
1 @Composable
2 fun MainScreen(modifier: Modifier = Modifier, studentViewModel: StudentsViewModel) {
3     // Create a navigation controller to handle navigation between different screens
4     val navController = rememberNavController()
5
6     // Main container that organizes the UI vertically and fills the screen
7     Column(modifier = modifier.fillMaxSize()) {
8         // Top section containing the login option buttons
9         // Takes up half the screen space (weight = 1f)
10        Column(
11            modifier = Modifier
12                .fillMaxWidth()
13                .weight(1f),
14            horizontalAlignment = Alignment.CenterHorizontally, // Center content horizontally
15            verticalArrangement = Arrangement.Top           // Arrange content from top to bottom
16        ) {
17
18            Spacer(modifier = Modifier.height(20.dp)) // Add vertical space for visual separation
19
20
21            // Staff login button - navigates to the staff login screen when clicked
22            Button(
23                onClick = {
24                    navController.navigate(MainActivityScreen.StaffLogin.route)
25                },
26                modifier = Modifier.fillMaxWidth(0.8f) // Button takes up 80% of screen width
27            ) {
28                Text(
29                    text = "Staff Login",
30                    modifier = Modifier.fillMaxWidth(),
31                    textAlign = androidx.compose.ui.text.style.TextAlign.Center
32                )
33            }
34            Spacer(modifier = Modifier.height(20.dp)) // Add vertical space for visual separation
35
36            // Student login button - navigates to the student login screen when clicked
37            Button(
38                onClick = {
39                    navController.navigate(MainActivityScreen.StudentLogin.route)
40                },
41                modifier = Modifier.fillMaxWidth(0.8f) // Button takes up 80% of screen width
42            ) {
43                Text(
44                    text = "Student Login",
45                    modifier = Modifier.fillMaxWidth(),
46                    textAlign = androidx.compose.ui.text.style.TextAlign.Center
47                )
48            }
49        }
50    }
51    // Bottom section containing the navigation host
52    // Takes up the remaining screen space (weight = 1f)
53    NavHostSection(
54        navController = navController,
55        studentViewModel = studentViewModel,
56        modifier = Modifier.weight(1f)
57    )
58 }
59
60 }
```

The NavHostSection composable function serves as the critical navigation infrastructure for the application's authentication system, acting as a container that dynamically renders different login screens based on user selection. It accepts three key parameters: a NavHostController that manages navigation state, the StudentsViewModel that provides database access for student authentication, and an optional Modifier for layout customisation. At its core, the function initialises a NavHost component with StaffLogin.route as the default starting destination, ensuring users see the staff login interface first when the application launches. Within this NavHost, the function defines navigation routes through two composable destinations that map to distinct UI screens:

MainActivityScreen.StaffLogin.route connects to the StaffLoginScreen composable, which handles administrator authentication using hardcoded credentials, while

MainActivityScreen.StudentLogin.route links to the StudentLoginScreen composable, passing along the essential studentsViewModel that enables database verification of student credentials. When navigation events are triggered by button clicks in the MainScreen's top section, the NavHostSection responds by replacing its content with the appropriate login screen, maintaining the application's state and providing a seamless transition between different authentication paths without requiring full-screen redraws.

```
1 @Composable
2 fun NavHostSection(
3     navController: NavHostController, // Navigation controller to handle screen transitions
4     studentViewModel: StudentsViewModel, // ViewModel to access student data
5     modifier: Modifier = Modifier // Optional modifier for customizing layout
6 ) {
7     // Set up navigation host to manage different login screens
8     NavHost(
9         navController = navController, // Controller that handles navigation events
10        startDestination = MainActivityScreen.StaffLogin.route, // Initial screen is staff login
11        modifier = modifier // Apply any modifiers passed to this composable
12    ) {
13        // Define the Staff Login screen destination
14        composable(MainActivityScreen.StaffLogin.route) {
15            StaffLoginScreen() // Display the staff login interface
16        }
17        // Define the Student Login screen destination
18        composable(MainActivityScreen.StudentLogin.route) {
19            StudentLoginScreen(studentsViewModel = studentViewModel) // Display student login with required ViewModel
20        }
21    }
22 }
23 }
```

codesnap.dev

```
1 @Composable
2 fun StudentLoginScreen(studentsViewModel: StudentsViewModel = viewModel()) {
3     // State variables to track and remember user input across recompositions
4     val studentId = androidx.compose.runtime.remember { mutableStateOf("") } // Stores student ID input
5     val password = androidx.compose.runtime.remember { mutableStateOf("") } // Stores password input securely
6     var isLoggedIn = remember { mutableStateOf(false) } // Tracks authentication status
7     val _context = LocalContext.current // Access to the current Android context for Toast and Intent
8
9     // Main container for login form with centered alignment
10    Column(
11        modifier = Modifier
```

```

11     modifier = modifier
12         .fillMaxSize()
13         .padding(16.dp),
14     horizontalAlignment = Alignment.CenterHorizontally,
15     verticalArrangement = Arrangement.Center
16 ) {
17     // Login screen title with appropriate typography and spacing
18     Text(
19         text = "Student Login",
20         modifier = Modifier.padding(bottom = 24.dp),
21         style = androidx.compose.material3.MaterialTheme.typography.headlineMedium
22     )
23
24     // Input field for student ID with label
25     TextField(
26         value = studentId.value,
27         onValueChange = { studentId.value = it }, // Update state when text changes
28         label = { Text("Student ID") },
29         modifier = Modifier
30             .fillMaxWidth()
31             .padding(bottom = 16.dp)
32     )
33
34     // Password input field with secure visual transformation (shows dots instead of characters)
35     TextField(
36         value = password.value,
37         onValueChange = { password.value = it }, // Update state when password changes
38         label = { Text("Password") },
39         visualTransformation = PasswordVisualTransformation(), // Hide password characters
40         modifier = Modifier.fillMaxWidth()
41     )
42
43     val context = LocalContext.current
44     val rememberusername = remember { studentId.value }
45     val rememberpassword = remember { password.value }
46
47     // Login button that triggers authentication
48     Button(onClick = {
49         // Verify credentials against database
50         isLoggedIn.value = isAuthorized(studentId.value, password.value, studentsViewModel)
51
52         if (isLoggedIn.value) {
53             // If login successful, store student ID in auth manager for session tracking
54             AuthManager.login(studentId.value)
55
56             // Show success message with student ID
57             Toast.makeText(context, "Login Successful ${studentId.value}", Toast.LENGTH_SHORT)
58                 .show()
59
60             // Create and start the Student Dashboard activity
61             val intent = Intent(context, StudentsDashboard::class.java)
62             context.startActivity(intent)
63
64         } else {
65             // Show error message for failed authentication
66             Toast.makeText(context, "Invalid Credentials", Toast.LENGTH_SHORT).show()
67         }
68
69     }, modifier = Modifier.padding(top = 24.dp)) {
70         Text("Login") // Button label
71     }
72 }
73 }
74

```

The `isAuthorized` function serves as the critical authentication gateway for the student login process, employing a straightforward yet effective credential validation mechanism. Taking three parameters—`studentId`, `password`, and `studentsViewModel`—this function bridges the UI layer with the

underlying database operations. When invoked during login attempts, it first declares but doesn't use a flow for all students (a vestigial line that could be optimised away), then proceeds to the core validation process. The function employs Kotlin's runBlocking coroutine builder to transform the asynchronous database query into a synchronous operation, temporarily suspending the current thread. Within this synchronous block, it calls studentsViewModel.getStudentById(studentId) to retrieve the specific student record matching the provided ID. After obtaining the student record, the function performs two sequential validation checks: first, it confirms that the student exists in the database (returning false if the student is not found), and then it compares the provided password with the stored student password using direct string equality (not using secure hashing, which is advisable for production applications). Only when both conditions are satisfied does the function return true; otherwise, it defaults to false, effectively creating a simple yet functional authentication mechanism that integrates with the application's Room database architecture.

```
1 /**
2  * Validates student credentials against the database
3  * @param studentId The student ID to authenticate
4  * @param password The password to validate
5  * @param studentsViewModel ViewModel to access student data
6  * @return Boolean indicating whether authentication was successful
7 */
8
9 fun isAuthorized(
10     studentId: String,
11     password: String,
12     studentsViewModel: StudentsViewModel
13 ): Boolean {
14     var allStudents: Flow<List<Student>> = studentsViewModel.allStudents // Get access to all students (unused)
15     var aStudent: Student
16
17     // Use blocking call to retrieve student by ID synchronously
18     runBlocking {
19         var aFlowStudent: Student = studentsViewModel.getStudentById(studentId)
20         // Commented code: if(aFlowStudent==null) return false
21         aStudent = aFlowStudent
22     }
23
24     // Authentication checks
25     if (aStudent == null) return false // Student ID doesn't exist
26     if (aStudent.studentPassword != password) return false // Password doesn't match
27
28     return true // Authentication successful
29 }
```

codesnap.dev

```
1 @Composable
2 fun StaffLoginScreen() {
3     // State for username and password
4     val username = androidx.compose.runtime.remember { mutableStateOf("") }
5     val password = androidx.compose.runtime.remember { mutableStateOf("") }
6
7     Column(
8         modifier = Modifier
```

```
9         .fillMaxSize()
10        .padding(16.dp),
11        horizontalAlignment = Alignment.CenterHorizontally,
12        verticalArrangement = Arrangement.Center
13    ) {
14        // Title
15        Text(
16            text = "Staff Login",
17            modifier = Modifier.padding(bottom = 24.dp),
18            style = androidx.compose.material3.MaterialTheme.typography.headlineMedium
19        )
20
21        // Username TextField
22        TextField(
23            value = username.value,
24            onValueChange = { username.value = it },
25            label = { Text("Username") },
26            modifier = Modifier
27                .fillMaxWidth()
28                .padding(bottom = 16.dp)
29        )
30
31        // Password TextField
32        TextField(
33            value = password.value,
34            onValueChange = { password.value = it },
35            label = { Text("Password") },
36            modifier = Modifier.fillMaxWidth()
37        )
38
39        // Login Button
40
41        val context = androidx.compose.ui.platform.LocalContext.current
42        val isLoggedIn = remember { true }
43        val role = remember { "staff" }
44
45        Button(
46            onClick = {
47                if (username.value == "admin" && password.value == "admin") {
48                    val intent = Intent(context, TeacherDashboard::class.java)
49                    context.startActivity(intent)
50                }
51            },
52            modifier = Modifier.padding(top = 24.dp)
53        ) {
54            Text("Login")
55        }
56    }
57 }
58
59 }
```

# **FIT2081 Week 9**

## **App Architecture (MVVM)**

# Week 9 - Lecture Reading Materials

---

## Introduction

Welcome to our lecture on App Architecture in Android development. Last week, we explored the Coroutines & Async Programming. Today, we'll learn the best practice of organising the app's codebase into layers or components that address different aspects of the app's functionality without them interfering with each other.

Consider our previous fitness app example: when you want to expand your app with more and more screens, features, and functionalities, have you wondered where you should put the new code? You might have already realised it is not a good practice to combine all codes in one file or put everything relevant to one screen into one Activity class. But do you understand why it is a bad idea? Assuming you need to access and manipulate data from different data sources (the Room database and online APIs), do you know which module/component should take care of the data operations? You might have had a sneak peek in the previous weeks of how and why we separate the components in the Room database (Data Entity, DAO, and Database) and Retrofit API calls (Retrofit interface, object, repository, etc).

In Android development, you deal with **lots of moving parts** — Activities, Fragments, network calls, databases, business rules, and more. Without a clear structure, all of this logic might end up **jammed into one place** (usually Activities). That's where the trouble starts.

A proper architecture gives you a **blueprint** for organizing your code. It defines **where each type of logic belongs**, so:

- UI logic goes in Activities/Fragments
- User actions and state management go in ViewModels (in MVVM)
- Data operations go in Repositories or Use Cases
- External data (API or local DB) is accessed through Data Sources

This **clear separation** means your codebase stays **organized and predictable** as it grows.

## Learning Objectives

By the end of this lecture, you will be able to:

- understand the principle of Separation of Concerns (SoC);
- understand the recommended architecture for Android apps and what the layers are in the architecture;
- understand the pattern of using the repository;
- understand MVVM architecture

- refactor an existing Android project to MVVM

# Separation of Concerns (SoC) in Mobile Development

## What is SoC?

In software engineering, **Separation of Concerns (SoC)** is a fundamental design principle that promotes dividing an application into distinct sections, each responsible for a specific aspect or functionality. The idea is simple yet powerful: each part of your application should only be concerned with one thing, and that concern should be entirely encapsulated within that part. This principle helps developers manage complexity by avoiding the entanglement of different functionalities, leading to more maintainable, testable, and scalable code.

To illustrate this concept in a non-technical context, consider a restaurant. In a well-organised restaurant, the chef focuses on preparing food, the waiter serves customers, and the cashier handles payments. Each role has a clear responsibility, and none interferes with others' duties. If everyone tried to do everything, chaos would ensue. Similarly, in mobile app development, user interface code should focus solely on how things look and how users interact with them. It should not be mixed with data storage logic or business rules.

## The Importance of SoC in Mobile Applications

Mobile applications can become quite complex, especially as they grow in features and scale. The code can quickly become tangled without proper separation, making it difficult to maintain, debug, or extend. When UI logic, data management, and business rules are all mixed in the same files or classes, even small changes can lead to unintended consequences elsewhere in the app. By adhering to SoC, developers can isolate changes to specific areas of the codebase, reducing the risk of introducing bugs and improving collaboration among team members.

For example, consider a scenario where you need to change how data is fetched from a remote server. If your data-fetching logic is tightly coupled with UI components like Activities or Fragments, you might have to modify multiple parts of the application just to make this change. However, if data handling is properly separated, you can update the data layer without touching the UI code at all.

## Common Layers in Mobile Development

In Android development, SoC is often achieved by structuring the application into three main layers: the **View Layer**, the **Logic Layer**, and the **Data Layer**.

- The **View Layer** includes Activities, Fragments, or Composables that are responsible solely for presenting data to the user and handling user interactions.
- The **Logic Layer**, often represented by ViewModels in modern Android architecture, acts as a mediator between the View and the Data layers. It handles UI-related logic, processes user actions, and prepares data for display.
- The **Data Layer** is responsible for managing data sources, which may include local databases,

remote APIs, or file storage. This layer abstracts away the details of data retrieval and storage from the rest of the application.

## A Simple Kotlin Example Demonstrating SoC

To make this more concrete, let's look at a simple example. Suppose we are developing an app that displays the name of a user. Without applying SoC, one might write all the code directly inside an Activity. For instance:

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        val textView = findViewById<TextView>(R.id.textView)  
        val userName = "Alice"  
        textView.text = userName  
    }  
}
```

 In this case, the UI, the data (userName), and the logic are all handled together, which violates SoC.

Now, applying SoC using a basic MVVM pattern

### Data Layer:

```
data class User(val name: String)
```

### Logic Layer (ViewModel):

```
class UserViewModel : ViewModel() {  
    var userName by mutableStateOf("")  
    private set  
  
    fun loadUser() {  
        val user = User("Alice")  
        userName = user.name  
    }  
}
```

### View Layer:

```
@Composable  
fun UserScreen(viewModel: UserViewModel = viewModel()) {  
    val name = viewModel.userName  
  
    LaunchedEffect(Unit) {  
        viewModel.loadUser()  
    }
```

```
}

Column(
    modifier = Modifier
        .fillMaxSize()
        .padding(16.dp),
    verticalArrangement = Arrangement.Center,
    horizontalAlignment = Alignment.CenterHorizontally
) {
    Text(text = "User Name: $name", fontSize = 24.sp)
}
}
```

 In this improved version, the data is defined separately as a model, the logic for managing and providing the data resides in the ViewModel, and the Activity is only responsible for observing data changes and updating the UI.

## Benefits of Applying Separation of Concerns

By following SoC in mobile development, applications become more modular and easier to understand. Each part of the app can be developed, tested, and maintained independently. For instance, developers can test the ViewModel without involving the UI or data sources. It also simplifies the process of introducing new features or modifying existing ones. When different concerns are properly separated, making changes in the data retrieval method (e.g., switching from a REST API to GraphQL) does not impact how the UI is rendered.

Moreover, SoC supports better collaboration in development teams. Front-end developers can focus on designing and refining the UI, while back-end developers or data engineers work on optimizing data storage or retrieval mechanisms. This clear division of responsibilities reduces conflicts and allows parallel development.

## Conclusion

Separation of Concerns is a vital principle in mobile development that promotes clean, maintainable, and scalable code. By structuring your app into distinct layers such as View, Logic, and Data, you ensure that each part of your application has a single responsibility. This makes it easier to develop, test, and extend your application over time. Whether you're building small apps or large-scale systems, adhering to SoC will help you manage complexity effectively and build better software.

# Model-View-ViewModel (MVVM) Architecture

## What is MVVM?

In modern Android development, the **Model-View-ViewModel (MVVM)** architectural pattern has become a widely adopted approach for building well-structured, maintainable, and testable mobile applications. MVVM is designed to help developers achieve **Separation of Concerns (SoC)** by clearly dividing the responsibilities of different parts of an application. Each component of MVVM – Model, View, and ViewModel – has a specific role, allowing developers to write modular code that is easier to manage and evolve.

The MVVM pattern is particularly well-suited to Android development due to its compatibility with Android's lifecycle-aware components such as **LiveData**, **ViewModel**, and **Data Binding**. By using MVVM, developers can ensure that their applications remain robust and responsive, even as complexity increases.

## The Role of Each Component in MVVM

**In MVVM, each component plays a distinct role:**

### **Model:**

The Model represents the **data layer** of the application. It is responsible for **managing the data and business logic**. This can include retrieving data from a local database, making network calls, or applying business rules. The Model is entirely unaware of the user interface and should only focus on the data it provides.

For example, in a typical Android app, the Model might involve data classes, database entities, and repositories that fetch data from remote servers or local storage.

### **View:**

The View refers to the **user interface (UI)** components of the application. In Android, this includes Activities, Fragments, or Jetpack Compose UI elements. The View's main responsibility is to **display data to the user and handle basic interactions** like button clicks or text inputs. However, the View should not contain complex logic; instead, it should rely on the ViewModel to provide the data and handle user actions.

### **ViewModel:**

The ViewModel acts as a **bridge** between the View and the Model. **It holds UI-related data and UI logic** that is needed by the View, but it does not reference any UI elements directly. The ViewModel is lifecycle-aware, meaning it survives configuration changes like screen rotations. It often uses LiveData or StateFlow to expose data to the View, enabling automatic updates whenever the data changes.

# A Practical Example of MVVM in Kotlin

Let's consider a simple example where we want to build an app that displays a list of users fetched from a data source.

## **Model Layer:**

This layer includes the `User` data class and a simple repository to provide user data.

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

class UserRepository {
    fun getUsers(): List<User> {
        // Simulated data source, could be replaced with API or database.
        return listOf(
            User(1, "Alice"),
            User(2, "Bob"),
            User(3, "Charlie")
        )
    }
}
```

## **ViewModel Layer:**

The ViewModel fetches data from the repository and exposes it via **StateFlow** for Compose to observe.

```
class UserViewModel : ViewModel() {
    private val repository = UserRepository()
    private val _users = MutableStateFlow<List<User>>(emptyList())
    val users: StateFlow<List<User>> = _users.asStateFlow()

    fun loadUsers() {
        // In a real app, this would likely be asynchronous.
        _users.value = repository.getUsers()
    }
}
```

## **View Layer:**

The UI collects from `users` StateFlow and displays a list of user names in Compose.

```
@Composable
fun UserListScreen(viewModel: UserViewModel = viewModel()) {
    val userList by viewModel.users.collectAsState()

    LaunchedEffect(Unit) {
        viewModel.loadUsers()
    }
}
```

```

Column(
    modifier = Modifier
        .fillMaxSize()
        .padding(16.dp)
) {
    Text(text = "User List", fontSize = 24.sp, modifier = Modifier.padding(bottom = 16.dp))

    userList.forEach { user ->
        Text(text = "${user.name}", fontSize = 18.sp, modifier = Modifier.padding(4.dp))
    }
}

```

 In this example, the **Model** provides the raw data, the **ViewModel** handles the retrieval and preparation of this data, and the **View** displays it. The View and ViewModel communicate through LiveData, ensuring that the UI updates automatically when the data changes.

## Advantages of Using MVVM

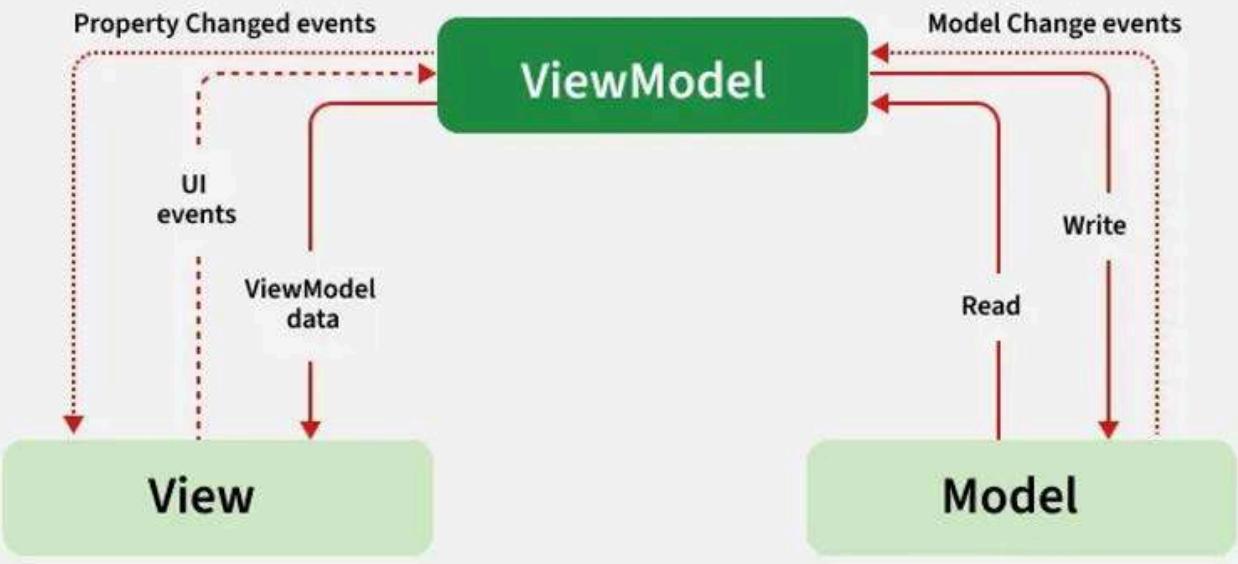
The MVVM pattern offers several advantages that make it particularly suitable for Android development. Firstly, it allows for **clear separation** between the UI and business logic, which simplifies both development and maintenance. The ViewModel's lifecycle-awareness means that data is preserved during configuration changes, improving user experience. Additionally, because ViewModels do not reference UI components directly, they are **easier to test** in isolation, which leads to more robust applications.

Moreover, MVVM supports **reactive programming** through LiveData or StateFlow, enabling the UI to automatically react to changes in the underlying data. This results in less boilerplate code and a more responsive UI.

## Real-world Connection

Major professional apps like Instagram, Netflix, and Google Maps use MVVM or similar architectures. These patterns allow large teams to **work on different components simultaneously** and make it easier to add features or fix bugs without breaking existing functionality.

## Data Flow in MVVM



User Action --> View --> ViewModel --> Repository/Model --> Data



UI Updates (LiveData / StateFlow)

## Conclusion

The **Model-View-ViewModel** architecture is a powerful design pattern that enables Android developers to write clean, maintainable, and scalable applications. By separating the data, logic, and UI into distinct components, MVVM promotes better code organization and easier testing. Through the use of lifecycle-aware components like ViewModel and LiveData, developers can build responsive and robust mobile apps that handle configuration changes gracefully. As mobile applications continue to evolve, adopting MVVM ensures that your codebase remains flexible, testable, and ready for growth.

# ViewModel Recap, LiveData vs StateFlow

As part of the **MVVM architecture**, the **ViewModel** plays a crucial role in managing UI-related data in a lifecycle-conscious way. Understanding the ViewModel is fundamental for building robust and responsive Android applications. Alongside the ViewModel, tools like **LiveData** and **StateFlow** allow developers to create dynamic and reactive user interfaces by efficiently handling data updates. This section will provide a comprehensive recap of the ViewModel, followed by an introduction to LiveData and StateFlow, explaining when and how to use each.

## ViewModel Recap: Lifecycle-Aware UI Logic

The **ViewModel** is a class designed to store and manage UI-related data for an Android component, such as an Activity or Fragment. One of its most significant advantages is that it survives configuration changes, such as screen rotations, which would otherwise cause data loss if stored directly in UI components.

Unlike Activities or Fragments, which are tied closely to the Android lifecycle, the ViewModel exists independently and persists as long as the scope of its lifecycle owner allows. This means that data stored in a ViewModel remains available across re-creations of the UI. Moreover, the ViewModel is not responsible for holding references to UI elements, ensuring that memory leaks are avoided.

For example, instead of saving a list of users directly in an Activity, the list is maintained in a ViewModel. The Activity observes the ViewModel, and whenever the data changes, the UI is updated automatically.

## LiveData: Observable and Lifecycle-Aware Data

LiveData is a data holder class that can be observed within a given lifecycle. It allows UI components to observe changes in data, ensuring that updates are only sent when the observer is in an active state (such as STARTED or RESUMED). This prevents memory leaks and unnecessary UI updates when the app is not in the foreground.

LiveData is particularly useful for handling simple reactive patterns in MVVM, where the View observes LiveData exposed by the ViewModel. When the data changes, the observer (usually the Activity or Fragment) is notified and can update the UI accordingly.

Here is a simple example of LiveData in use:

**ViewModel with LiveData:**

```

class UserViewModel : ViewModel() {
    private val _userName = MutableLiveData("Alice")
    val userName: LiveData<String> = _userName

    fun changeName(newName: String) {
        _userName.value = newName
    }
}

```

### **Composable with LiveData:**

```

@Composable
fun UserScreen(viewModel: UserViewModel = viewModel()) {
    val name by viewModel.userName.observeAsState("")

    Column(modifier = Modifier.padding(16.dp)) {
        Text(text = "User Name: $name", fontSize = 20.sp)
        Spacer(modifier = Modifier.height(8.dp))
        Button(onClick = { viewModel.changeName("Bob") }) {
            Text("Change Name")
        }
    }
}

```

## **StateFlow: A Coroutine-Based Alternative**

**StateFlow** is a stateholder observable flow that emits the current and new state updates to its collectors. Unlike LiveData, StateFlow does not rely on the Android lifecycle, meaning that developers must manually handle stopping and starting collection, typically using lifecycle-aware coroutine scopes like `lifecycleScope`.

StateFlow is particularly useful when dealing with continuous or streaming data, such as tracking UI state or processing asynchronous data updates from repositories.

Here is an example of StateFlow in a ViewModel:

### **ViewModel with StateFlow:**

```

class CounterViewModel : ViewModel() {
    private val _count = MutableStateFlow(0)
    val count: StateFlow<Int> = _count.asStateFlow()

    fun increment() {
        _count.value += 1
    }
}

```

### **Composable with StateFlow:**

```
@Composable
```

```
fun CounterScreen(viewModel: CounterViewModel = viewModel()) {
    val count by viewModel.count.collectAsState()

    Column(
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text(text = "Count: $count", fontSize = 24.sp)
        Spacer(modifier = Modifier.height(16.dp))
        Button(onClick = { viewModel.increment() }) {
            Text("Increment")
        }
    }
}
```

## When to Use LiveData vs. StateFlow

Both LiveData and StateFlow serve similar purposes but excel in different scenarios.

- **LiveData** is recommended for simple UI data binding, especially if you are not using coroutines extensively. It integrates seamlessly with the Android lifecycle and requires less boilerplate for basic reactive updates.
- **StateFlow** is ideal when working with Kotlin coroutines, especially in more complex applications that need advanced control over data streams, concurrency, and transformation. StateFlow also allows greater flexibility and is more suitable for unidirectional data flow architectures.

# Android App Architecture

A well-structured architecture is essential for building high-quality Android apps. It ensures better collaboration, easier maintenance, and a smoother user experience.

## What is Android App Architecture?

- It refers to the overall structure and organization of an Android application.
- A good architecture improves code readability, testability, scalability, and maintainability.

## Why Architecture Matters

- Encourages clean code separation
- Makes the app easier to test and debug
- Supports teamwork and large-scale development
- Reduces the risk of bugs when scaling or adding features

## Core Principles of App Architecture

- **Separation of Concerns (SoC)**: Divide code by responsibility
- **Single Responsibility Principle (SRP)**: Each class/function should do only one thing
- **Decoupling**: Minimize direct dependencies between components
- **Reusability and Scalability**: Structure code for reuse and growth
- **Testability**: Design components that are easy to test in isolation

## Android Recommended App Architecture

## UI Layer

UI elements



State holders

## Domain Layer (optional)

Data Layer

Repositories



Data Sources

The **UI layer** that displays application data on the screen

- **UI elements** that render the data on the screen
- **State holders** (such as **ViewModel** classes) that hold data, expose it to the UI, and handle UI logic

The **Data Layer** that contains the business logic of your app and exposes application data

- Each **Repository** can contain zero to many **Data Sources**

- Each **Data Source** class works with only one source of data (e.g., a file, a network source, or a local database)

## Android Architecture Components

Jetpack libraries that help implement architecture patterns:

- **ViewModel** – Stores UI-related data
- **LiveData / StateFlow** – Observables for UI updates
- **Room** – SQLite object mapping
- **Repository** – Abstracts data access
- **Navigation** – Manages in-app navigation

# Repository and Data Layer

The Repository and Data Layer are critical in building scalable, testable, and maintainable Android applications. They ensure that higher layers remain clean and focused on UI or business logic, while data access is handled efficiently and modularly.

## What is the Data Layer?

The Data Layer manages all data-related operations in an Android application. This includes fetching, caching, storing, and updating data from various sources (e.g., local database, remote APIs).

## Typical Data Layer Components

- **Repository**
  - Orchestrates access to local and remote sources
- **Local Data Source**
  - Room database or SharedPreferences
  - Persistent, offline-accessible data
- **Remote Data Source**
  - Retrofit, Ktor, or other HTTP clients
  - Interacts with RESTful APIs

## What is a Repository?

A Repository is a class that abstracts access to multiple data sources. It acts as a single point of truth for data and separates the data logic from the rest of the app.

## Why Use a Repository?

- Centralizes data access logic
- Decouples ViewModel and UseCases from specific data sources
- Simplifies testing and mocking
- Supports multiple data sources (e.g., cache + network fallback)
- Follows the **Single Responsibility Principle**

## Repository Responsibilities

- Expose clean APIs to the domain or presentation layers
- Decide whether to fetch from local or remote sources
- Map data models (DTOs to domain models if needed)
- Handle data transformations, error wrapping, or retry strategies

---

## \* References and Additional Reading \*

### Recommended Online Materials

- **Understanding the Model-View-ViewModel (MVVM) Pattern: A Guide for Software Developers** by Nikita @Medium
  - <https://medium.com/@nikitinsn6/understanding-the-model-view-viewmodel-mvvm-pattern-a-guide-for-software-developers-aa5ce155263c>
- **Sample MVVM architecture app** by Amit Shekhar @ Github
  - <https://github.com/amitshekhariitbhu/MVVM-Architecture-Android>

### Recommended Video Tutorials

- **MVVM in 100 seconds** by Philipp Lackner
  - <https://www.youtube.com/watch?v=-xTqfilaYow>
- **StateFlow vs. Flow vs. SharedFlow vs. LiveData** by Philipp Lackner
  - <https://www.youtube.com/watch?v=6Jc6-lNantQ>
- **ViewModels & Configuration Changes** by Philipp Lackner
  - <https://www.youtube.com/watch?v=9sqvBydNJSg>
- Example of **converting Android app to MVVM** (Youtube)
  - <https://www.youtube.com/watch?v=SlaF0cw9C8s>



MONASH  
University

**FIT2081**  
**Mobile Application**  
**Development**



# Advanced App Architecture & MVVM

**Week 9**

Jiazhou 'Joe' Liu

# Learning Objectives

- **Fundamental concepts**
  - Separation of Concerns (SoC) and Dependency Injection (DI)
  - MVVM Advanced Topics (State, UI event, Unidirectional Data Flow, State Hoisting)
  - Android App Architecture
- **Coding Demo: create ViewModel class with own lifecycle**

# Separation of Concerns (SoC) & Dependency Injection (DI)

# Separation of Concerns (SoC)

Question: What happens when you put all your code inside the `MainActivity.kt` file?

## Hard to Maintain & Scale

- As the app grows, the `Activity class` becomes a **giant file with thousands of lines of code**.
- Any change requires navigating a **huge, messy codebase**.

## Difficult to Test

- Unit testing becomes nearly **impossible** since the `Activity class` is tied to UI elements.
- You can't test logic **without launching the whole UI**, making tests **slow and unreliable**.

## Bad UI Performance & Memory Leaks

- If the `Activity class` directly **fetches data** from an API or database, it **blocks the main thread**, making the UI **laggy**.
- Holding long-lived objects (like database references) can cause **memory leaks** when the `Activity class` is destroyed.

## Crashes on Configuration Changes

- When the screen **rotates** or a **language changes**, Android **destroys and recreates** the `Activity class`.
- If the `Activity class` holds UI state (like form data), it gets lost, **frustrating the user**.



# Benefits of SoC

**Code Reusability**

**Readability**

**Testability**

**Maintainability**

**Scalability**



# App Architecture using SoC

- The **app folder** contains the main codebase for the Android app.
  - Inside `src/main/java`, the code is organised according to package names.
- The **data package** contains data-related code.
  - The `models` package includes the `Task.java` file, which defines the `Task` model class.
  - The `repositories` package contains the `TaskRepository.java` file, responsible for data operations.
- The **di package** contains files related to dependency injection.
- The **presentation package** handles UI-related code.
  - The `activities` package includes files for different activities.
  - The `adapters` package contains adapters for displaying tasks.
  - The `viewmodels` package contains the `TaskViewModel.java` file, which manages the UI logic for tasks.
- The **utils package** stores utility/helper files.

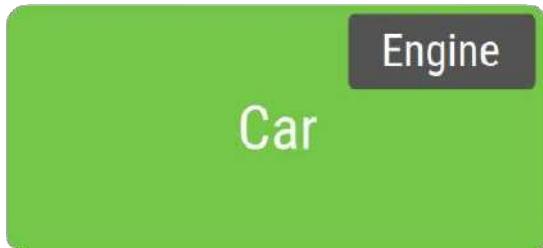
Note: In our unit, app architecture using SoC or MVVM doesn't mean that you need to strictly follow the same pattern for file organisation. SoC or MVVM focuses more on using **abstraction and different layers to decouple UI (View), presentation logic (ViewModel), and data logic (Model)**.

```
- app
- src
  - main
    - java
      - com.example.todoapp
        - data
          - models
            - Task.java
          - repositories
            - TaskRepository.java
    - di
      - AppModule.java
      - ViewModelModule.java
  - presentation
    - activities
      - HomeActivity.java
      - AddTaskActivity.java
      - TaskDetailsActivity.java
    - adapters
      - TaskAdapter.java
    - viewmodels
      - TaskViewModel.java
  - utils
    - DateHelper.java
  - App.java
- res
  - ...
```



# Classes Without Dependency Injection (DI)

```
class Car {  
  
    private val engine = Engine()  
  
    fun start() {  
        engine.start()  
    }  
  
    fun main(args: Array) {  
        val car = Car()  
        car.start()  
    }  
}
```

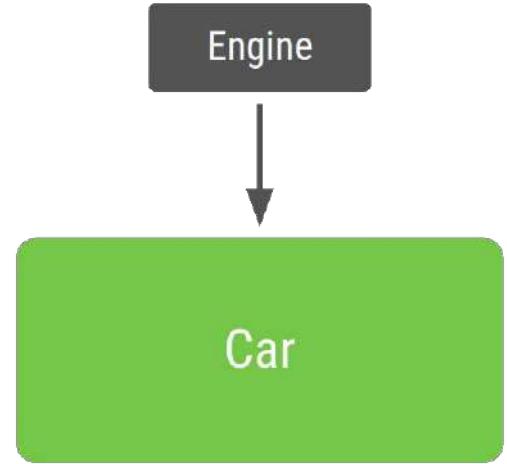


## Problems:

- **Car and Engine are tightly coupled**
  - an instance of Car uses one type of Engine, and no subclasses or alternative implementations can easily be used.
- The hard dependency on Engine **makes testing more difficult**. Car uses a real instance of Engine, thus preventing you from modifying Engine for different test cases.

# Dependency Injection (DI) Example

```
class Car(private val engine: Engine) {  
    fun start() {  
        engine.start()  
    }  
}  
  
fun main(args: Array) {  
    val engine = Engine()  
    val car = Car(engine)  
    car.start()  
}
```



**Dependency Injection (DI)** is a software **design pattern** where an object receives its dependencies from an external source rather than creating them itself

# Two Injection Methods

## Constructor Injection

```
class Car(private val engine: Engine) {  
    fun start() {  
        engine.start()  
    }  
  
    fun main(args: Array) {  
        val engine = Engine()  
        val car = Car(engine)  
        car.start()  
    }  
}
```

## Field Injection (or Setter Injection)

```
class Car {  
    lateinit var engine: Engine  
  
    fun start() {  
        engine.start()  
    }  
  
    fun main(args: Array) {  
        val car = Car()  
        car.engine = Engine()  
        car.start()  
    }  
}
```

# Manual DI vs Automated DI

**Manual DI:** you created, provided, and managed the dependencies of the different classes yourself, without relying on a library.

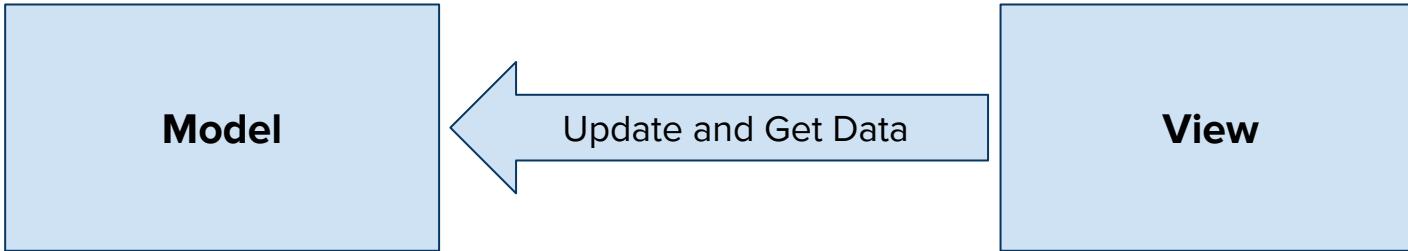
- Works well for small apps and simple code structure.
- Presents problems in other situations:
  - **Tedious coding with a large amount of boilerplate code.** E.g., A real car needs an engine, a transmission, a chassis, etc., while an engine needs cylinders and spark plugs, etc.
  - **Manually manage lifetimes** of your dependencies in memory

**Automated DI** uses libraries that solve this problem by automating the process of creating and providing dependencies.

[\*\*Dagger\*\*](#) is a popular dependency injection library for Java, Kotlin, and Android that is maintained by Google. Dagger facilitates using DI in your app by creating and managing the graph of dependencies for you.

# MVVM Architecture

# Traditional UI/App Development Limitations

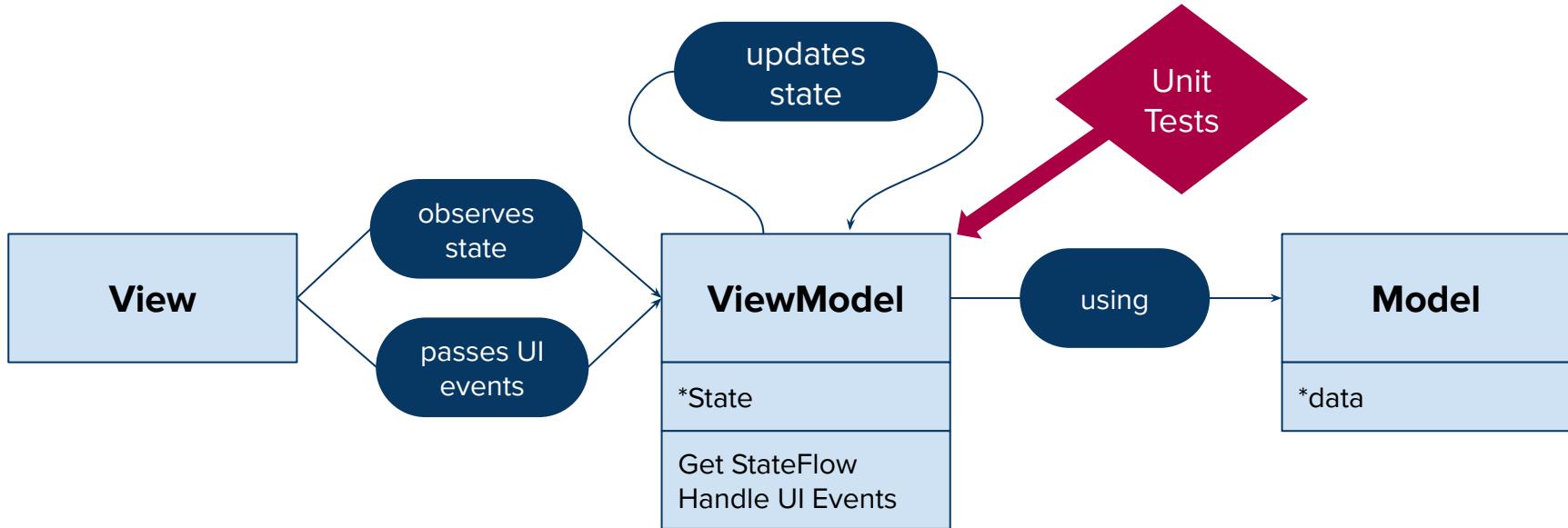


## Limitations

- Difficult to test
- Encourages using UI as data storage
- Complex classes and a huge amount of code
- No code sharing
- Strong dependency between UI & logic
- Hard to redesign UI

# Model-View-ViewModel (MVVM) Architecture

**Model-View-ViewModel (MVVM)** is a design pattern that originated from Microsoft's WPF (Windows Presentation Foundation) framework and gained popularity in Android development due to its effectiveness in handling UI-related concerns.



# State

*In a general context of Kotlin Compose:*

- **State** is a piece of data that, when it changes, automatically makes the UI update itself.

*In MVVM architecture:*

- **State** is the current data or status that the UI (View) is showing, managed by the ViewModel.
- State serves as the **ultimate source of truth**.

```
data class ArticlesUiState( // In the ViewModel
    val isLoading: Boolean = false,
    val articles: List<Article> = emptyList(),
    val errorMessage: String? = null
)
```

```
// In the View
val uiState: StateFlow<ArticlesUiState>
```

- If `isLoading = true`, the View shows a loading spinner.
- If `articles` are available, the View shows the list.
- If `errorMessage != null`, the View shows an error message.



# How ViewModel Manages State

The State is **immutable**. Because we don't want multiple places to update it without synchronization.

- You could have a **private mutable flow** so that the ViewModel can update the current state freely, and **an immutable version of it** so that the observer cannot inadvertently modify the flow.

```
private val _state: MutableStateFlow<State> = MutableStateFlow(State())
val state: StateFlow<State> = _state
```

- To update the State, you can take the current version of the state, make necessary changes, and then push it to the flow. Here's an example:

```
state.update { currentState ->
    currentState.copy(inputError = exception.message)
}
```

# ViewModel

The **ViewModel** operates independently of the **View**, which is quite beneficial. Its primary role is to **generate new states based on data or events from the Model**. The ViewModel does not have any reference to the View in any manner.

## Features

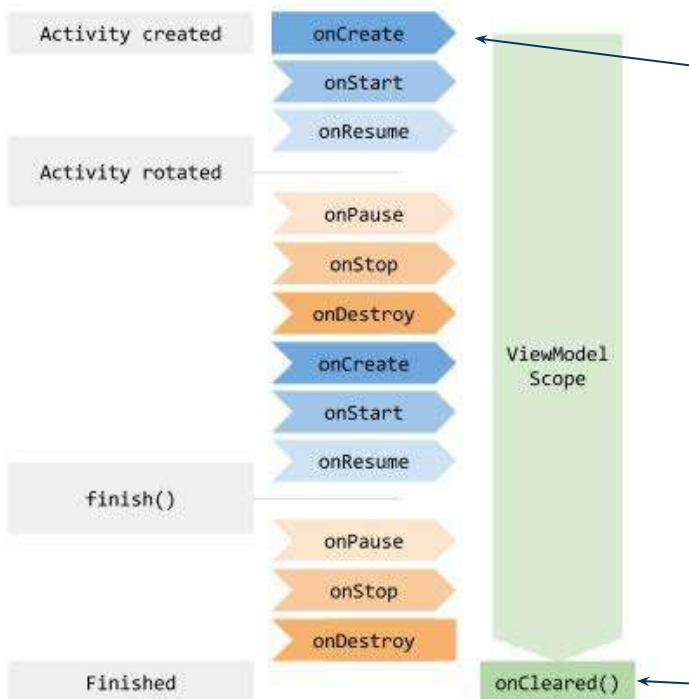
- Non-visual class encapsulates the presentation logic required.
- Can't see View (no direct reference).
- Coordinates the view's interaction with the model.
- May provide data validation and error reporting.
- View, as a observer, will get notified of any state changes if use observable data (e.g., LiveData, StateFlow, Flow)

# ViewModel for Testing

The ViewModel is completely **independent** of Android or UI specifics; its sole purpose is to respond to UI events and generate new State. This isolation makes it a perfect **black box**. Most tests will follow this pattern:

- Set up the system under test (model data, initial state).
- Trigger a UI event.
- Verify if the state update meets expectations.

# ViewModel and Lifecycle Awareness



```
KOTLIN
```

```
val myViewModel: MyViewModel = viewModel()
```

This scope will be canceled when ViewModel will be cleared, i.e `ViewModel.onCleared()` is called.

This method will be called when this ViewModel is no longer used and will be destroyed. It is useful when ViewModel observes some data and you need to clear this subscription to **prevent a leak of this ViewModel**.

```
KOTLIN
```

```
1 override fun onCleared() {  
2     super.onCleared()  
3     // Cleanup logic here  
4 }
```



# Create a ViewModel with Own Lifecycle



LIVE CODING

1. Make sure your *ViewModel* class inherit from *ViewModel* (*androidx.lifecycle*)

```
class StudentsViewModel : ViewModel() { ... }
```

2. In the View-related files, depending on where you create the instance:
  - a. In the *MainActivity* class:

```
class MainActivity : ComponentActivity() {  
    private val viewModel by viewModels<StudentsViewModel>()  
    ...  
}
```

- b. In a Compose function: (you need to add a dependency in the gradle first)

```
implementation ("androidx.lifecycle:lifecycle-viewmodel-compose:2.6.1")
```

```
val viewModel = viewModel<StudentsViewModel>()
```



MONASH  
University

# Create a ViewModel with Own Lifecycle (cont.)

With dependency from other classes, for example a context or a repository

```
class StudentsViewModel(context: Context) : ViewModel() {...}
```

Use a constructive argument either in the View (activity class or compose functions)

```
val context = LocalContext.current
val viewModel = viewModel<StudentsViewModel>(
    factory = object : ViewModelProvider.Factory {
        override fun <T : ViewModel> create(modelClass: Class<T>): T {
            return StudentsViewModel(context) as T
        }
    }
)
```

Or in the ViewModel class

```
class StudentsViewModel(context: Context) : ViewModel() {
    ...
    class StudentsViewModelFactory(context: Context) : ViewModelProvider.Factory {
        private val context = context.applicationContext
        override fun <T : ViewModel> create(modelClass: Class<T>): T =
            StudentsViewModel(context) as T
    }
}
```

# View

The **View** maintains a reference to the **ViewModel** because it must observe the **State** and transmit **UI events** to it.

Upon receiving each new **State** gathered by the **View**, the UI is updated to reflect **the current state produced by the ViewModel**.

All operations related to **UI handling** should be encapsulated within the **ViewModel**, while **business logic should reside in the Model**.

## Features

- Visual element defines the controls and their visual layout and styling.
- Can see all others components.
- Defines and handles UI visual behaviour, such as animations or transitions.

# UI Events

The View is responsible for informing the ViewModel about UI interactions.

```
fun addButtonClicked () {
    viewModelScope.launch {
        // Functionality
    }
}
```

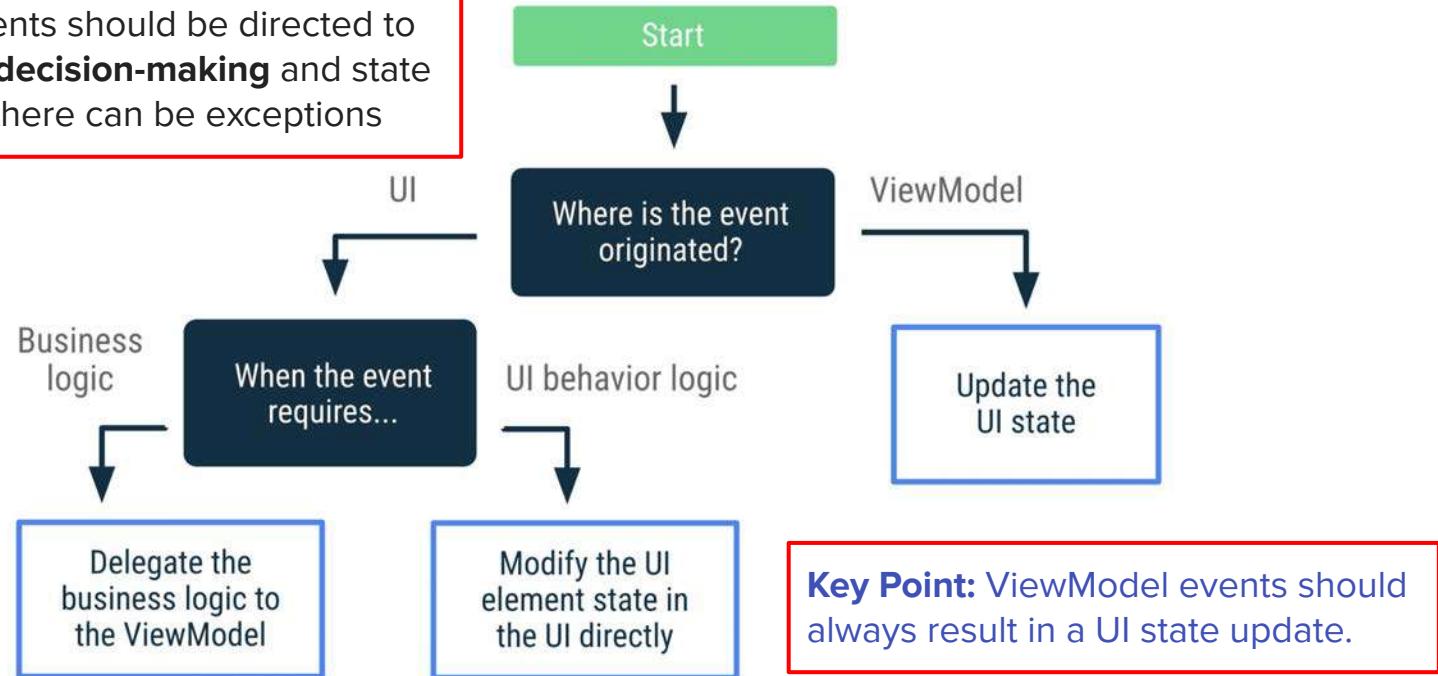
```
Button(
    onClick = {
        vm.addButtonClicked()
    },
)
```

```
sealed class Event {
    data object AddButtonClicked : Event()
    class InputChanged(val input: String) : Event()
    class RemoveItem(val item: Int) : Event()
}

// Inside ViewModel:
fun handleEvent(event: Event) {
    when (event) {
        // calling private method
        Event.AddButtonClicked -> addButtonClicked()
        is Event.InputChanged -> inputChanged( event.input )
        is Event.RemoveItem -> removeItem( event.item )
    }
}
```

# UI Events – Decision Tree

In general, all UI events should be directed to the **ViewModel for decision-making** and state updates. However, there can be exceptions



# UI Events – Navigation Events

If the **navigation event** is triggered in the UI because the user tapped on a button, the **UI takes care of that** by calling the navigation controller or exposing the event to the caller composable as appropriate.

```
@Composable
fun LoginScreen (
    onHelp: () -> Unit, // Caller navigates
    to the right screen
    viewModel: LoginViewModel = viewModel()
) {
    // Rest of the UI

    Button(onClick = onHelp) {
        Text("Get help")
    }
}
```

If the data input requires some **business logic validation** before navigating, the ViewModel would need to expose that state to the UI. The UI would react to that state change and navigate accordingly.

# Composable Previews

You shouldn't pass the *ViewModel* instances down to other composable functions.

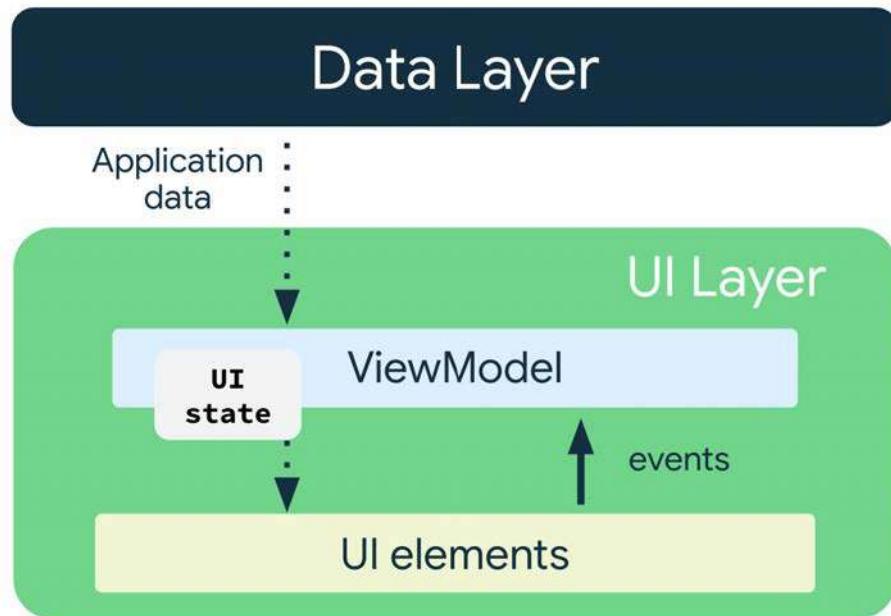
When you do this, you're coupling the composables with the *ViewModel* type, which makes them **less reusable, harder to test, and unable to preview**.

```
@Composable
fun MyComposable (viewModel: MyViewModel) {
    val state by
    viewModel.state.collectAsStateWithLifecycle()
    MyComposable(
        state = state
    )
}
@Composable
fun MyComposable (state: MyState) {
    ...
}

@Preview
@Composable
fun MyComposablePreview (state: MyState = MyState()) {
    MyComposable(state = state)
}
```

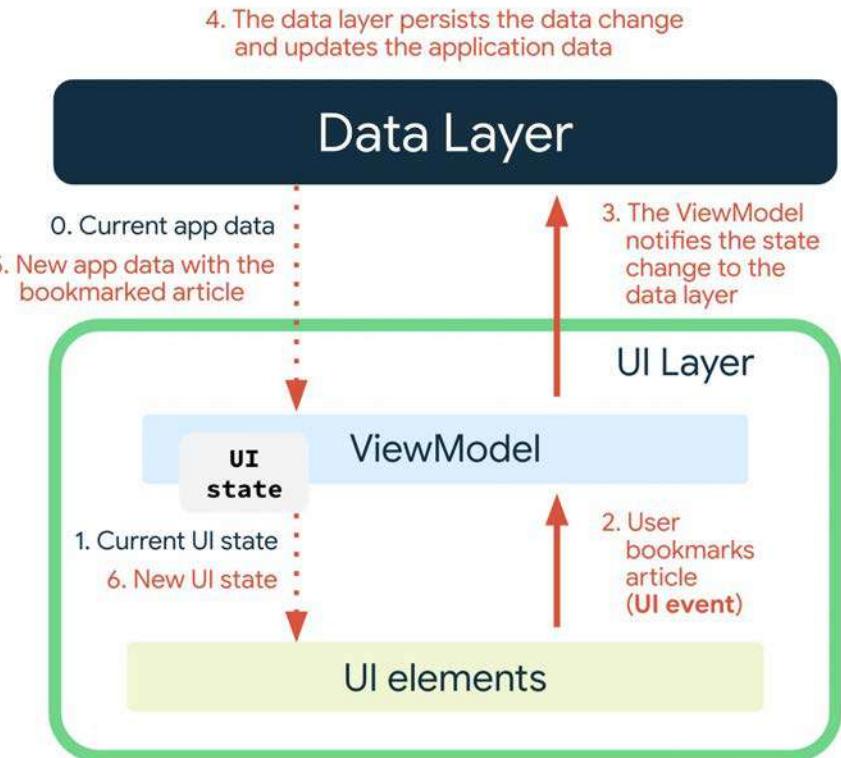
# Unidirectional Data Flow

- The ViewModel **holds and exposes** the state to be consumed by the UI. The UI state is application data transformed by the ViewModel.
- The UI **notifies** the ViewModel of user events.
- The ViewModel **handles** the user actions and **updates** the state.
- The updated state is **fed back** to the UI to render.
- The above is repeated for any event that causes a mutation of state.



# Unidirectional Data Flow (example)

*Example:* A user requesting to bookmark an article is an example of an event that can cause state mutations. As the state producer, it's the **ViewModel**'s responsibility to define all the logic required in order to populate all fields in the UI state and process the events needed for the UI to render fully.



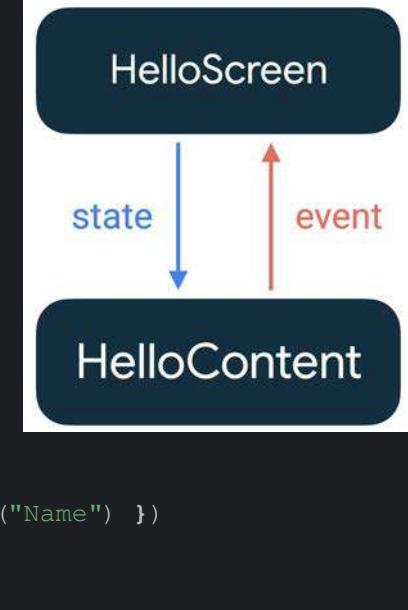
# State Hoisting & Unidirectional Data Flow

**State hoisting** in Compose is a pattern of moving state to a composable's caller to make a composable stateless.

```
@Composable
fun HelloScreen() {
    var name by rememberSaveable { mutableStateOf("") }

    HelloContent(name = name, onNameChange = { name = it })
}

@Composable
fun HelloContent(name: String, onNameChange: (String) -> Unit) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(
            text = "Hello, $name",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.bodyMedium
        )
        OutlinedTextField(value = name, onValueChange = onNameChange, label = { Text("Name") })
    }
}
```



# Model

The **Model** in **MVVM** encompasses the business logic tailored to your specific scenario.

*While the **ViewModel** manages UI-related logic exclusively, any functionality unrelated to the UI should reside within the **domain model code**.*

## Features

- Non-visual classes that encapsulate the application's data and business logic
- Can't see ViewModel or View
- Should not contain any user task-specific behaviour or application logic
- Notifies other components of any state changes
- May provide data validation and error reporting

# Android App Architecture

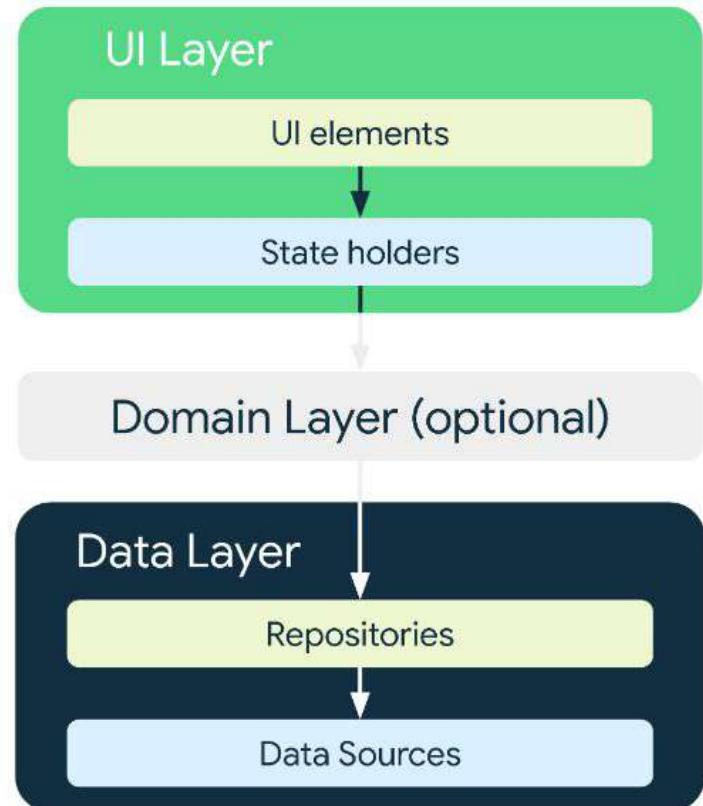
# Android Recommended App Architecture

The **UI layer** that displays application data on the screen

- **UI elements** that render the data on the screen
- **State holders** (such as **ViewModel** classes) that hold data, expose it to the UI, and handle UI logic

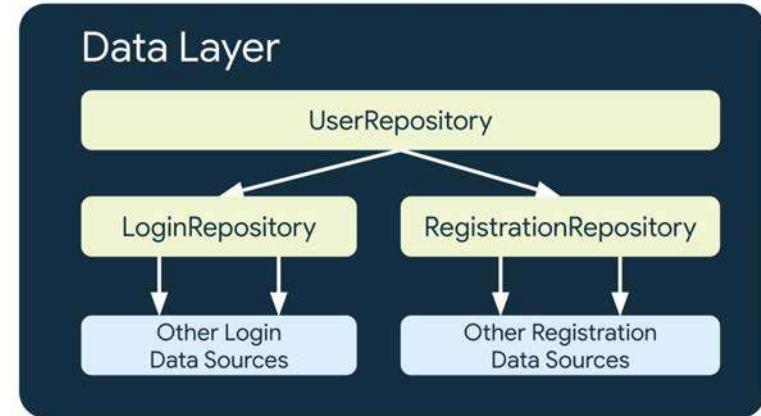
The **Data Layer** that contains the business logic of your app and exposes application data

- Each **Repository** can contain zero to many **Data Sources**
- Each **Data Source** class works with only one source of data (e.g., a file, a network source, or a local database)



# Repository

**Repository** serves as an **intermediary layer** used to abstract and centralize data operations. It's typically utilised to **encapsulate database interactions**, such as inserting, updating, deleting, and querying data.



## Unified Data Source Management

- The repository decides whether to fetch data from the Room database (local cache), a remote API, or another source.

## Hide Data Complexity:

- The ViewModel doesn't need to know if data comes from Room, an API, or a file. The repository handles the logic.

## Centralized Data Operations:

- Combines Room DAO methods, API calls, and transformations into cohesive operations.

## Thread Management:

- Ensures Room database operations (which must run off the UI thread) are executed using coroutines, RxJava, or ExecutorService.

# Reminders and Announcements

## Assignments:

- Peer Engagement Weekly Task (2%) - Due Friday 11:55 pm

## Lab activities this week

- Multi-source data application
- MVVM design
- Calling API

## Week 10

- *Advanced Compose UI & Theming*
- *Animation and Transition in Jetpack Compose*

# Reference and Further Reading

- Dependency injection in Android:  
<https://developer.android.com/training/dependency-injection>
- Android Presentation Patterns: MVVM: <https://swiderski.tech/2024-02-09-MVVM/>
- UI events: <https://developer.android.com/topic/architecture/ui-layer/events>
- Preview your UI with composable previews:  
<https://developer.android.com/develop/ui/compose/tooling/previews>
- State Hoisting: <https://developer.android.com/develop/ui/compose/state-hoisting>



# Lab: Multi-source Data + MVVM + API Integration

---

## Introduction

This week, we will focus on the following key objectives:

- Building applications that integrate data from multiple sources
- Applying the MVVM architectural pattern
- Calling and interacting with remote APIs

You will develop a basic social media-style application, similar to Twitter (now known as X). This app will communicate with a remote server to either:

- **Retrieve a list of current posts**, or
- **Generate a new post**

To simplify the task, the post content will be automatically generated by the server using predefined random data—meaning that the client side (your app) will not compose the content manually.

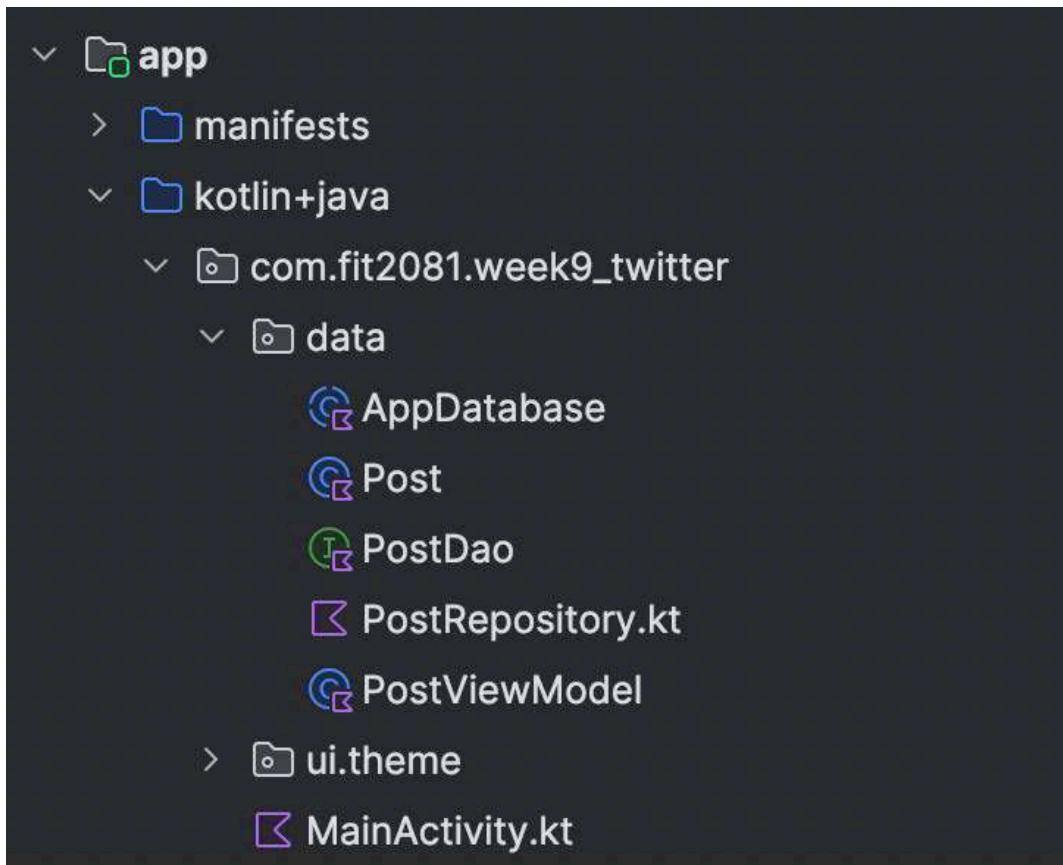
Note: The terms **Post** and **Tweet** are used interchangeably throughout this lab.

## Learning Objectives

By the end of this lab, you will be able to:

- Understand how to structure an application that interacts with multiple data sources, including remote APIs.
- Implement the MVVM architecture to separate responsibilities across the UI, ViewModel, and data layers.
- Retrieve dynamic content (posts/tweets) from a remote server via API calls.
- Trigger new post generation by sending requests to a server.
- Maintain and observe UI state changes driven by external data updates.
- Build a simple interface that displays remote content using Jetpack Compose.
- Apply these practices in the context of real-world apps that rely on server communication and data-driven UIs (e.g., social media platforms).

# App Structure – Detailed Version



**i** <!-- Add internet permission --> in your manifest file.

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

## data/

This folder contains all the core components responsible for data management. It encapsulates the **Model** and **Repository** layers of MVVM.

### 1. Post.kt

This is a **data class** that defines the structure of a Post (or Tweet). It typically includes fields such as postId, userName, subject, and content. This class represents the basic unit of data used throughout the app.

### 2. PostDao.kt

This is a **Data Access Object (DAO)** interface for interacting with the local database. It defines methods such as insertPost(), deleteAllPosts() and getAllPosts() that Room uses to generate the necessary SQL code behind the scenes. This DAO allows the app to work with **Post** objects stored

locally.

### 3. AppDatabase.kt

This class defines the **Room database** for the app. It declares the database schema and provides access to the DAOs (like `PostDao`). This is the entry point for local storage using Room.

### 4. PostRepository.kt

This is the **Repository** class that acts as the single source of truth for data. It decides whether to fetch data from the remote server or the local database and provides clean methods for the ViewModel to call. It might internally call functions like `PostDao.getAllPosts()` or trigger an API request to generate a new post.

- \* - When online, data is fetched from the remote API and cached locally
- \* - When offline, data is served from the local database

### 5. PostViewModel.kt

This class belongs to the **ViewModel** layer. It exposes state and logic to the UI and receives user actions. The ViewModel communicates with the Repository to fetch or update data and holds UI state (e.g., a list of posts). It ensures that the UI remains reactive and up-to-date.

## ui.theme/

This folder contains files related to the visual theme of the app, such as color schemes, typography, and shapes. These are mostly automatically generated when you create a Jetpack Compose project, and they control the app's appearance but do not influence logic or data.

## MainActivity.kt

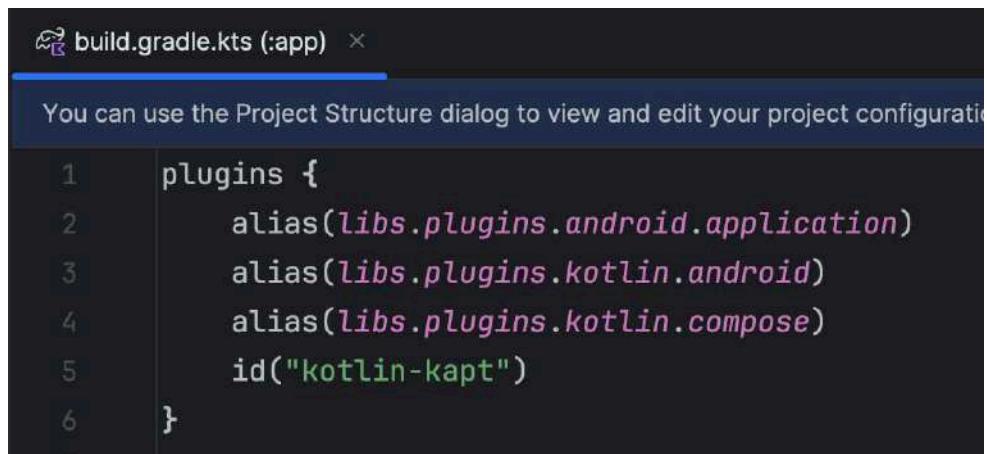
This is the app's **entry point** and UI host. It sets up the main content view using Jetpack Compose and connects the `PostViewModel` to the UI layer. It may contain composable functions that display the list of posts and handle user interactions (like pressing a button to generate a new post). It also observes the state provided by the ViewModel and reacts accordingly.

***This structure follows a clean separation of concerns:***

- **Model & Data Layer** → `data/`
- **Business Logic & State** → `PostViewModel`
- **UI Entry Point** → `MainActivity`
- **Visual Styling** → `ui.theme/`

# Gradle Pre-Setting for this week

**Step 1:** Go to build.gradle.kts file and add "id("kotlin-kapt")" in the *plugins{}*



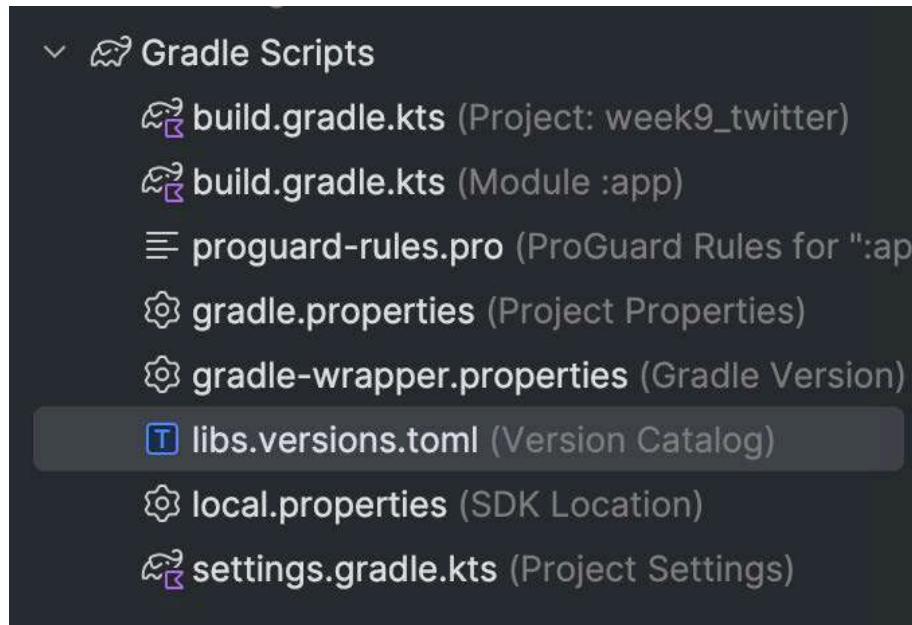
The screenshot shows the Android Studio code editor with the file 'build.gradle.kts' selected. The code is as follows:

```
1 plugins {
2     alias(libs.plugins.android.application)
3     alias(libs.plugins.kotlin.android)
4     alias(libs.plugins.kotlin.compose)
5     id("kotlin-kapt")
6 }
```

**Step 2:** Go to build.gradle.kts file and add the following in the *dependencies{}*

```
// Room dependencies
implementation(libs.androidx.room.runtime)
implementation(libs.androidx.room.ktx)
kapt(libs.androidx.room.compiler)
```

**Step 3:** Find the libs.versions.toml file:



```
1 [versions]
2     agp = "8.9.2"
3     kotlin = "2.0.21"
4     coreKtx = "1.16.0"
5     junit = "4.13.2"
6     junitVersion = "1.2.1"
7     espressoCore = "3.6.1"
8     lifecycleRuntimeKtx = "2.8.7"
9     activityCompose = "1.10.1"
10    composeBom = "2024.09.00"
11    room = "2.6.1"
```

Add: room = "2.6.1" in line 11

```
27 androidx-material3 = { group = "androidx.compose.material3", name = "material3" }
28 androidx-room-runtime = { group = "androidx.room", name = "room-runtime", version.ref = "room" }
29 androidx-room-ktx = { group = "androidx.room", name = "room-ktx", version.ref = "room" }
30 androidx-room-compiler = { group = "androidx.room", name = "room-compiler", version.ref = "room" }
```

Add the following after line 27 ↓

```
androidx-room-runtime = { group = "androidx.room", name = "room-runtime", version.ref = "room" }
androidx-room-ktx = { group = "androidx.room", name = "room-ktx", version.ref = "room" }
androidx-room-compiler = { group = "androidx.room", name = "room-compiler", version.ref = "room" }
```

## Step 1: Entity

The `Post` class is the foundation of the app's data layer. It represents each post or tweet in a structured format, capturing all the essential information needed for a social media message. Each post includes:

- a unique ID
- the user's name
- a subject
- the full content of the post
- and the time it was created

This class is annotated with `@Entity`

- which marks it as a Room entity and allows it to be stored in the local database. The `@PrimaryKey` annotation on `postId` defines the unique identifier for each record. The rest of the fields—such as `userName`, `subject`, `content`, and `createdAt`—map directly to columns in the database table.

As part of the MVVM architecture, this data model acts as the bridge between the persistence layer and the UI. It ensures a consistent, type-safe way to handle tweet data throughout the app. Whether the data is retrieved from an API or loaded from local storage, the `Post` class supports smooth data flow to the UI and makes it easier to display content or update views.

```
1 package com.fit2081.week9_twitter.data
2
3 import androidx.room.Entity
4 import androidx.room.PrimaryKey
5
6 /**
7  * Entity representing a post/tweet in the database.
8  * This class defines the structure of a post with all its attributes.
9  * It's annotated as a Room entity, which allows Room to create a database table
10 * for storing post data.
11 */
12 @Entity(tableName = "posts")
13 data class Post(
14     /**
15      * Unique identifier for this post.
16      * This serves as the primary key in the database table.
17      */
18     @PrimaryKey
19     val postId: String,
20     /**
21      * The name of the user who created this post.
22      * Typically includes first and last name.
23      */
24     val userName: String,
25     /**
26      * The subject or title of the post.
27      * Provides a brief description of what the post is about.
28      */
29     val subject: String,
30     /**
31      * The main content of the post.
32      * Contains the actual message that the user wants to share.
33      */
34     val content: String,
35     /**
36      * The date and time when this post was created.
37      * Stored as a string to maintain compatibility with different time formats.
38      */
39     val createdAt: String
40 )
```

## Step 2: Dao

The `PostDao` interface defines the methods for interacting with the local Room database. It is annotated with `@Dao`, which tells Room to treat this interface as a Data Access Object. This layer provides the app with type-safe methods to **query**, **insert**, and **delete** post data.

The `PostDao` provides a clean and simple interface to perform basic CRUD operations on the `posts` table. It helps keep the repository and ViewModel layers free from SQL logic and makes Room integration seamless. By using `Flow` and coroutines, it ensures that all data operations are efficient, lifecycle-aware, and non-blocking.

### `getAllPosts()`

```
@Query("SELECT * FROM posts ORDER BY createdAt DESC")
fun getAllPosts(): Flow<List<Post>>
```

- Retrieves all posts from the database, ordered by creation time (newest first).
- Returns a `Flow`, meaning changes to the database will automatically trigger UI updates.
- Ideal for reactive UI built with Jetpack Compose.

### `insertPost(...)`

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun insertPost(post: Post): Long
```

- Inserts a single post into the database.
- If the post ID already exists, it will be replaced.
- `suspend` keyword allows it to run inside a coroutine without blocking the main thread.

### `insertPosts(...)`

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun insertPosts(posts: List<Post>): List<Long>
```

- Inserts multiple posts in a single operation.
- Useful for bulk operations like populating the database with sample data.
- The `onConflict = OnConflictStrategy.REPLACE` Setting tells Room what to do when a conflict occurs during an insert operation—specifically, when a row with the same **primary key** already exists in the database.

In this case, **REPLACE** means that if the new data has the same primary key as an existing record, Room will **delete the old row and insert the new one** in its place. This ensures that the new data completely overwrites the previous version.

It's useful when:

- You want to avoid duplicates based on primary keys.
- You treat each insert as a way to update the most recent version of the data.

**Important:** REPLACE is destructive—it deletes the existing row before inserting the new one, which can affect foreign key relationships if any exist.

## deleteAllPosts()

```
@Query("DELETE FROM posts")  
suspend fun deleteAllPosts()
```

- Deletes **all** posts from the database.
- Commonly used for reset or “clear all” functionality.
- Also a `suspend` function to run on a background thread.

## PostDao.kt

```
1 package com.fit2081.week9_twitter.data
2
3 import androidx.room.Dao
4 import androidx.room.Insert
5 import androidx.room.OnConflictStrategy
6 import androidx.room.Query
7 import kotlinx.coroutines.flow.Flow
8 import java.util.Date
9 import java.util.Random
10 import java.util.UUID
11 import kotlin.random.Random as KotlinRandom
12
13 /**
14 * Data Access Object (DAO) for the Post entity.
15 * Provides methods to access and manipulate posts in the database.
16 */
17 @Dao
18 interface PostDao {
19     /**
20      * Get all posts from the database, ordered by creation time (newest first)
21      */
22     @Query("SELECT * FROM posts ORDER BY createdAt DESC")
23     fun getAllPosts(): Flow<List<Post>>
24
25     /**
26      * Insert a new post into the database
27      */
28     @Insert(onConflict = OnConflictStrategy.REPLACE)
29     suspend fun insertPost(post: Post): Long
30
31     /**
32      * Insert multiple posts into the database
33      */
34     @Insert(onConflict = OnConflictStrategy.REPLACE)
35     suspend fun insertPosts(posts: List<Post>): List<Long>
36
37     /**
38      * Delete all posts from the database
39      */
40     @Query("DELETE FROM posts")
41     suspend fun deleteAllPosts()
42 }
```

## Step 3: Database Class

The `AppDatabase` class is the central component for managing local data in this application. It is annotated with `@Database`, which tells Room to create and manage a table based on the `Post` entity. This sets up the structure for storing post data on the device.

Inside the class, the `postDao()` method gives access to the `PostDao` interface. This is how other parts of the app perform actions like inserting or retrieving posts from the local database.

To make sure the app only creates one database instance, this class uses the **singleton pattern**. That means only one connection to the database is shared throughout the app. This helps save memory and avoids bugs from multiple database instances.

The database instance is created through the `getDatabase()` function. This method checks if an instance already exists; if not, it builds one using `Room.databaseBuilder()` and assigns it to a shared variable called `INSTANCE`.

In summary, `AppDatabase` handles all database setup and access. It works with Room to simplify data operations, supports offline storage of posts, and plays an important role in connecting the repository and the UI in a clean, architecture-friendly way.

```
1 package com.fit2081.week9_twitter.data
2
3 import android.content.Context
4 import androidx.room.Database
5 import androidx.room.Room
6 import androidx.room.RoomDatabase
7
8 /**
9  * Main database class for the application.
10 *
11 * This abstract class provides an interface for accessing the application's database.
12 * It uses Room persistence library to handle database operations and defines
13 * the database configuration including entities and DAOs.
14 */
15 @Database(entities = [Post::class], version = 1, exportSchema = false)
16 abstract class AppDatabase : RoomDatabase() {
17
18     /**
19      * Provides access to the Post Data Access Object.
20      *
21      * @return PostDao instance for database operations related to posts.
22      */
23     abstract fun postDao(): PostDao
24 }
```

```
25     /**
26      * Companion object to manage database instance using the Singleton pattern.
27      * This ensures only one instance of the database is created throughout the app,
28      * which is an important consideration for resource management.
29      */
30     companion object {
31         /**
32          * Volatile annotation ensures the INSTANCE is always up-to-date
33          * and the same for all execution threads.
34          */
35         @Volatile
36         private var INSTANCE: AppDatabase? = null
37
38         /**
39          * Gets the singleton database instance, creating it if it doesn't exist.
40          *
41          * @param context The application context needed to create the database.
42          * @return The singleton AppDatabase instance.
43          */
44         fun getDatabase(context: Context): AppDatabase {
45             return INSTANCE ?: synchronized(this) {
46                 val instance = Room.databaseBuilder(
47                     context.applicationContext,
48                     AppDatabase::class.java,
49                     "tweets_database"
50                 )
51                 .build()
52                 // Assign the newly created instance to INSTANCE
53                 INSTANCE = instance
54                 instance // Return the instance
55             }
56         }
57     }
58 }
```

## Step 4: Repository: Managing Multi-source data

The `PostRepository` class is a key part of the app's architecture. It follows the Repository pattern, which helps separate data logic from the rest of the application. This class sits between the data sources and the ViewModel, acting as the middle layer that controls where and how data is retrieved or saved.

The repository handles both *local* and *remote* data sources. It fetches posts from the local Room database or from the remote API, depending on network availability. This is known as an **offline-first approach**—the app continues working even when the device is offline by using cached data stored locally.

In addition to switching between data sources, the repository also takes care of important tasks like *detecting network state*, *handling errors* (such as failed API calls), and transforming raw data into a format the UI can easily use. This means that the ViewModel doesn't need to worry about these details and can focus on business logic, while the UI only needs to focus on what to display.

Overall, `PostRepository` plays a critical role in ensuring smooth and efficient data flow throughout the app. It simplifies complex data operations, improves performance, and makes the app more reliable for users in real-world conditions.



// Remember to implement the following in the build.gradle.kts file.  
// Retrofit for network calls

```
implementation("com.squareup.retrofit2:retrofit:2.9.0")
implementation("com.squareup.retrofit2:converter-gson:2.9.0")
implementation("com.google.code.gson:gson:2.10.1")
```

```
package com.fit2081.week9_twitter.data

import android.content.Context
import android.net.ConnectivityManager
import android.net.NetworkCapabilities
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.withContext
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory
import retrofit2.http.GET
import retrofit2.http.POST
import java.time.LocalDateTime
import java.time.format.DateTimeFormatter
import java.util.UUID
```

```
class PostRepository(private val applicationContext: Context) {
    // Get database instance and create repository
    val database = AppDatabase.getDatabase(applicationContext)
    val postDao = database.postDao()
```

↑Get a database instance and create the repository.

Retrofit instance configured for API communication

- Uses the base URL for the tweets API service
- Configures Gson converter for JSON parsing

```
private val retrofit = Retrofit.Builder()
    .baseUrl("http://34.129.121.193:3000/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

The backend server holds the posts it generates for users and responds to clients' GET requests.  
Sample of the JSON object:

```
{
  "postId": "31cf7090-7cd6-42d8-931d-921e38080b3b",
  "userName": "Sarah Davis",
  "content": "Hiking in the mountains gives me so much peace.",
  "subject": "Education",
  "createdAt": "2025-04-30T00:13:32.944Z"
}
```

API service interface implementation created by Retrofit. This is used to make network requests to the remote API

```
private val apiService = retrofit.create(Tweet ApiService::class.java)
```

Retrofit service interface that defines API endpoints for tweet operations. Retrofit uses this interface to generate a concrete implementation for making HTTP requests to the remote API server.

```
interface Tweet ApiService {  
    /**  
     * GET endpoint to retrieve all tweets from the server.  
     *  
     * @return List<Post> containing all posts from the API  
     */  
    @GET("api/tweets")  
    suspend fun getTweets(): List<Post>  
  
    /**  
     * POST endpoint to create a new tweet on the server.  
     *  
     * @return Post the newly created post returned by the API  
     */  
    @POST("api/tweets/new")  
    suspend fun createTweet(): Post  
}
```

Now, let's develop a function to fetch all posts from either the network or the local database.

Implementation details:

- Checks for network availability first
- If online, attempts to fetch from the API and update the local database
- If the API call fails or is offline, it falls back to the local database
- In worst-case scenarios, it returns an empty list to avoid null values

```
suspend fun getAllPosts(): List<Post> {  
    return try {  
        if (isNetworkAvailable()) {  
            try {  
                // Fetch from API if online  
                val apiPosts = withContext(Dispatchers.IO) {  
                    apiService.getTweets()  
                }  
                // Convert API response to Post entities and insert into database  
                val posts = apiPosts.map { apiPost ->
```

```

        Post(
            postId = apiPost.postId ?: "",
            userName = apiPost.userName ?: "Unknown User",
            subject = apiPost.subject ?: "No Subject",
            content = apiPost.content ?: "",
            createdAt = apiPost.createdAt?: ""
        )
    }
    // Save API posts to local database for offline access
    if (posts.isNotEmpty()) {
        withContext(Dispatchers.IO) {
            posts.forEach { post ->
                postDao.insertPost(post)
            }
        }
    }
    posts
} catch (e: Exception) {
    // If API call fails, fall back to local data
    e.printStackTrace()
    withContext(Dispatchers.IO) {
        postDao.getAllPosts().first()
    }
}
} else {
    // If offline, return local data
    withContext(Dispatchers.IO) {
        postDao.getAllPosts().first()
    }
}
} catch (e: Exception) {
    // If everything fails, at least return an empty list
    e.printStackTrace()
    emptyList()
}
}
}

```

The function **isNetworkAvailable** checks if the device has an active internet connection.  
Implementation details:

- Obtains the ConnectivityManager system service
- Queries for active network capabilities
- Checks if Wi-Fi, cellular, or Ethernet transport is available

```

fun isNetworkAvailable(): Boolean {
    // Get the ConnectivityManager system service
    val connectivityManager = applicationContext.getSystemService(Context.CONNECTIVITY_SERVICE)
    // Check if the device has an active network
    val network = connectivityManager.activeNetwork ?: return false

```

```
// Get the network capabilities for the active network
val capabilities = connectivityManager.getNetworkCapabilities(network) ?: return false
// Check if the network has any of the following transports:
return capabilities.hasTransport(NetworkCapabilities.TRANSPORT_WIFI) ||
    capabilities.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR) ||
    capabilities.hasTransport(NetworkCapabilities.TRANSPORT_ETHERNET)
}
```

The function **createPost** creates a new post, either by calling the remote API or locally.  
Implementation details:

- If online, it calls the API endpoint to generate a new post
- If offline, it creates a random post locally
- Updates the local database in both scenarios
- After creating the post, refresh the posts list

```
suspend fun createPost(): Boolean {
    return try {
        if (isNetworkAvailable()) {
            // Make API call to create a post on the server
            val apiResponse = withContext(Dispatchers.IO) {
                apiService.createTweet()
            }
            // Refresh posts to include the newly created one
            getAllPosts()
            return true
        } else {
            // If offline, create a local post with random content
            val randomPost = createLocalPost()
            postDao.insertPost(randomPost)
            // Refresh posts to include the newly created one
            getAllPosts()
            return true
        }
    } catch (e: Exception) {
        e.printStackTrace()
        return false
    }
}
```

**Deletes all posts** from the local database.

Note: This operation only affects local data and doesn't delete

Posts from the remote API server.

```
suspend fun deleteAllPosts() {
    withContext(Dispatchers.IO) {
        postDao.deleteAllPosts()
    }
}
```

**Creates a random post** with generated content for offline use.

This helper method generates:

- A random username from predefined options
- A subject title
- Content based on content templates
- A formatted date-time string for the current time

```
private fun createLocalPost(): Post {
    // Define date formatters for consistent formatting
    val firstApiFormat = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")
    val formatter = DateTimeFormatter.ofPattern("HH:mm:ss dd-MM-yyyy")
    val now = LocalDateTime.now()
    // Format the current date-time
    val formatDate = now.format(formatter)

    // Sample data collections for random generation
    val userNames = listOf(
        "John Smith", "Emma Wilson", "Michael Brown",
        "Sophia Johnson", "Robert Davis", "Olivia Jones"
    )
    val subjects = listOf("Local Post")
    val contentTemplates = listOf(
        "Just had an amazing experience with %s!",
        "Does anyone else think %s is overrated?",
        "I can't believe what's happening in %s right now",
        "Looking for recommendations about %s",
        "My thoughts on %s: absolutely fascinating!",
        "Today I learned something new about %s"
    )

    // Generate the post content
    val selectedSubject = subjects.random()
    val content = contentTemplates.random().format(selectedSubject.lowercase())

    // Create and return the post with a unique ID
    return Post(
        postId = UUID.randomUUID().toString(),
        userName = userNames.random(),
        subject = selectedSubject,
```

```
    content = content,  
    createdAt = formatDateTime  
)  
}
```

## Step 5: ViewModel

This ViewModel acts as a **bridge between the UI and the Repository**. It holds UI-related data and exposes it as observable state, while delegating all data operations to the repository.

The PostViewModel class is a pivotal element in the application's MVVM architecture, acting as a seamless intermediary between the UI and data layers. It encapsulates all data-related operations, freeing the activity and composables to focus exclusively on presentation. By extending AndroidViewModel, it leverages the application context for repository interactions and persists through configuration changes. Utilising Kotlin coroutines and StateFlow, it delivers a reactive programming model, enabling the UI to reflect data updates effortlessly. The ViewModel manages threading complexity, ensuring that intensive tasks, such as network requests or database operations, run on a non-UI thread. This design fosters a maintainable, testable, and scalable codebase by distinctly separating concerns across application layers.

```
class PostViewModel(application: Application) : AndroidViewModel(application)
```

- Inherits from `AndroidViewModel`, giving access to `applicationContext`.
- This is needed because the repository relies on Room, which requires context.

```
private val repository = PostRepository(application)
```

- The repository handles all data operations, including Room and API access.
- ViewModel never directly touches the database or network.

```
private val _allPosts = MutableStateFlow<List<Post>>(emptyList())
val allPosts: StateFlow<List<Post>>
    get() = _allPosts.asStateFlow()
```

- `_allPosts` is the private mutable state.
- `allPosts` is the public, read-only version that the UI can observe.
- This makes state flow one-directional and safe.

```
fun generateRandomPost() {
    viewModelScope.launch(Dispatchers.IO) {
        repository.createPost()
    }
}
```

- Called when the “Add Tweet” button is pressed.
- Launches a coroutine to create a new random post.

```
fun refreshPosts() {
```

```
    viewModelScope.launch(Dispatchers.IO) {  
        _allPosts.value = repository.getAllPosts()  
    }  
}
```

- Retrieves all posts from the repository.

```
fun deleteAllPosts() {
    viewModelScope.launch(Dispatchers.IO) {
        repository.deleteAllPosts()
        refreshPosts()
    }
}
```

- Clears all posts from the database.
  - Useful when the user presses the “Delete” button.
  - Triggers `refreshPosts()` to update the UI immediately.

```
fun isNetworkAvailable(): Boolean {  
    return repository.isNetworkAvailable()  
}
```

- Allows the UI to determine whether the app is online.
  - Useful for enabling or disabling certain features (e.g., if (`isOnline`) "Online" else "Offline")

```
1 package com.fit2081.week9_twitter.data
2
3 import android.app.Application
4 import androidx.lifecycle.AndroidViewModel
5 import androidx.lifecycle.viewModelScope
6 import kotlinx.coroutines.Dispatchers
7 import kotlinx.coroutines.flow.MutableStateFlow
8 import kotlinx.coroutines.flow.StateFlow
9 import kotlinx.coroutines.flow.asStateFlow
10 import kotlinx.coroutines.launch
11
12 /**
13 * ViewModel for handling post-related UI operations and data management.
14 *
15 * This class extends AndroidViewModel to have access to the Application context
16 * which is required for the repository. It serves as the communication center
17 * between the UI (MainActivity) and the data layer (PostRepository).
18 *
19 * Key responsibilities:
20 * - Provide observable state of posts to the UI
21 * - Handle UI-triggered operations like post creation and deletion
22 * - Manage data loading and refreshing operations
23 * - Abstract away the complexity of threading and data fetching
24 *
25 * @property application The application instance provided by the Android framework
26 */
27 class PostViewModel(application: Application) : AndroidViewModel(application) {
28
29     /**
30      * Repository instance for handling all data operations
31      */
32     private val repository = PostRepository(application)
33
34     // Observable state for posts
35     private val _posts = MutableStateFlow<List<Post>?>(null)
36     val posts: StateFlow<List<Post>?> get() = _posts
37
38     // UI thread launch scope
39     private val uiScope = viewModelScope
40
41     // Initialize repository
42     init {
43         uiScope.launch {
44             repository.fetchPosts()
45         }
46     }
47
48     // Create a new post
49     fun createPost(post: Post) {
50         uiScope.launch {
51             repository.createPost(post)
52         }
53     }
54
55     // Delete a post
56     fun deletePost(postId: String) {
57         uiScope.launch {
58             repository.deletePost(postId)
59         }
60     }
61
62     // Refresh posts
63     fun refreshPosts() {
64         uiScope.launch {
65             repository.refreshPosts()
66         }
67     }
68 }
```

```
31     * This is the single point of contact for the ViewModel to interact with data sources.
32     */
33     private val repository: PostRepository = PostRepository(application.applicationContext)
34
35     /**
36      * Private mutable state flow that stores the current list of posts.
37      * Using StateFlow provides a way to observe changes to the data over time.
38      */
39     private val _allPosts = MutableStateFlow<List<Post>>(emptyList())
40
41     /**
42      * Public immutable StateFlow that exposes the current list of posts to observers.
43      * This property enables the UI to react to changes in the post data while
44      * preventing direct mutation from outside this class.
45      */
46     val allPosts: StateFlow<List<Post>>
47         get() = _allPosts.asStateFlow()
48
49     /**
50      * Initialize the ViewModel by loading posts from the repository.
51      * This ensures data is available as soon as the UI starts observing.
52      */
53     init {
54         // Trigger initial data fetch on ViewModel creation
55         refreshPosts()
56     }
57
58     /**
59      * Generates and inserts a new random post into the database.
60      *
61      * This method is typically called in response to user actions like
62      * pressing the "Add Tweet" button. It delegates the actual work
63      * of creating the post to the repository and runs in a background thread.
64      */
65     fun generateRandomPost() {
66         viewModelScope.launch(Dispatchers.IO) {
67             repository.createPost()
68         }
69     }
70
71     /**
72      * Refreshes the posts list by fetching the latest data from the repository.
73      *
74      * This method is responsible for updating the observed StateFlow with
75      * the most current data. It can be triggered manually (e.g., by pulling to refresh)
76      * or automatically when data changes might have occurred.
77      */
78     fun refreshPosts() {
79         viewModelScope.launch {
80             _allPosts.value = repository.getAllPosts()
81             println("Posts refreshed: ${_allPosts.value.size} posts loaded")
82         }
83     }
84
85     /**
86      * Checks if the device currently has an active internet connection.
87      *
88      * This method delegates to the repository's network checking functionality
89      * and is used by the UI to display the online/offline status indicator.
90      *
91      * @return true if the device is connected to the internet, false otherwise
92      */
93     fun isNetworkAvailable(): Boolean {
94         return repository.isNetworkAvailable()
```

```
95 }
96 /**
97  * Deletes all posts from the local database.
98  *
99  * This is typically called in response to the user clicking the delete/bin
100 * icon and confirming the deletion action. After deleting all posts,
101 * it refreshes the UI to reflect the empty state.
102 */
103 fun deleteAllPosts() {
104     viewModelScope.launch {
105         repository.deleteAllPosts()
106         refreshPosts() // Refresh the UI after deletion
107     }
108 }
109 }
110 }
111
112 }
```

codesnap.dev

- Inherits from `AndroidViewModel`, giving access to `applicationContext`.
- This is needed because the repository relies on Room, which requires context.

## Step 6: MainActivity.kt – UI Entry Point

The `MainActivity` class serves as the main user interface entry point for the app. It uses **Jetpack Compose** to build a dynamic, fully declarative UI. The layout is cleanly divided into modular composable functions, with each part responsible for a specific UI task. Below is an overview of each functional component:

### MainActivity and setContent

- The activity sets up the main content view using `setContent { }`.
- It retrieves a reference to the `PostViewModel`, which holds the app's state.
- It also checks network availability and passes it into the UI.
- The root composable is `MainScreen( ... )`, which serves as the main layout container.

### MainScreen( ... )

- This composable builds the **overall scaffold** of the screen using `Scaffold()` with a `TopAppBar` and a floating action button (FAB).
- The FAB allows users to generate a new random post by calling `generateRandomPost()` from the ViewModel.
- The TopAppBar includes a delete icon button that prompts a confirmation dialog before deleting all posts.
- It displays the current **network status** (online or offline) to the user.

### ListTweets( ... )

- This composable renders the **scrollable list of posts**, using `LazyColumn`.
- It observes the list of posts from the ViewModel's `StateFlow` and displays each post using the `PostCard( ... )` composable.
- The list updates automatically whenever the ViewModel's state changes.
- It handles the empty-state scenario (e.g., when there are no tweets).

### PostCard( ... )

- This function defines the UI layout for an individual tweet/post.
- Each card shows the user's name, subject, content, and timestamp.
- The card layout follows Material Design 3 standards for padding, typography, and elevation.
- It keeps the design simple and clean, focusing on readability.

## ConfirmDeleteDialog( . . . )

- A reusable composable that shows a confirmation dialog when the user wants to delete all posts.
- The dialog includes "Confirm" and "Cancel" buttons.
- It calls the `onConfirm()` callback only when the user explicitly confirms deletion.

## Summary

- The UI is reactive: it updates automatically based on ViewModel state.
- It uses clean modular composables for maintainability.
- It supports both **offline and online modes** with visual feedback.
- The use of Material Design 3 ensures consistency with modern Android design standards.
- All business logic is delegated to the ViewModel—UI just observes and responds.

```

1 package com.fit2081.week9_twitter
2
3 import android.os.Bundle
4 import androidx.activity.ComponentActivity
5 import androidx.activity.setContent
6 import androidx.activity.enableEdgeToEdge
7 import androidx.activity.viewModels
8 import androidx.compose.foundation.layout.*
9 import androidx.compose.foundation.lazy.LazyColumn
10 import androidx.compose.foundation.lazy.items
11 import androidx.compose.material.icons(Icons
12 import androidx.compose.material.icons.filled.Add
13 import androidx.compose.material.icons.filled.Delete
14 import androidx.compose.material.icons.filled.Refresh
15 import androidx.compose.material3.*
16 import androidx.compose.runtime.*
17 import androidx.compose.ui.Alignment
18 import androidx.compose.ui.Modifier
19 import androidx.compose.ui.text.font.FontWeight
20 import androidx.compose.ui.text.style.TextOverflow
21 import androidx.compose.ui.unit.dp
22 import com.fit2081.week9_twitter.data.Post
23 import com.fit2081.week9_twitter.data.PostViewModel
24 import com.fit2081.week9_twitter.ui.theme.Week9_twitterTheme
25 import kotlinx.coroutines.delay
26
27 /**
28 * Main activity for the Twitter Clone application.
29 * This class serves as the entry point of the application and hosts the main UI components
30 * built with Jetpack Compose.
31 *
32 * The app follows MVVM architecture pattern:

```

```
33 * - View: MainActivity and its composables (MainScreen, ListTweets, PostCard)
34 * - ViewModel: PostViewModel that handles business logic and data operations
35 * - Model: Post entity and repository (handled through PostViewModel)
36 */
37 class MainActivity : ComponentActivity() {
38     /**
39      * ViewModel instance that handles data operations and UI state
40      * Using by viewModels() delegates the lifecycle management to the activity
41      */
42     private val postViewModel: PostViewModel by viewModels()
43
44     @OptIn(ExperimentalMaterial3Api::class)
45     override fun onCreate(savedInstanceState: Bundle?) {
46         super.onCreate(savedInstanceState)
47         // Enable edge-to-edge display (immersive mode)
48         enableEdgeToEdge()
49         // Set up the Compose UI
50         setContent {
51             // Apply the application theme
52             Week9_twitterTheme {
53                 // Display the main screen with the ViewModel instance
54                 MainScreen(postViewModel)
55             }
56         }
57     }
58 }
59
60 /**
61 * Main screen composable that sets up the app scaffold with top bar and content area.
62 *
63 * Features:
64 * - Top app bar with network status indicator
65 * - Refresh button to manually update posts
66 * - Delete button to remove all posts (with confirmation)
67 * - Add button to create new random posts
68 * - Content area displaying the list of tweets
69 *
70 * @param postViewModel The ViewModel that provides data and handles user actions
71 */
72 @OptIn(ExperimentalMaterial3Api::class)
73 @Composable
74 fun MainScreen(postViewModel: PostViewModel) {
75     // State to track network connection status
76     var isOnline by remember { mutableStateOf(false) }
77     // State to control delete confirmation dialog
78     var showDeleteConfirmationDialog by remember { mutableStateOf(false) }
79
80     // Check network status periodically (every 5 seconds)
81     // this could be improved with a more efficient approach such as using
82     // a BroadcastReceiver that listens for network changes
83     LaunchedEffect(Unit) {
84         while (true) {
85             isOnline = postViewModel.isNetworkAvailable()
86             delay(5000)
87         }
88     }
89
90     // Main scaffold that provides the app structure
91     Scaffold(
92         topBar = {
93             TopAppBar(
94                 title = {
95                     Row(verticalAlignment = Alignment.CenterVertically) {
96                         Text("Twitter Clone")
97                         Spacer(modifier = Modifier.width(8.dp))
98                         // Network status badge (green for online, red for offline)
99                         Surface(
100                             color = if (isOnline) MaterialTheme.colorScheme.primary
101                             else MaterialTheme.colorScheme.error,
102                             shape = MaterialTheme.shapes.small
103                         )
104                     }
105                 }
106             )
107         }
108         content = {
109             ListTweets(postViewModel)
110         }
111     )
112 }
```

```
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
    ) {
        Text(
            text = if (isOnline) "Online" else "Offline",
            modifier = Modifier.padding(horizontal = 8.dp, vertical = 4.dp),
            style = MaterialTheme.typography.bodySmall,
            color = MaterialTheme.colorScheme.onPrimary
        )
    }
},
actions = {
    // Delete all posts button (triggers confirmation dialog)
    IconButton(onClick = { showDeleteConfirmationDialog = true }) {
        Icon(
            imageVector = Icons.Default.Delete,
            contentDescription = "Delete All Posts"
        )
    }
    // Manually refresh posts from source
    IconButton(onClick = { postViewModel.refreshPosts() }) {
        Icon(
            imageVector = Icons.Default.Refresh,
            contentDescription = "Refresh Posts"
        )
    }
    // Create a new random post
    Button(
        onClick = { postViewModel.generateRandomPost() },
        modifier = Modifier.padding(end = 16.dp)
    ) {
        Icon(
            imageVector = Icons.Filled.Add,
            contentDescription = "Add",
        )
    }
}
),
modifier = Modifier.fillMaxSize()
) { innerPadding →
    // Content area displaying the list of tweets
    ListTweets(
        viewModel = postViewModel,
        modifier = Modifier.padding(innerPadding)
    )
}

// Confirmation dialog for deleting all posts (shown conditionally)
if (showDeleteConfirmationDialog) {
    AlertDialog(
        onDismissRequest = { showDeleteConfirmationDialog = false },
        title = { Text("Delete All Tweets") },
        text = { Text("Are you sure you want to delete all tweets? This action cannot be undone.") },
        confirmButton = {
            // Delete button (red to indicate destructive action)
            Button(
                onClick = {
                    postViewModel.deleteAllPosts()
                    showDeleteConfirmationDialog = false
                },
                colors = ButtonDefaults.buttonColors(
                    containerColor = MaterialTheme.colorScheme.error
                )
            ) {
                Text("Delete")
            }
        },
        dismissButton = {
            // Cancel button
        }
    )
}
```

```
172             OutlinedButton(
173                 onClick = { showDeleteConfirmationDialog = false }
174             ) {
175                 Text("Cancel")
176             }
177         )
178     )
179 }
180 }
181
182 /**
183 * Composable that displays the list of tweets from the ViewModel.
184 *
185 * Features:
186 * - Shows a list of tweets in a scrollable column when available
187 * - Displays an empty state message when no tweets exist
188 * - Each tweet is represented as a card
189 *
190 * @param viewModel The ViewModel that provides the tweet data
191 * @param modifier Modifier for customizing the layout
192 */
193 @Composable
194 fun ListTweets(viewModel: PostViewModel, modifier: Modifier = Modifier) {
195     // Trigger posts refresh when component is composed
196     viewModel.refreshPosts()
197
198     // Collect posts from viewModel.allPosts flow using collectAsState
199     val posts by viewModel.allPosts.collectAsState(initial = emptyList())
200
201     if (posts.isEmpty()) {
202         // Empty state: Display a message when no posts are available
203         Box(
204             modifier = modifier.fillMaxSize(),
205             contentAlignment = Alignment.Center
206         ) {
207             Column(
208                 horizontalAlignment = Alignment.CenterHorizontally,
209                 verticalArrangement = Arrangement.Center
210             ) {
211                 Text(
212                     text = "No tweets available",
213                     style = MaterialTheme.typography.bodyLarge
214                 )
215                 Spacer(modifier = Modifier.height(16.dp))
216                 Text(
217                     text = "Click 'Add Tweet' to create one",
218                     style = MaterialTheme.typography.bodyMedium
219                 )
220             }
221         }
222     } else {
223         // List state: Display all posts in a scrollable LazyColumn
224         LazyColumn(
225             modifier = modifier.fillMaxSize(),
226             contentPadding = PaddingValues(16.dp),
227             verticalArrangement = Arrangement.spacedBy(12.dp)
228         ) {
229             items(posts) { post →
230                 PostCard(post)
231             }
232         }
233     }
234 }
235
236 /**
237 * Composable card that displays a single tweet/post.
238 *
239 * Features:
240 * - Shows the username and creation time
241 * - Displays the subject in a highlighted style
```

```
242 * - Shows the content with ellipsis for long text
243 * - Applies consistent styling using Material Design
244 *
245 * @param post The Post entity containing the data to display
246 */
247 @Composable
248 fun PostCard(post: Post) {
249     Card(
250         modifier = Modifier.fillMaxWidth(),
251         elevation = CardDefaults.cardElevation(defaultElevation = 4.dp)
252     ) {
253         Column(
254             modifier = Modifier
255                 .fillMaxWidth()
256                 .padding(16.dp)
257         ) {
258             // Header row: Username and timestamp
259             Row(
260                 modifier = Modifier.fillMaxWidth(),
261                 horizontalArrangement = Arrangement.SpaceBetween,
262                 verticalAlignment = Alignment.CenterVertically
263             ) {
264                 Text(
265                     text = post.userName,
266                     style = MaterialTheme.typography.titleMedium,
267                     fontWeight = FontWeight.Bold
268                 )
269                 Text(
270                     text = post.createdAt,
271                     style = MaterialTheme.typography.bodySmall
272                 )
273             }
274             Spacer(modifier = Modifier.height(8.dp))
275
276             // Subject line (highlighted with primary color)
277             Text(
278                 text = post.subject,
279                 style = MaterialTheme.typography.titleSmall,
280                 color = MaterialTheme.colorScheme.primary,
281                 fontWeight = FontWeight.Medium
282             )
283             Spacer(modifier = Modifier.height(4.dp))
284
285             // Main content with ellipsis for long text
286             Text(
287                 text = post.content,
288                 style = MaterialTheme.typography.bodyMedium,
289                 maxLines = 4,
290                 overflow = TextOverflow.Ellipsis
291             )
292         }
293     }
294 }
```

# FIT2081 Week 10

## Advanced Compose UI & Theming

# Week 10 - Lecture Reading Materials

---

## Introduction

Welcome to our lecture on Mobile User Interface Design in Android development. Last week, we explored the MVVM app architecture. Today, we'll move back to the design of your app and learn some advanced concepts of designing your mobile apps.

Consider our previous fitness app example: assuming you have already implemented all the logic (i.e., business logic and UI logic) and persistent data storage. However, you feel that there might be a missing design guideline for how your user interface (e.g., each screen) should look. And you may wonder whether your designed interface is professional enough or not. That is why we need to stick with the suggested design guidelines or principles, as well as learning some advanced techniques (e.g., animations and transitions) to make your app more visually appealing and engaging.

**Animations** add polish and help the app feel smooth and intentional rather than clunky or abrupt. Think of an app without animations like a silent movie; it might still work, but it feels flat and lifeless. While **transitions** visually guide users from one screen or state to another, reducing cognitive load and making the app feel responsive. They are like visual signposts, which tell users “where they came from” and “where they’re going.

## Learning Objectives

By the end of this week, you will be able to:

- **Identify different mobile design styles and paradigms**, such as Skeuomorphism, Flat Design, and Material Design.
- **Understand the principles of effective mobile UI design**, including layout, hierarchy, consistency, and accessibility.
- **Apply Material Design guidelines** to create clean and intuitive user interfaces using Jetpack Compose.
- **Identify different types of animations in Compose**, such as:
  - Simple state animations (`animate*AsState`)
  - Visibility transitions (`AnimatedVisibility`)
  - Content transitions (`Crossfade`)
- **Implement basic and advanced animations** using Jetpack Compose’s `updateTransition`, `rememberInfiniteTransition`, and key animation APIs.
- **Differentiate between transitions and animations**, and explain when each should be used.

# Mobile Design Styles and Paradigms

In the world of mobile app design, several design paradigms have emerged over the years, each serving different user experiences, visual aesthetics, and interaction patterns. Understanding these paradigms helps developers and designers create intuitive and effective mobile applications. In this section, we will explore three significant design styles: **Skeuomorphism**, **Flat Design**, and **Material Design**.

## Skeuomorphism

Skeuomorphism is a design concept that incorporates elements that mimic their real-world counterparts in a digital environment. The idea is to make digital objects look like physical objects in order to make the interface more familiar and intuitive to users.

### **Characteristics:**

- **Realistic Textures and Effects:** Often uses shadows, gradients, and textures to create the illusion of depth and a real-world appearance.
- **3D Effects:** Buttons, icons, and other UI elements may have bevels or embossing to look three-dimensional.
- **Mimicking Physical Objects:** App icons, such as a calendar that looks like a real paper calendar, or a music app that imitates a record player.

### **Example:**

- Early iOS designs (iPhone OS 1.x to 6.x) used skeuomorphic elements, like the "Notes" app looking like a real yellow legal pad, or the "Calendar" app that resembled an actual paper calendar.



Image credit: Robert Irish. Posted in [AppleInsider](#)

### Pros:

- Provides a familiar experience for users by simulating physical objects.
- Helps in making the app more intuitive, especially for new users unfamiliar with digital interactions.

### Cons:

- Can be visually overwhelming with excessive realism, leading to a cluttered UI.
- As devices became more powerful, the need for heavy skeuomorphic design faded, as users became more accustomed to digital interfaces.

## Flat Design

Flat Design is a minimalist design approach that eliminates any three-dimensional effects, such as gradients, shadows, and textures, and focuses on simplicity and clarity. The goal is to create a clean and easy-to-use interface with minimal distraction.

### Characteristics:

- **Simplified UI:** No 3D effects, gradients, or shadows. Icons and elements are flat with solid colors.
- **Minimalist Aesthetics:** Clean lines, simple shapes, and a focus on typography and color contrast.
- **2D Icons and UI Elements:** Icons are simplified, often using only outlines or solid colors, to communicate functionality without visual noise.

## **Example:**

- Microsoft's Metro design (now called Modern UI), used in Windows Phone and later in Windows 8, is an example of flat design with its use of simple typography and clean, grid-based layouts.



*Image Credit: iLink Digital @ [iLink Digital](#)*

- iOS 7, which moved away from skeuomorphism, adopted flat design principles, resulting in a much cleaner, more minimalistic interface.



*Image Credit: Rose Etherington @ [dezeen](#)*

### **Pros:**

- Provides a modern, clean, and simple user experience.
- Increases load times as there is less visual complexity (fewer textures and graphics).
- Easier to scale for different screen sizes and resolutions.

### **Cons:**

- Can feel too simplistic or bland if not executed thoughtfully.
- Sometimes lacks depth or visual hierarchy, which can confuse users about the importance of different elements.

## **Material Design**

Material Design, introduced by Google in 2014, is a design language that combines elements of flat design with subtle depth effects. It focuses on tactile surfaces, realistic lighting, and smooth animations to create an intuitive and responsive interface. Material Design emphasizes the use of "material" in its metaphor—real-world surfaces and the way light interacts with those surfaces.

### **Characteristics:**

- **Layered Design:** Elements are layered with shadows to create depth and indicate the

interaction state. For example, a raised button will have a shadow to indicate its clickable nature.

- **Movement and Animation:** Material Design heavily incorporates animations to show transitions between elements, adding dynamism and enhancing user experience.
- **Responsive Touch:** It focuses on touch feedback with visual responses to user interactions, such as ripples and shadows.
- **Bold and Graphic Design:** Bright colors, large typography, and grid-based layouts are common features of Material Design.

***Example:***

- Google's Android operating system from Lollipop (Android 5.0) onward uses Material Design extensively, with clear guidelines for shadow, motion, and interface components.

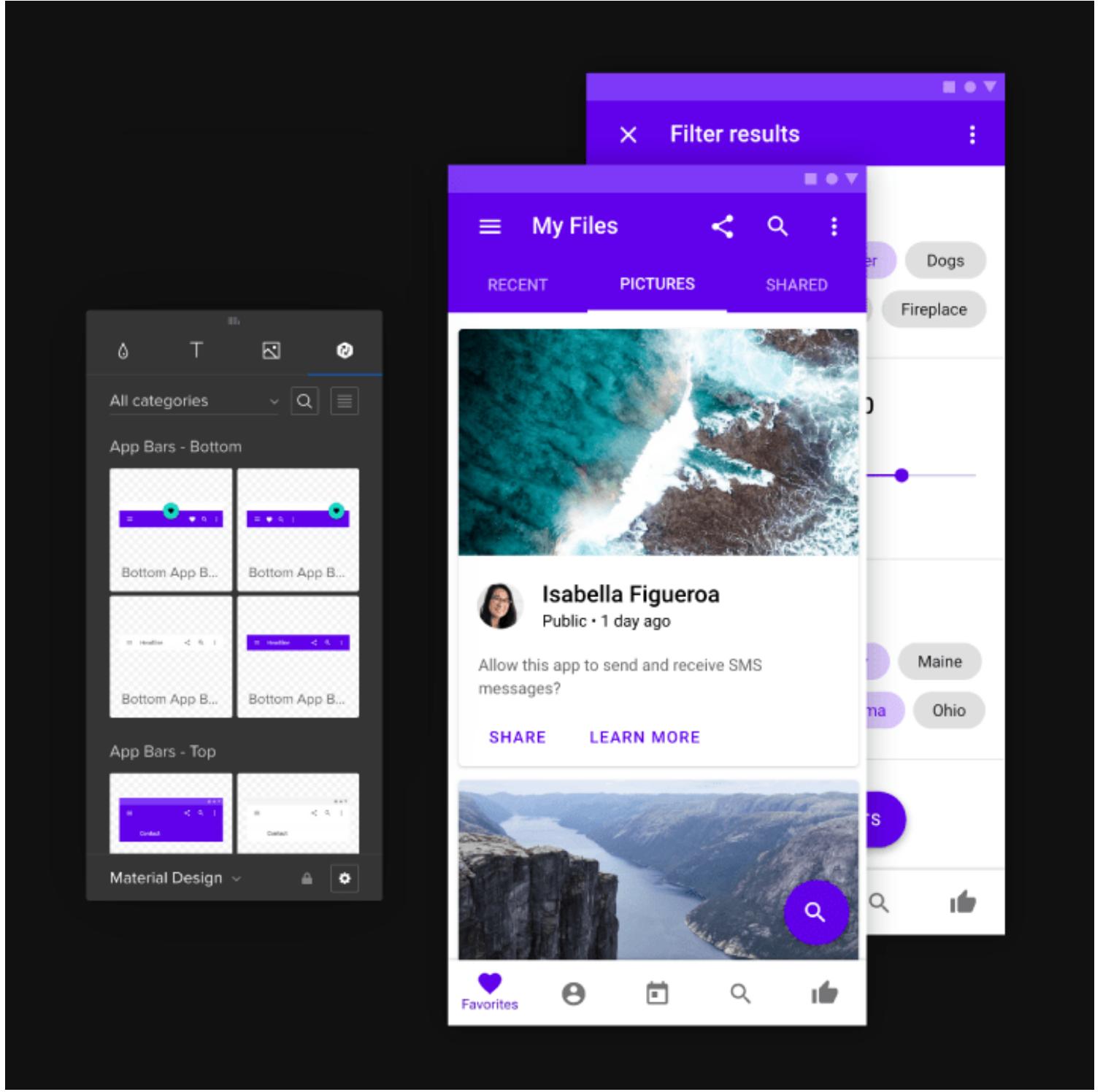


Image Credit: [uxpin.com](http://uxpin.com)

### Pros:

- The use of depth and motion helps users understand the app's behavior and flow.
- Provides a tactile, engaging experience while maintaining simplicity.
- Consistent design guidelines make it easy to implement and create unified user interfaces across Android apps.

### Cons:

- The need for animations and shadows can make the interface feel heavy on certain devices.
- Can be challenging to maintain consistency across different devices if not executed properly.

# Conclusion

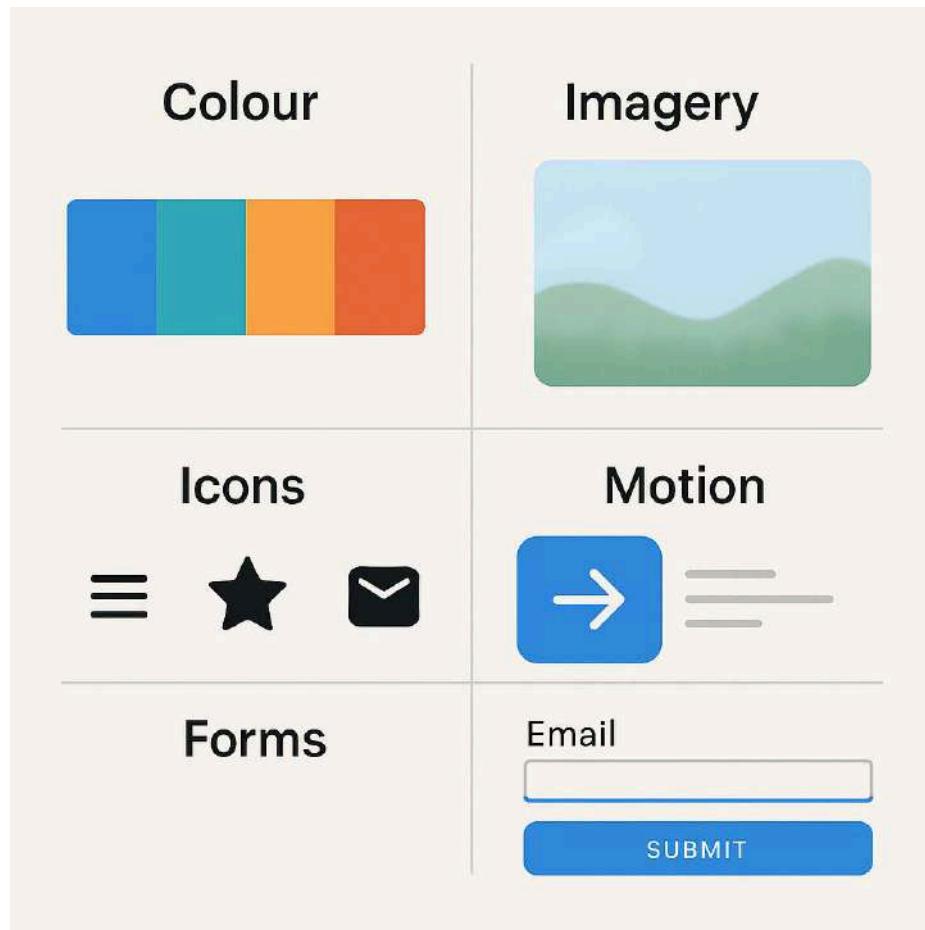
Each of these design styles has its place in the evolution of mobile UI/UX design:

- **Skeuomorphism** works best for apps that aim to closely mimic physical objects or are targeted at users less familiar with digital interfaces.
- **Flat Design** appeals to a minimalist aesthetic and is ideal for creating clean, modern, and fast interfaces.
- **Material Design** provides a balance between flat design and realism, offering rich interactivity with subtle, tactile elements that guide the user experience.

As mobile developers, understanding these paradigms and their evolution is key to choosing the right design principles for your app and providing users with an engaging, intuitive experience.

# Material Design for Android

Material Design, introduced by Google in 2014, is the design language for Android that provides a comprehensive set of guidelines for building user interfaces. The core principle of Material Design is to create a user-friendly, tactile experience that mimics the feel of physical materials, with a focus on realism, depth, and dynamic motion. In this section, we'll explore five key elements of Material Design that define its visual identity: **Colour, Imagery, Icons, Motion, and Forms**.



## Colour

Colour is a fundamental element in Material Design, playing a significant role in defining the look and feel of an app. The guidelines emphasize the use of vibrant, contrasting colours that help users navigate and interact with the app easily. Material Design also introduces a palette of primary and secondary colours that should be used consistently across the UI.

### **Key Points:**

- **Primary and Secondary Colours:** Every Material Design app should have a primary colour that reflects its brand identity, as well as secondary colours to support key interactions.
- **Accent Colours:** These are used to draw attention to specific UI elements, such as buttons,

sliders, or icons.

- **Colour Contrast:** Ensuring that there is enough contrast between text and background colours to maintain readability, especially for accessibility.
- **Shades and Tints:** Material Design encourages the use of various shades and tints of a base colour to provide visual depth and hierarchy.

### **Example:**

- A primary colour (e.g., blue) could be used for top navigation, while secondary colours (e.g., green or red) might highlight buttons or actions.

## **Imagery**

Imagery in Material Design includes the use of photographs, illustrations, and other visual assets. The guidelines stress that imagery should complement the design by enhancing storytelling, while maintaining clarity and avoiding distraction.

### **Key Points:**

- **Content-Focused Imagery:** Imagery should be used in a way that enhances the user experience without overwhelming the UI. For example, background images should be subtle and should not compete with the text or UI elements.
- **Authentic and Purposeful:** Images should feel natural and contribute to the content. They should reflect the app's purpose or narrative.
- **Adjusting for Screen Sizes:** Since Android devices come in various sizes and resolutions, imagery should be flexible and adaptable, using scalable vector graphics (SVG) and high-resolution images when appropriate.

### **Example:**

- Using a soft-focus image as a background that doesn't overpower the text on top, like the background in Google's News or Weather apps.

## **Icons**

Icons are a crucial part of Material Design, used to represent actions, objects, and navigation elements. They need to be clear, simple, and easily recognizable to provide a seamless interaction experience.

### **Key Points:**

- **Consistency:** Icons should follow a consistent style, ensuring that they are recognizable and match the app's overall aesthetic. Material Design specifies the use of simple geometric shapes

for icons.

- **Size and Spacing:** Icons should be the right size for touch targets and properly spaced to ensure they are tappable without cluttering the UI.
- **Minimalism:** Icons should use minimal details to convey their meaning, avoiding unnecessary embellishments.

#### **Example:**

- Common icons like the hamburger menu (three horizontal lines) or the floating action button (FAB), which is often a circular icon that indicates a primary action.

## **Motion**

Motion plays a key role in Material Design by providing visual feedback, guiding the user, and enhancing the sense of interactivity. Motion should be smooth and intuitive, helping users understand the transitions between different app states.

#### **Key Points:**

- **Meaningful Transitions:** Every motion in Material Design should have a clear purpose, whether it's to draw attention to a new element or to show the relationship between two UI components.
- **Easing and Timing:** Motion should use natural easing curves and appropriate timing to create a comfortable experience. Fast or jarring animations can disrupt the user's flow.
- **Responsive Motion:** Animation can be used to give immediate feedback, such as a button's ripple effect when pressed, to show the action has been registered.

#### **Example:**

- The transition between screens, where the UI elements move smoothly from one screen to another. This is seen in the transition of cards or modal dialogs in Android apps.

## **Forms**

Forms are essential UI elements in mobile apps, used for gathering user input. Material Design provides guidelines on how to create user-friendly forms that are easy to navigate, visually appealing, and responsive.

#### **Key Points:**

- **Input Fields:** Form fields should be clear, with proper labels, and should use floating labels to ensure that the label doesn't interfere with the input.
- **Error States:** Error states should be visually distinct to let the user know there's an issue with

the entered information, such as red underlines or error text.

- **Progress Indicators:** For longer forms or processes, progress indicators should show the user's current progress to keep them informed.

***Example:***

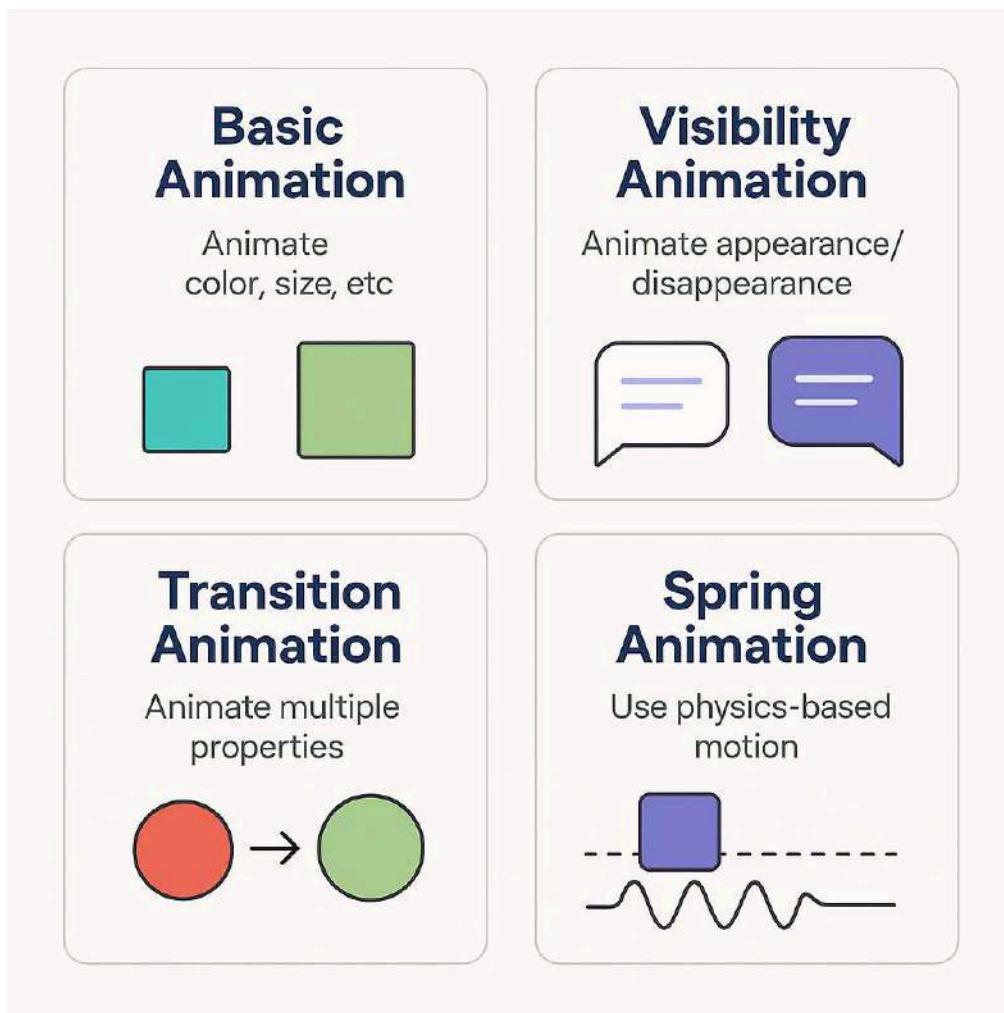
- Forms like the login screen, where fields are labeled clearly, and error states are indicated through animations or colour changes.

## Conclusion

Material Design's elements—colour, imagery, icons, motion, and forms—work together to create a cohesive, functional, and visually appealing user experience on Android. By adhering to these principles, developers can ensure that their apps are intuitive, responsive, and accessible. Whether you are designing a new app or refining an existing one, understanding these guidelines will help you build better, more engaging Android applications.

# Animations in Compose

Animations are an essential part of creating engaging user interfaces. In [Jetpack Compose](#), animations help make the user experience more dynamic and interactive. Whether it's a simple transition between UI elements or complex effects, Compose provides powerful and flexible tools for implementing animations.



## Types of Animations in Jetpack Compose

Jetpack Compose offers a variety of animation APIs, including:

- **Tween Animation:** This is the most commonly used animation type in Compose, where the animation's start and end states are defined, and the framework handles the interpolation between these states.
- **Keyframe Animation:** Keyframe animations allow you to define intermediate steps between two points of an animation. You can use this for more complex animations where you want the animation to follow a specific path.
- **Spring Animation:** This animation type mimics natural movements with oscillations and damping, similar to how objects behave in the real world.
- **Infinite Animation:** Infinite animations allow you to create loops or repeatable animations.

## Basic Animation with `animate*AsState`

The `animate*AsState` functions in Compose are the simplest way to animate properties in a Composable. You can animate almost any property such as color, size, position, or alpha, among others.

### Example: Animating a Box's Size

```
@Composable
fun AnimatedBox(modifier: Modifier = Modifier) {
    var expanded by remember { mutableStateOf(false) }

    val size by animateDpAsState(
        targetValue = if (expanded) 200.dp else 100.dp,
        animationSpec = tween(durationMillis = 500)
    )
    Column(
        modifier = modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center){
        Box(
            modifier = Modifier
                .size(size)
                .background(Color.Blue)
                .clickable { expanded = !expanded }
        )
    }
}
```

- `animateDpAsState` is used to animate the `size` property.
- When the `expanded` state changes (when the user clicks on the Box), the size transitions smoothly between 100.dp and 200.dp.

## Using `AnimatedVisibility` for Visibility Transitions

`AnimatedVisibility` allows you to animate the appearance or disappearance of a Composable based on visibility changes. It's ideal for managing UI elements that appear and disappear dynamically.

### Example: `AnimatedVisibility` for Showing and Hiding a Text

```
@Composable
fun ToggleTextVisibility() {
    var visible by remember { mutableStateOf(true) }

    Column {
```

```

        AnimatedVisibility(visible = visible) {
            Text("This text will fade in and out.")
        }

        Button(onClick = { visible = !visible }) {
            Text("Toggle Visibility")
        }
    }
}

```

- When the button is clicked, the `visible` state is toggled.
- `AnimatedVisibility` animates the appearance or disappearance of the `Text` Composable.

## Transition Animation with `updateTransition`

If you need more complex animations based on multiple states or conditions, you can use `updateTransition`. This provides greater control over the animation sequence and allows you to animate different properties of multiple Composables simultaneously.

### *Example: Animate Between Two States Using `updateTransition`*

```

@Composable
fun AnimatedStateTransition() {
    var selected by remember { mutableStateOf(false) }

    val transition = updateTransition(targetState = selected, label = "")

    val color by transition.animateColor(
        label = "",
        transitionSpec = { tween(durationMillis = 1000) }
    ) { state ->
        if (state) Color.Green else Color.Red
    }

    val size by transition.animateDp(
        label = "",
        transitionSpec = { tween(durationMillis = 1000) }
    ) { state ->
        if (state) 200.dp else 100.dp
    }

    Box(
        modifier = Modifier
            .size(size)
            .background(color)
            .clickable { selected = !selected }
    )
}

```

- `updateTransition` is used to manage the transition between two states (`selected` being `true` or `false`).
- Both `color` and `size` animate when the state changes.

## Spring Animations for Natural Movement

For more realistic animations, such as when a UI element behaves like it's bouncing or moving due to natural forces, use spring-based animations.

### ***Example: Bouncing Box with Spring Animation***

```
@Composable
fun SpringAnimationExample() {
    var tapped by remember { mutableStateOf(false) }

    val offset by animateFloatAsState(
        targetValue = if (tapped) 300f else 0f,
        animationSpec = spring(dampingRatio = Spring.DampingRatioMediumBouncy)
    )

    Box(
        modifier = Modifier
            .size(100.dp)
            .offset(x = offset.dp)
            .background(Color.Magenta)
            .clickable { tapped = !tapped }
    )
}
```

- The `spring` animation spec provides a bouncing effect.
- The box moves horizontally when clicked, and the transition is animated with a spring effect.

## Infinite Animations

Infinite animations allow you to create loops or ongoing effects. This is useful for loading indicators or any animation that needs to repeat indefinitely.

### ***Example: Infinite Rotation***

```
@Composable
fun InfiniteRotation() {
    val rotation by animateFloatAsState(
        targetValue = 360f,
        animationSpec = infiniteRepeatable(
            animation = tween(durationMillis = 1000),
            repeatMode = RepeatMode.Restart
    )
}
```

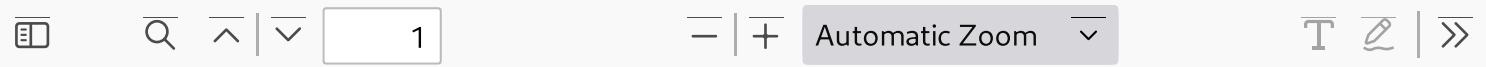
```
        )  
    )  
  
Box(  
    modifier = Modifier  
        .size(100.dp)  
        .graphicsLayer(rotationZ = rotation)  
        .background(Color.Cyan)  
    )  
}
```

- The box continuously rotates 360 degrees, using `infiniteRepeatable` to keep the animation running in a loop.

## Conclusion

Animations in Jetpack Compose provide an easy way to enhance the user experience with fluid transitions and interactive feedback. Whether you're using simple `animate*AsState` functions or more complex animations like `updateTransition` and `spring`, Compose offers a variety of tools to animate your UI. By understanding these animation tools, you can make your Android apps feel more dynamic, engaging, and responsive.

# Compose Animation Cheat Sheet



# Mobile Accessibility

Mobile accessibility refers to designing mobile apps in such a way that they can be used by everyone, including individuals with disabilities. The **Web Content Accessibility Guidelines (WCAG)**, developed by the W3C, provide guidelines for creating accessible digital content. These guidelines are organized into four principles: **Perceivable**, **Operable**, **Understandable**, and **Robust**. These principles serve as a foundation for building apps that are inclusive and usable by as many people as possible.

## Principle 1: Perceivable

The Perceivable principle focuses on making sure that users can perceive the information and elements within the app. If users cannot perceive content, they cannot use it, no matter how operable or understandable the app is.

### **Key Points:**

- **Text Alternatives:** Provide text alternatives (such as alt text for images) so that users who are blind or have low vision can access the content.
- **Time-Based Media:** Offer alternatives for time-based media such as audio and video, such as captions or transcripts.
- **Adaptable Content:** Ensure that content can be presented in different ways, such as through high-contrast modes or screen readers, without losing meaning.
- **Color Contrast:** Use sufficient color contrast between text and background so that people with low vision can read the content easily.

### **Example:**

- Providing alt text for images, so a screen reader can describe the image to a user who is visually impaired.

## Principle 2: Operable

The Operable principle focuses on ensuring that users can operate the app's interface. The app must be fully functional and navigable using different inputs, including keyboards, touchscreens, and voice commands.

### **Key Points:**

- **Keyboard Accessibility:** Ensure that all interactive elements can be navigated and activated

via keyboard for users with mobility impairments.

- **Accessible Touch Targets:** Ensure that buttons, links, and other interactive elements are large enough to be tapped by users with limited dexterity.
- **Navigation:** Provide clear navigation mechanisms, such as a menu, to help users with disabilities easily explore the app's content.
- **Time Controls:** Allow users to control or adjust time-based events, such as timeouts or animations, especially for users with cognitive impairments.

#### ***Example:***

- Ensuring that buttons on a mobile app are large enough to be tapped comfortably, and providing options to increase the size of touch targets for users with motor impairments.

## **Principle 3: Understandable**

The Understandable principle ensures that the app's content and functionality are easy to understand. This includes both the information presented to the user and the operations they can perform.

#### ***Key Points:***

- **Clear and Simple Language:** Use simple, concise, and easy-to-understand language to convey your message.
- **Error Prevention and Suggestions:** Provide users with clear instructions and error suggestions. For instance, if they make an input error, offer suggestions to help correct it.
- **Consistent Navigation:** Ensure that the app's layout and design are consistent across all screens to help users predict where they can find certain functions.
- **Input Assistance:** Provide forms and input fields that guide the user on what is expected. Use hints, labels, and error messages effectively.

#### ***Example:***

- When filling out a form, provide instructions or examples to explain what format is expected (e.g., a phone number should include country code).

## **Principle 4: Robust**

The Robust principle ensures that the app works well with a wide variety of devices, assistive technologies, and future advancements in technology. The idea is to ensure that the app remains accessible as technology evolves.

#### ***Key Points:***

- **Compatibility with Assistive Technologies:** Make sure the app is compatible with a variety of assistive technologies, such as screen readers, speech recognition software, or alternative input devices.
- **Semantic HTML and ARIA:** In case of web apps, use semantic HTML and ARIA (Accessible Rich Internet Applications) roles and properties to ensure the app works well with screen readers and other assistive tools.
- **Future-Proofing:** Develop the app using flexible and standardized technologies that can adapt to future updates, devices, and accessibility features.

***Example:***

- Ensuring that the app remains compatible with both older and newer versions of assistive technology, and works seamlessly on different devices (phones, tablets, wearables).

## Conclusion

Accessibility in mobile app development is crucial for creating inclusive digital experiences. By following the four principles of accessibility—**Perceivable, Operable, Understandable, and Robust**—developers can ensure that their apps are usable by everyone, including people with disabilities. Adopting accessibility principles not only improves user experience but also broadens the app's reach and complies with legal and ethical standards in many regions.

# \* References and Additional Reading \*

## **Mobile App Design:**

- Jarret, C. And Gafney, G. (2009). Forms that Work: Designing Web Forms for Usability, Chapter 5, 6.
- Theresa Neil, 2014, Mobile Design Pattern Gallery: UI Patterns for Smartphone Apps, O'Reilly Media; 2 edition. Chapter 2.
- Shneiderman, Plasiant, Cohen, Jacobs (2014) *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Pearson. Chapter 6
- Luke Wroblewski (2015), Showing Passwords on Log-In Screens  
<http://www.lukew.com/ff/entry.asp?1941>
- Luke Wroblewski (2013) Designing for Thumb Flow <http://www.lukew.com/ff/entry.asp?1734>=
- Mobile Form Design Strategies by Chui Chui Tan <http://bit.ly/1g36jwU>
- How the float label pattern started by Matt Smith (2013) <http://mds.is/float-label-pattern>

## **Mobile Accessibility by W3C:**

- <https://www.w3.org/TR/mobile-accessibility-mapping/>

## **Material Design for Jetpack Compose by Google:**

- Overview: <https://m3.material.io/develop/android/jetpack-compose>
- Developer Documentation:  
<https://developer.android.com/jetpack/androidx/releases/compose-material3>
- Material 3 Introduction by Philipp Lackner: <https://www.youtube.com/watch?v=l3eT32LXAKc>

## **Transitions and Animations:**

- A Video: Building beautiful transitions with Material Motion for Android  
<https://www.youtube.com/watch?v=iuvmnxTRgRM>
- Choose an animation API:  
<https://developer.android.com/develop/ui/compose/animation/choose-api>

# Animations in Compose Cheat Sheet

## Animate appearing / disappearing 🎉

Use `AnimatedVisibility` to hide/show a Composable.  
⚠️ Children inside `AnimatedVisibility` can use `Modifier.animateEnterExit` for their own enter/exit transition.

```
AnimatedVisibility(visible) {  
    // your composable here  
}
```

⚠️ Adds / Removes the item from composition.  
OR

```
val animatedAlpha = animateFloatAsState(/*your  
changing value here*/)  
Column(modifier = Modifier.graphicsLayer(  
    alpha = animatedAlpha.value  
) {  
    // your composable here  
}
```

⚠️ Keeps item in the composition - animates its alpha instead

## Animate size changes 🎊➡️🎉

Use `animateContentSize` for animations between composable size changes.

```
var message by remember { mutableStateOf("Hello") }  
Box(  
    modifier = Modifier  
        .background(Color.Blue)  
        .animateContentSize()  
) {  
    Text(text = message)  
}
```

⚠️ Ordering of the modifier matters, make sure to place it before any size modifiers.

## Animate individual properties from state 💡

Use `animateAsState` APIs for any property animations based on state (for instance: state from a ViewModel). Use the variable in a Modifier or Canvas drawing properties.

```
val animatedColor = animateColorAsState(/*your  
changing value here*/)  
Column(modifier = Modifier.drawBehind {  
    drawRect(animatedColor.value)  
) {  
    // your composable here  
}  
  
animateDpAsState()  
animateOffsetAsState()  
animateFloatAsState()  
animateSizeAsState()  
animateRectAsState()  
animateIntAsState()
```

## Animated Vector Drawable ~

Animate VectorDrawable paths with `animatedVectorResource`

```
val image = AnimatedImageVector  
    .animatedVectorResource(R.drawable.avd_hourglass)  
  
var atEnd by remember { mutableStateOf(false) }  
Image(  
    painter = rememberAnimatedVectorPainter(image, atEnd),  
    modifier = Modifier.clickable {  
        atEnd = !atEnd  
    },  
    contentScale = ContentScale.Crop,  
    contentDescription = "hourglass"  
)
```

## Lazy list item changes 🐻

Animate item reordering of items in a list use `animateItemPlacement`.

```
LazyColumn {  
    items(books, key = { it.id }) {  
        Row(modifier = Modifier.animateItemPlacement(  
            tween(durationMillis = 250)  
        )) {  
            // ...  
        }  
    }  
}
```

⚠️ Make sure to specify a key for the correct replacement.  
⚠️ Additions and deletions are coming soon.

## Animate changes between Composables 🌟

Change between different Composables based on state changes using `AnimatedContent`. For a simple fade between the two, use `CrossFade` instead.

```
AnimatedContent(state) { targetState ->  
    when (targetState) {  
        Loaded -> /* your composable */  
        Loading -> /* your composable */  
    }  
}
```

## Animate multiple properties at once 💕

Use the `Transition` API to animate multiple properties at the same time when transitioning between different states.

```
var currentState by remember { mutableStateOf(Collapsed) }  
val transition = updateTransition(currentState)  
  
val rect by transition.animateRect { state ->  
    when (state) {  
        Collapsed -> Rect(0f, 0f, 100f, 100f)  
        Expanded -> Rect(100f, 100f, 300f, 300f)  
    }  
}  
val borderWidth by transition.animateDp { state ->  
    when (state) {  
        Collapsed -> 1.dp  
        Expanded -> 0.dp  
    }  
}
```

## Repeat an animation 🔄

Use `infiniteRepeatable` to continuously repeat your animation. Change `RepeatMode` to specify how it should go back and forth.

Use `finiteRepeatable` to repeat a set number of times.

```
val infiniteTransition = rememberInfiniteTransition()  
val color by infiniteTransition.animateColor(  
    initialValue = Color.Red,  
    targetValue = Color.Green,  
    animationSpec = infiniteRepeatable(  
        animation = tween(1000, easing = LinearEasing),  
        repeatMode = RepeatMode.Reverse  
    )  
)  
  
Box(modifier = fillMaxSize().background(color))
```

## Start an animation on launch 🚀

`LaunchedEffect` is run when a Composable enters the composition.

```
val alphaAnimation = remember {  
    Animatable(0f)  
}  
LaunchedEffect(key) {  
    alphaAnimation.animateTo(1f)  
}  
Box(modifier = Modifier.alpha(alphaAnimation))
```

⚠️ If used in a lazy layout, `LaunchedEffect` will be called every time you scroll your view on and off screen. You may need to `hold your state` outside of the lazy composable to have the animation only run once.

## Sequential animations ➔➔➔

Use the `Animatable` coroutine APIs to do sequential animations.

Calling `animateTo` on the `Animatable` one after the other will wait for the previous animations to finish before proceeding to the next as its a suspend function.

```
val alphaAnimation = remember { Animatable(0f) }  
val yAnimation = remember { Animatable(0f) }  
  
LaunchedEffect("animationKey") {  
    alphaAnimation.animateTo(1f)  
    yAnimation.animateTo(100f)  
    yAnimation.animateTo(500f, animationSpec = tween(100))  
}
```

## Animation specs 📈

Specify how animation value should transform between the start and target values:

👉 `tween` - animate (with easing) **between** two values with a duration.

👉 `spring` - physics-based animation with damping ratio and stiffness (no duration)

👉 `keyframes` - spec for specifying different specs at different key frames of the animation.

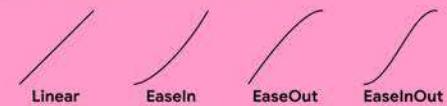
👉 `repeatable` - duration based spec runs repeatedly for # of iterations.

👉 `infiniteRepeatable` - duration based spec runs forever

👉 `snap` - spec that instantly switches to new value

## Easing functions 🌈

Describe the rate of change over time for an animation. `Linear` moves at the same constant speed. Others like `EaseIn`, are slow to start then progress to a linear function.



## Concurrent animations ➔➔➔

Use coroutine APIs, or Transition API (see transition block for alternative) for concurrent animations.

Using `launch` in a coroutine context will launch the animations at the same time.

```
val alphaAnimation = remember { Animatable(0f) }  
val yAnimation = remember { Animatable(0f) }
```

```
LaunchedEffect(key) {  
    launch {  
        alphaAnimation.animateTo(1f)  
    }  
    launch {  
        yAnimation.animateTo(100f)  
    }  
}
```

## Learn more 🌐

docs: [goole.com/compose-animation](https://goole.com/compose-animation)

codelab: [goole.com/compose-animation-codelab](https://goole.com/compose-animation-codelab)



MONASH  
University

**FIT2081**  
**Mobile Application**  
**Development**



# Advanced Mobile UI Design

**Week 10**

Jiazhou 'Joe' Liu

# Learning Objectives

- **Design Style and Guidelines**
  - Mobile Design Styles and Paradigms
  - Material Design
  - Form Guidelines
  - Mobile Accessibility Guidelines

# Design Styles

# Skeuomorphism

**Skeuomorphism** is a design concept where digital interface elements mimic real world physical objects.

Skeuomorphism uses **textures**, **shadows** and **gradients**, and **ornamental visuals** (such as spiral bindings, leather and wood) to suggest interactivity and create a familiar, tangible feel in digital interfaces.



*Image credit: StarLotus7 Posted in [Frutiger AeroFrutiger MetroSkeuomorphism](#)*



# Skeuomorphism (cont.)

In the early days of the digital age, skeuomorphism worked well by reducing the learning curve

Yet, over time, users learned how to interact with computers and mobile interfaces, and skeuomorphism was no longer necessary

Because the key limitation is that it can **clutter the interface** with unnecessary visual details, making apps **harder to scale, slow to load, and less adaptable** to modern minimalist design trends or diverse screen sizes.



Image credit: Robert Irish Posted in [AppleInsider](#)

# Flat Design

Flat design was introduced in opposition to skeuomorphism

- The core principle is **simplicity** and **minimalism** (using fewer objects)
- It drops ornamental visuals, shadows, highlights and gradients
- It uses **flat shapes** and tile menus in a 2D environment
- It uses **vibrant colours** and **clear typography**
- It focuses on functionality and usability
- It increases efficiency and responsiveness

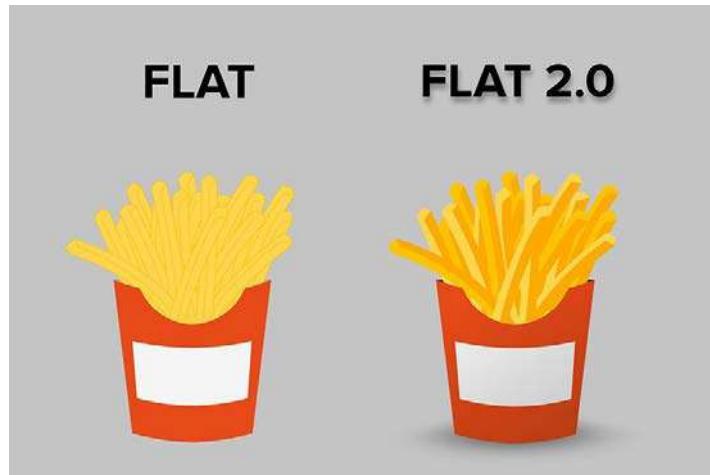


Metro interface

# Flat Design 2.0

Flat design removes all visual clues but the total **lack of the 3D effects** and **visual affordances** undermines usability

- E.g., it is not clear if an object is clickable or not
- Flat 2.0 adds shadows and highlights to create a 3D experience and make it easier to identify affordances



[Image credit](#)

# Material Design

**Material design** provides guidelines, tools, and code to support the best practices of user interface design for different devices and platforms

- The guidelines not just about the shapes and colours but also about the **user interactions, user behaviour, and accessibility**
- It is a **design language** and a living document
- It is **expandable**

# Material Design History

Material design was introduced by Google

- **Material Design 1 (M1)**: introduced in 2014, and it is archived
- **Material Design 2 (M2)**: launched in 2018, introduced code for components  
<https://m2.material.io/design>
- **Material Design 3 (M3)**: launched in 2021, includes new features
- All in one place: <https://m3.material.io/>



# Material Design (M3) Foundations

- **Accessibility:** Design apps that can be used by everyone, including users with disabilities, by following inclusive design practices.
- **Content design:** Craft clear, purposeful text and visuals that guide users and enhance their overall experience.
- **Customizing Material:** Adapt Material Design components and theming to reflect your app's unique brand and identity.
- **Design tokens:** Use standardized variables (like color, typography, spacing) to create consistent, scalable, and easily themeable UI.
- **Interaction states:** Represent different states of user interaction (like hover, focus, pressed) to give clear feedback and improve usability.
- **Layout:** Organize content effectively across different screen sizes and orientations to ensure clarity and responsiveness.
- **Material A-Z**
- **Building for all:** Design with empathy by considering diverse needs, environments, and devices to create apps that serve everyone.

# Build Jetpack Compose UIs with Material Design 3

## Dependency

To start using Material 3 in your Compose app, add the [Compose Material 3](#) dependency to your build.gradle files:

```
implementation "androidx.compose.material3:material3:$material3_version"
```

## Experimental APIs

Some M3 APIs are considered experimental. In such cases you need to opt in at the function or file level using the [ExperimentalMaterial3Api](#) annotation:

```
@OptIn(ExperimentalMaterial3Api ::class)
@Composable
fun TopAppBarExample() {
    val scrollBehavior = TopAppBarDefaults.enterAlwaysScrollBehavior(rememberTopAppBarState())
```



# Material Design Styles

# Colour

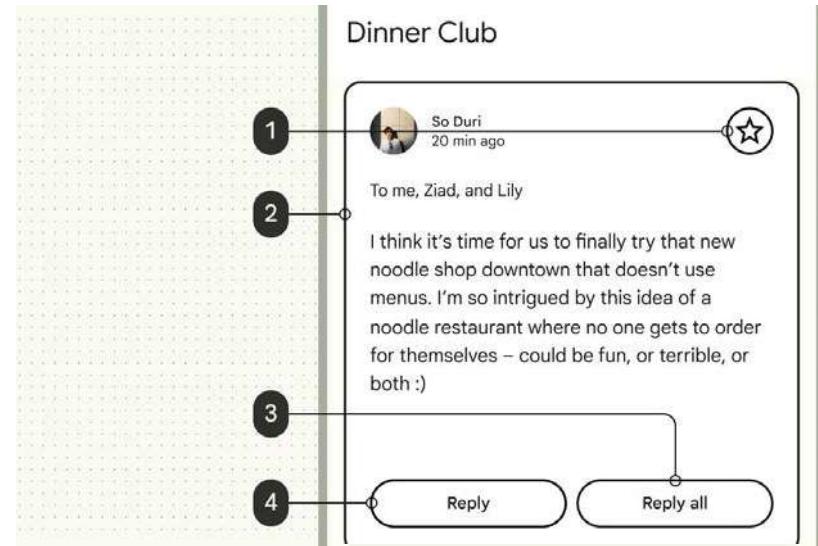
Color in Material Design helps **express brand identity, guide user attention, and communicate actions** through a well-structured system of palettes, contrast rules, and dynamic theming.

It ensures **accessibility, visual harmony, and consistency** across light and dark modes using standardized roles like primary, secondary, and surface colors.

[Material Theme Builder plugin](#) on Figma Design platform



LIVE DEMO\_



*“paint-by-number”*



MONASH  
University

# Colour Roles in M3

Color roles have six main categories:

**Primary, Secondary, Tertiary, Error, Surface, and Outline.**

They are assigned to UI elements based on **emphasis**, **container type**, and **relationship** with other elements. This ensures proper contrast and usage in any color scheme.

The color system is built on accessible color pairings. These color pairs provide an **accessible minimum 3:1 contrast**.

Roles are implemented in design and code through **tokens**. A design token represents a small, reusable design decision that's part of a design system's visual style.

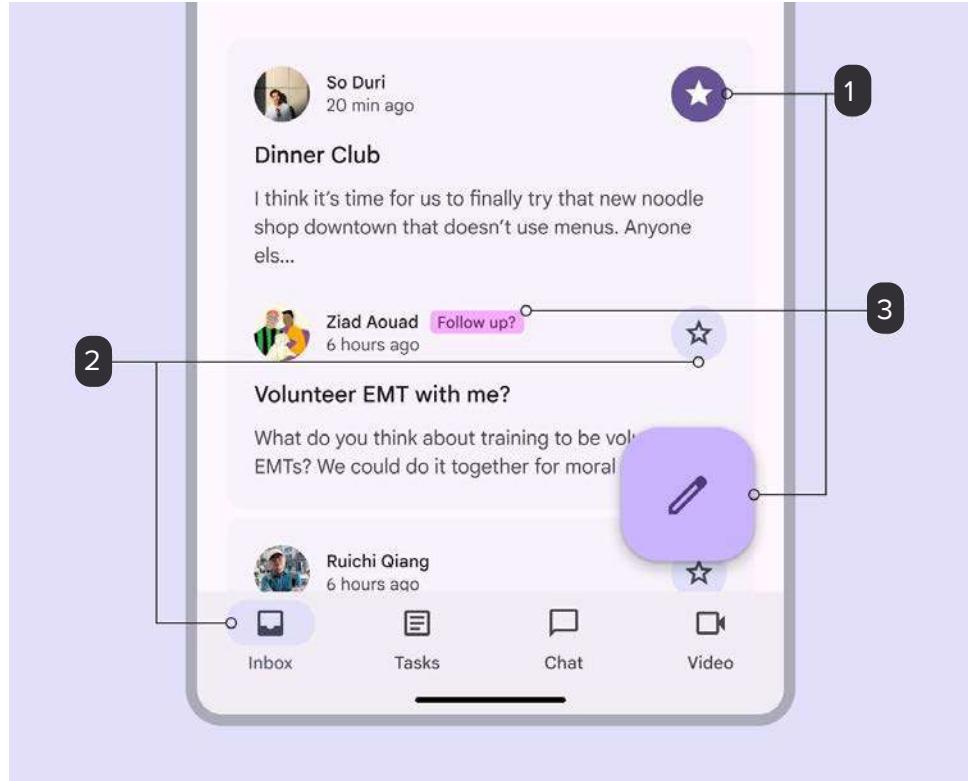
Primary	Secondary	Tertiary	Error
On Primary	On Secondary	On Tertiary	On Error
Primary Container	Secondary Container	Tertiary Container	Error Container
On Primary Container	On Secondary Container	On Tertiary Container	On Error Container
Primary Fixed	Primary Fixed Dim	Secondary Fixed	Secondary Fixed Dim
On Primary Fixed	On Secondary Fixed	On Tertiary Fixed	
On Primary Fixed Variant	On Secondary Fixed Variant	On Tertiary Fixed Variant	
Surface Dim	Surface	Surface Bright	Inverse Surface
Surface Container Lowest	Surface Container Low	Surface Container	Inverse On Surface
Surface Container High	Surface Container Highest		Inverse Primary
On Surface	On Surface Variant	Outline	Outline Variant
		Stroke	Shadow

<https://m3.material.io/styles/color/roles>



# Colour Roles in M3 (example)

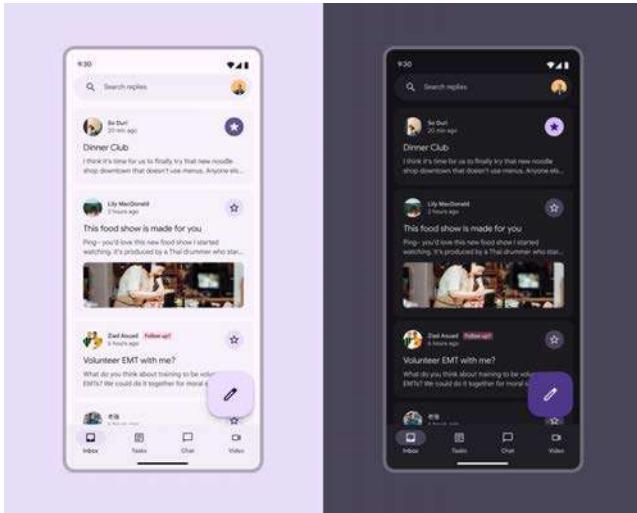
1. **Primary roles (labelled as 1)** are for important actions and elements needing the most emphasis, like a FAB to start a new message.
2. **Secondary roles (labelled as 2)** are for elements that don't need immediate attention and don't need emphasis, like the selected state of a navigation icon or a dismissive button.
3. **Tertiary roles (labelled as 3)** are for smaller elements that need special emphasis but don't require immediate attention, such as a badge or notification.



# Colour Schemes in M3

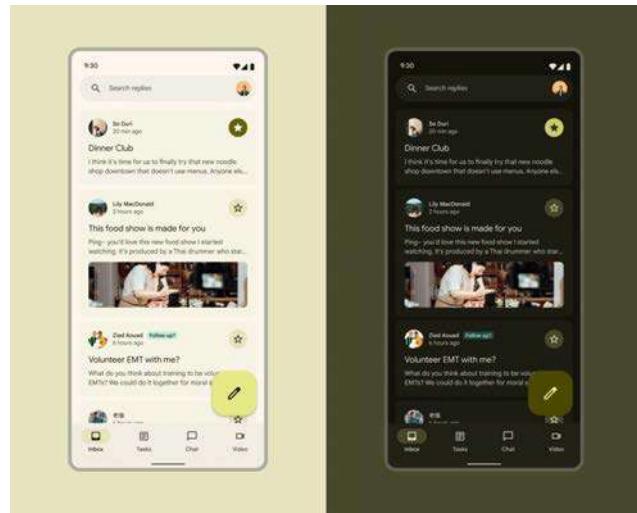
## Static colour

UI colors that don't change based on the user's wallpaper or in-app content. Once assigned to their respective color roles and UI elements, the colors remain constant.



## Dynamic colour

Dynamic color takes a single color from a user's wallpaper or in-app content and creates an accessible color scheme assigned to elements in the UI.



# Material Design Guidelines for Colour

## Start with a Source Color

Pick one or two key brand colors (source colors), and Material 3 will generate cohesive tonal palettes for light and dark themes.

## Use Semantic Color Roles

Apply colors using defined roles like `primary`, `onPrimary`, `surface`, `background`, `error`, and `outline`—each serves a functional purpose (e.g., `primary` highlights key actions).

## Use Tonal Palettes Instead of Manual Color Picking

Rather than selecting individual shades, use tones from the generated palettes (0–100) to maintain consistency and harmony across components.

## Support Dynamic Color (for Android 12+)

Allow the system to adapt your color scheme based on the user's wallpaper for personalized but consistent themes.

## Apply Contrast Ratios for Accessibility

Follow the contrast rules defined by Material 3 to ensure readable text and visible UI elements across backgrounds.

## Use Layered Surface Colors to Indicate Elevation

Differentiate elevated surfaces (cards, sheets) by applying higher tonal values of `surface` or using `surfaceContainer` variants.

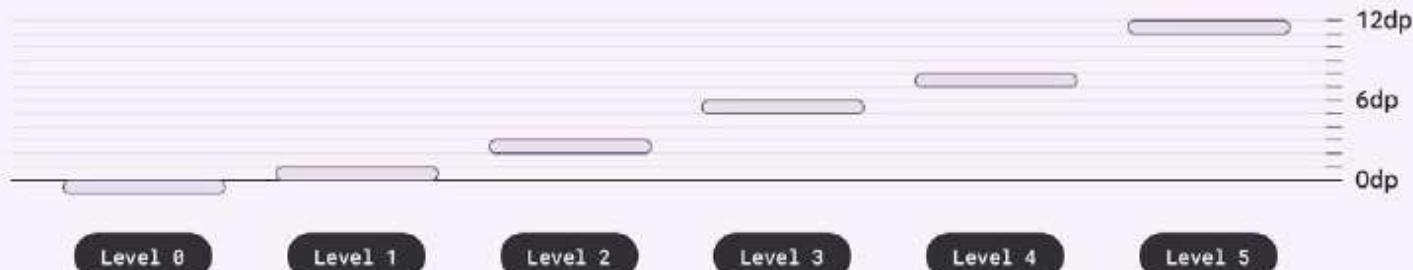
## Balance Color and Neutral Space

Don't overuse vibrant roles like `primaryContainer` or `tertiary`; mix them with neutral tones for visual comfort and focus.

# Elevation

Elevation is the distance between two surfaces on the z-axis

- Elevation is applied to all surfaces and components
- Elevation can be shown as tonal surface colors or shadows
- Avoid changing the default elevation of Material 3 components
- Stick to using a small amount of elevation levels



# Elevation Types

## Tonal elevation

Tonal elevation uses color to define elevation levels. Tonal elevation gets darker the higher up the elevation.



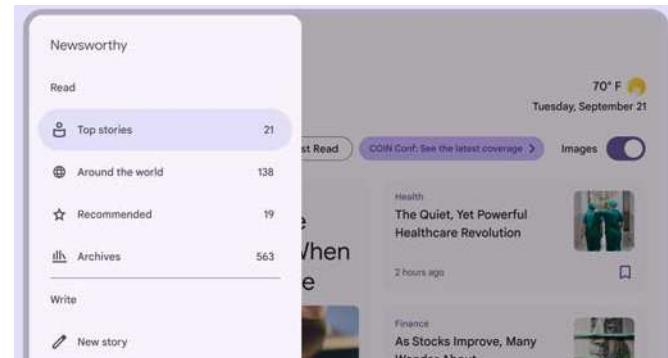
## Shadow elevation

Shadow elevation is simply using shadows to set levels of elevation.



# Elevation Guidelines

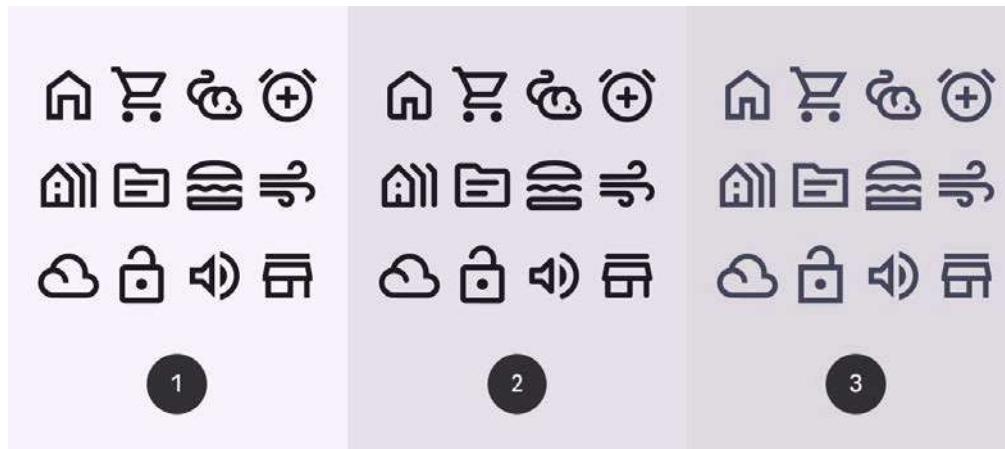
- Combining tonal and shadow elevation
  - For interactive components, edges must create sufficient contrast between surfaces
- When it comes to applying shadows, less is more. The fewer levels in your UI, the more power they have to direct attention and action.
- Using scrims for extra focus
  - Scrims are backdrops that use a dark, semi-transparent (**32% opacity**) background, covering the entire screen behind the element in focus.
- When a background is patterned or visually busy, the hairline style might not provide sufficient protection.
- Elements can temporarily lift on focus, selection, or another kind of interaction, like swipe.



# Icons

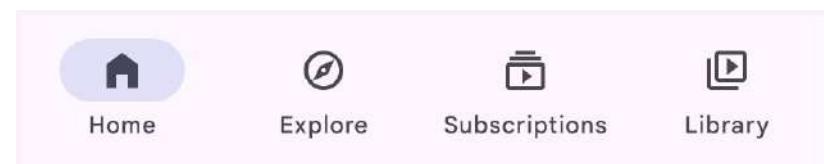
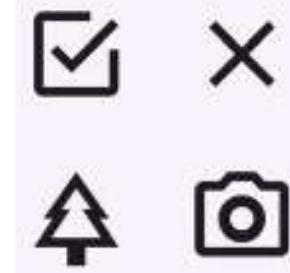
Icons are small symbols to easily identify actions and categories

- Material Symbols icons can be found at: <https://fonts.google.com/icons>
- Use the Material Symbols variable font (screenshot below: outlined (1), rounded (2), and sharp (3)) to enable dynamic styling in product
- You can change the weight, fill, optical size, and grade of variable font icons



# Design Principles for Icons

- Simplify icons for greater clarity and legibility
- Make icons graphic and bold
- Use and maintain a consistent visual style throughout one icon set
- Standard icons are displayed as 24dp x 24dp.
  - Icons support additional sizes: 20dp, 40dp, and 48dp.
- Use labels when icons and symbols are more abstract
  - Label text provides short, meaningful descriptions when symbols are more abstract.  
This can prove helpful in the case of navigation.
  - Remember that navigation items must have labels for clarity and accessibility
- Consider cultural interpretations of symbols.



# Motion

Motion makes a UI expressive and easy to use

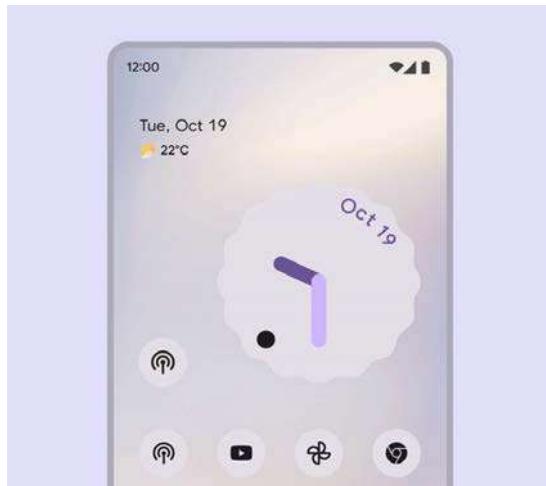
- Use motion in transitions and interactions
- There are six common transition patterns:
  1. Container transform
  2. Forward and backward
  3. Lateral
  4. Top level
  5. Enter and exit
  6. Skeleton loaders



# Motion Pattern Examples

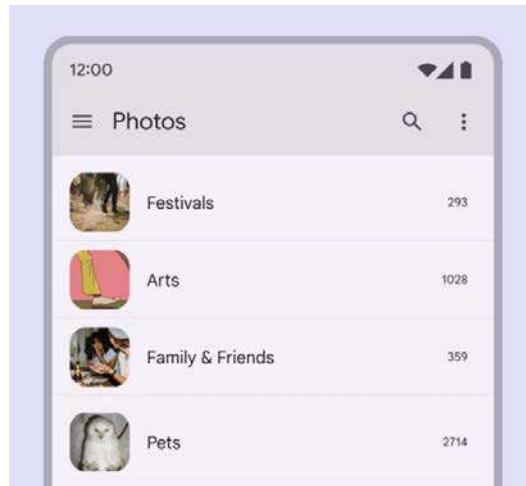
## Container Transform

Seamlessly transform an element to show more detail



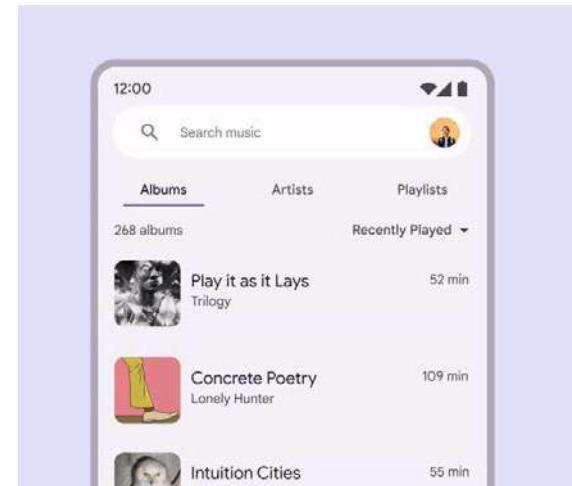
## Forward and Backward

Navigating between screens at consecutive levels of hierarchy



## Lateral

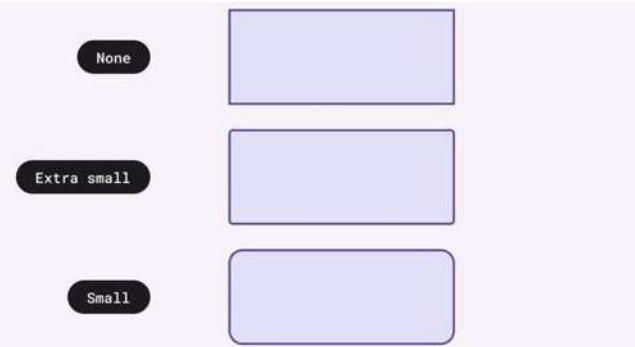
Navigating between peer content at the same level of hierarchy



# Shape

Shape can direct attention, communicate state, and express brand

- 7 styles are assigned to components based on the desired amount of roundedness. Square-cornered shapes are “None” and slightly rounded shapes are “extra-small,” while entirely circular shapes are “full.”

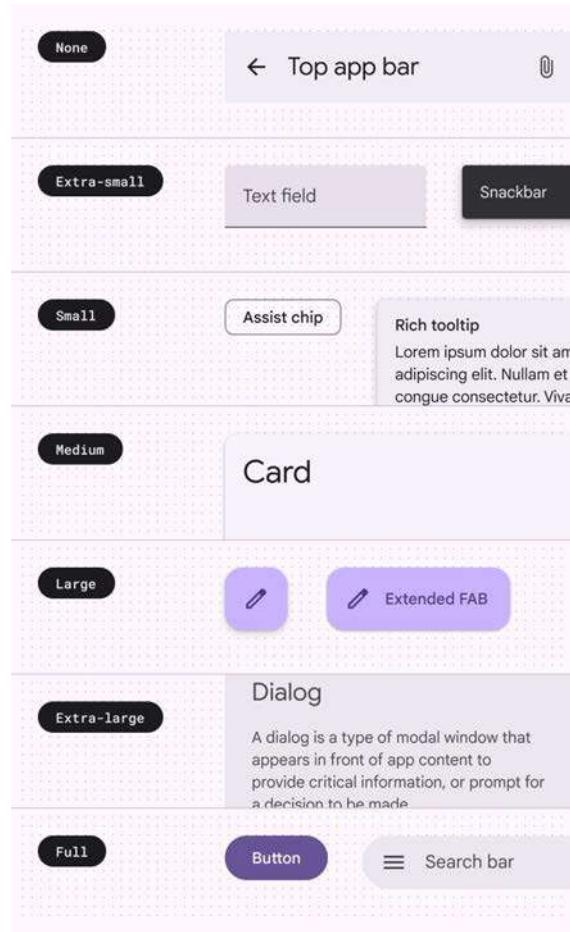


Shape style	Component
None	Banners, Bottom app bars, Full-screen dialogs, Lists, Navigation bars, Navigation rails, Progress indicators, Search view (full-screen), Side sheets (docked), Tabs, Top app bars
Extra small	Autocomplete menu, Select menu, Snackbars, Standard menu, Text fields
Small	Chips, Rich tooltip
Medium	Cards, Small FABs
Large	Extended FABs, FABs, Navigation drawers
Extra large	Bottom sheets (docked), Dialogs, Floating sheets, Large FABs, Search view (docked), Time picker, Time input
Full	Badge, Buttons, Icon buttons, Sliders, Switches, Search bar



# Shape Style Examples

Shape style	Component
<b>None</b>	Banners, Bottom app bars, Full-screen dialogs, Lists, Navigation bars, Navigation rails, Progress indicators, Search view (full-screen), Side sheets (docked), Tabs, Top app bars
<b>Extra small</b>	Autocomplete menu, Select menu, Snackbars, Standard menu, Text fields
<b>Small</b>	Chips, Rich tooltip
<b>Medium</b>	Cards, Small FABs
<b>Large</b>	Extended FABs, FABs, Navigation drawers
<b>Extra large</b>	Bottom sheets (docked), Dialogs, Floating sheets, Large FABs, Search view (docked), Time picker, Time input
<b>Full</b>	Badge, Buttons, Icon buttons, Sliders, Switches, Search bar



# Typography

Typography helps make writing legible and beautiful

- **Variable fonts** are a new technology offering more typographic control
  - A variable font allows granular style adjustments along different axes, such as weight or grade. In product, using a variable font is more efficient than multiple static fonts, improving product performance.
- Material 3 helps **scale typography decisions across devices**, including font, line height, size, tracking, and weight
- The default typeface for Android is [Roboto](#). It is a type of Sans-serif font.
- [Roboto Serif](#) is another variable font family, designed to create a comfortable reading experience.
- [Noto](#) is a global font collection for all modern and ancient languages.

Serif

Sans-Serif

Abc Abc



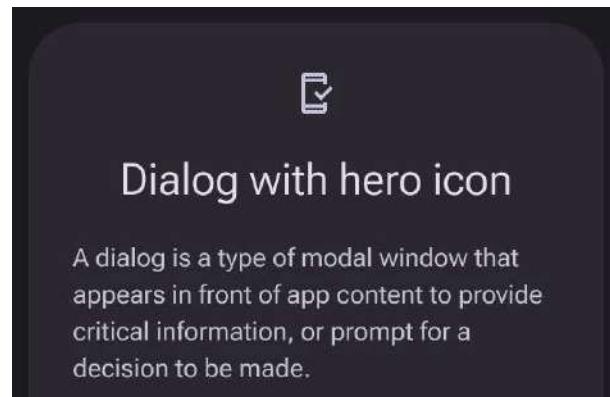
# Typography Styles

Material 3 organises typography styles into five roles:

- **Display styles** are reserved for short, important text or numerals. They work best on large screens.
- **Headlines** are best-suited for short, high-emphasis text on smaller screens.
- **Titles** are smaller than headline styles, and should be used for medium-emphasis text that remains relatively short.
- **Body** styles are used for longer passages of text.
- **Label** styles are smaller, utilitarian styles, used for things like the text inside components or for very small text in the content body, such as captions.



Display styles



Headline styles

# Mobile Forms

# Mobile Form Guidelines

- Create a smooth and natural conversational flow
  - Logical grouping and sequencing of fields
  - Differentiate grouped items
- Meaningful and familiar field labels
  - Consistent terminology and abbreviations
  - Provide hints and examples if needed
- Visually appealing layout (alignment)
- Optional and required fields clearly marked (\*)
- Minimise the number of the inputs
  - Combine similar input fields
  - Eliminate redundant and less important entries

The screenshot displays a mobile application's user interface for adding a medication. At the top, the title 'Add Medication' is centered. Below it is a text input field labeled 'Medication Name' with the placeholder text 'Placeholder'. To the right of this field is a row containing two smaller input fields: 'Dosage' (containing the value '1') and 'Recurrence' (set to 'Daily'). Further down is a date input field showing 'End Date' with the value 'August 28, 2017'. Below these is a section labeled 'Times Of Day' with four checkboxes: 'Morning', 'Afternoon', 'Evening', and 'Night', all of which are checked. At the very bottom of the screen is a large, prominent purple rectangular button with the word 'Save' in white text.

[Image credit](#)



# Mobile Form Guidelines (Cont.)

- Use lists if possible to minimise errors
- Validate as early as possible but don't break the conversational flow
  - Inline feedback wherever appropriate
- Error prevention and error messages
  - Use meaningful and clear messages
  - Error correction: refocus on the field containing the error
- Immediate feedback and completion feedback
  - Provide the user with the feedback on the current status and progress

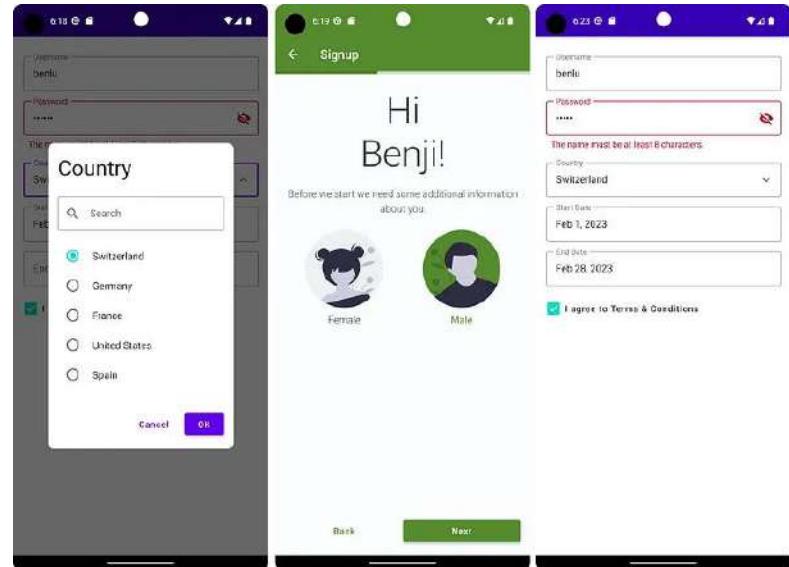


Image credit: [Benji @ Medium](#)



MONASH  
University

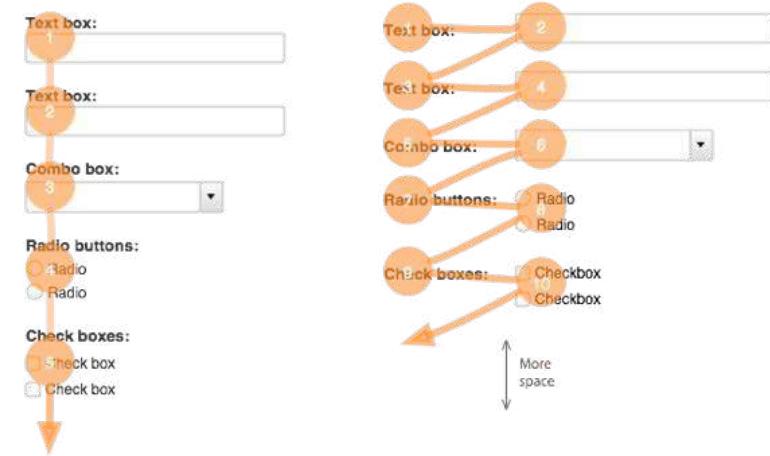
# Vertical Forms vs Horizontal Forms

- **Horizontal forms** using left and right aligned labels:

- labels can be truncated or overlapped
  - require two visual directions to fill out

- **Vertical forms** using top aligned labels

- Better spacing
  - Better visual flow
  - But can make the form long
    - Break forms into multiple pages

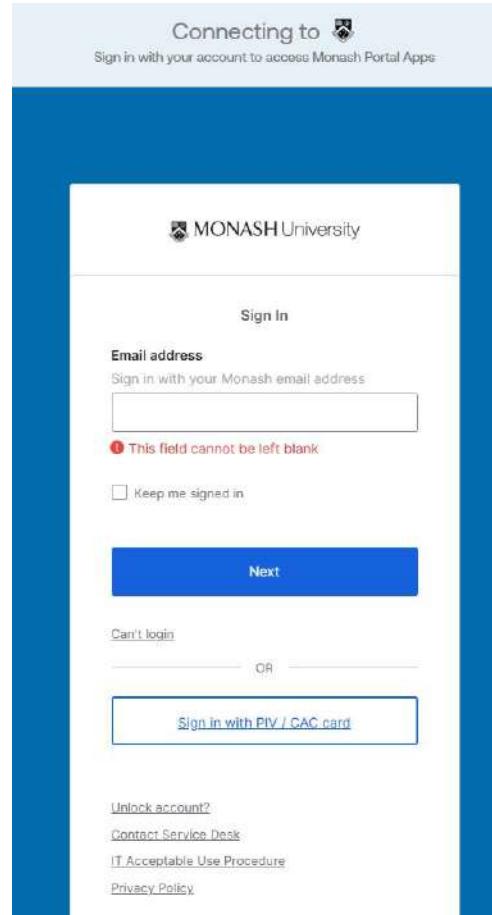


Top aligned labels (vertically arranged)  
5 visual fixations  
1 visual direction  
less vertical space

Left/right aligned labels (vertically arranged)  
10 visual fixations  
2 visual directions  
more vertical space

# Mobile Forms: Sign-in

- Forms with only few inputs
- Login or Sign-in for existing users
- An option to remember the password
- An option if the password is forgotten



# Sign-in Forms: Passwords

- In sign-in forms, passwords are usually masked
- Research shows that most users forget their passwords or have trouble remembering them
- Password masking can be a usability issue
- To increase usability, many designers adopted clear/visible passwords but unmasking passwords can cause security issues
- Better to provide the user with both options: show/hide passwords
  - <https://developer.android.com/develop/ui/compose/quick-guides/content/show-hide-password>



# Mobile Accessibility

# Mobile Accessibility at W3C

W3C (the World Wide Web Consortium) includes a number of organisations that join together to develop standards and guidelines based on the principles of **accessibility, internationalization, privacy and security**

**Mobile Accessibility at W3C** includes guidelines for building mobile applications that are more accessible to people with disabilities

Current website: <https://www.w3.org/WAI/standards-guidelines/mobile/>

W3C WAI accessibility guidelines (2015) <https://www.w3.org/TR/mobile-accessibility-mapping/>

# Principle 1: Perceivable

## 1.1. Small Screen Size

- Minimise the amount of information (e.g. fewer content items or images)
- Provide a reasonable default size for contents and touch controls to minimise the need to zoom in and out

## 1.2. Zoom/Magnification

- Provide methods that allow the user to control content/text size

## 1.3. Contrast

- Mobile devices are used in environments with different lightings so consider the right contrast ratios for text
- The contrast ratio is calculated based on the colour of text and its background, e.g., a recommended one is 4.5:1 (black text with the white background)

# Principle 2: Operable

- 1.1. Keyboard Control for Touchscreen Devices
  - Build applications that support external/alternative keyboards
- 1.2. Touch Target Size and Spacing
  - Targets must be big enough, with enough space from each other
- 1.3. Touchscreen Gestures
  - Mobile devices can be operated via gestures on a touchscreen
  - Gestures include simple tapping or multiple tapping or involving multiple fingers
  - Gestures should be as easy as possible
  - For complex gestures, design alternatives such as simple tap or swipe gestures

# Principle 2: Operable (Cont.)

## 1.4. Device Manipulation Gestures

- Physical manipulation of the device such as shaking or tilting
- Make sure to provide alternatives such as touch and keyboard operable control options

## 1.5. Placing buttons where they are easy to access

- Place buttons where they can be easily reached if the user holds the device in different positions

# Principle 3: Understandable

- 1.1. Changing Screen Orientation (Portrait/Landscape)
  - Support both orientation modes, and do not expect users to rotate
- 1.2. Consistent Layout
  - “Components that are repeated across multiple pages should be presented in a consistent layout.”
- 1.3. Positioning important page elements before the page scroll
  - This will assist users with low vision and those with cognitive impairments

# Principle 3: Understandable (Cont.)

- 1.4. Grouping operable elements that perform the same action
  - Multiple elements performing the same action (e.g. link icon with link text) should be contained within the same actionable element
- 1.5. Provide clear indication that elements are actionable
  - Use visual features to signal an actionable element (affordances)
- 1.6. Provide instructions for custom touchscreen and device manipulation gestures
  - Custom gestures can be challenging so provide instructions such as tooltips and tutorials

# Principle 4: Robust

- 1.1. Set the virtual keyboard to the type of data entry required
- 1.2. Provide easy methods for data entry
  - Reduce the amount of text entry by providing menus, lists, or autoentry
- 1.3. Support the characteristic properties of the platform
  - Mobile devices provide features to help users with disabilities but these features vary depending on the type of device and OS

# Reminders and Announcements

## Week 12 Online Off-campus eAssessment – Monday, 1pm (AEDT)

- 20 MCQs covering all contents from Week 1 to Week 10
- Mock exam environment will be released soon

## Lab activities this week

- Material Design Showcase
- Compose Animation Showcase
- Authentication with Google Firebase

## Week 11

- *Build Types, Product Flavours, Build Variants*
- *Building APKs & Submitting to Google Play App Store*

# Reference and Further Reading

- Pari Delir Haghghi (S1 2024) Mobile User Interface Design [PowerPoint slides], FIT5046: Mobile and Distributed Computing Systems, Monash University.
- Material Design for Jetpack Compose:
  - **Jetpack Compose:** <https://m3.material.io/develop/android/jetpack-compose>
  - **Design for Android:** <https://developer.android.com/design/ui>
- Mobile Forms Guideline:
  - Jarret, C. And Gafney, G. (2009) *Forms that Work: Designing Web Forms for Usability*, Chapter 5, 6.
  - Theresa Neil, 2014, *Mobile Design Pattern Gallery: UI Patterns for Smartphone Apps*, O'Reilly Media; 2 edition. Chapter 2.
  - Shneiderman, Plasiant, Cohen, Jacobs (2014) *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Pearson. Chapter 6
- Mobile Accessibility: <https://www.w3.org/TR/mobile-accessibility-mapping/>

# Lab: Material Design, Animation, Authentication

---

## Introduction

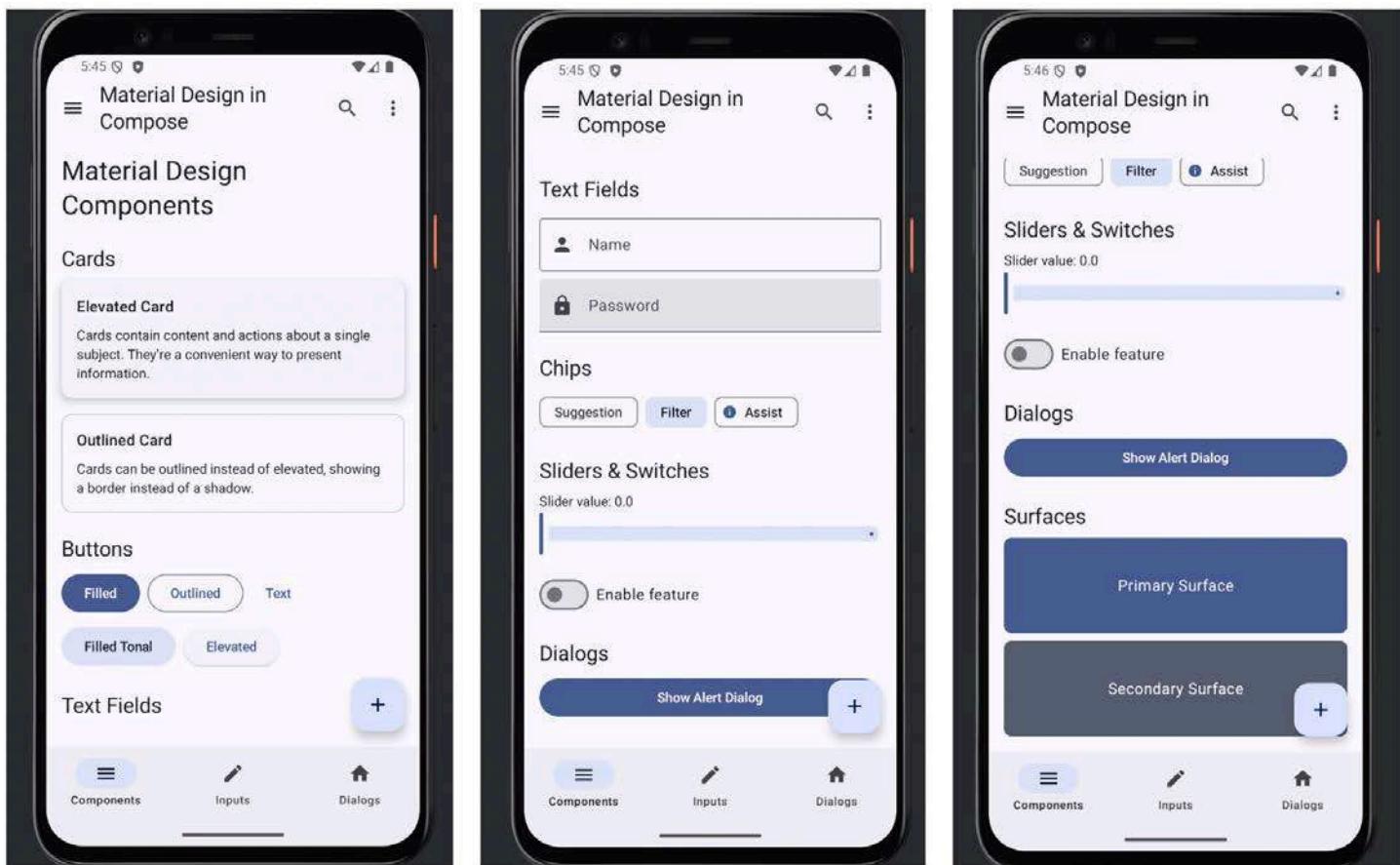
This week, we will focus on the following key objectives:

- **APP 1: Material Design:**
  - This app serves as a comprehensive demonstration of **Material Design 3 components** implemented using **Jetpack Compose**. Its primary goal is to help you understand how to structure and style modern Android UIs by exploring a wide range of built-in Material components such as **cards, buttons, text fields, chips, sliders, switches, dialogues, surfaces**, and more. The layout is structured using **Scaffold, NavigationDrawer, and BottomNavigationBar**, showcasing proper UI hierarchy and responsive design principles. Each section is interactive and state-driven, making this app an effective reference for applying Material theming, typography, and layout best practices in real-world Compose applications.
- **APP 2: Animation Showcase App:**
  - This app is an **interactive learning tool** designed to show how to implement various **animation techniques using Jetpack Compose**. It presents a curated collection of animation patterns commonly used in modern Android UI development, such as fade transitions, spring-based motion, infinite effects, and layout changes. Each animation is organised into its own self-contained composable function, allowing you to explore them one by one inside a vertically scrollable interface. By interacting with the app, you gain hands-on experience with both **simple and complex animation patterns**, helping you apply these concepts confidently in your own apps.
- **APP 3: Authentication with Google:**
  - This app demonstrates how to implement **user authentication using Firebase**, with a focus on **Google Sign-In** and **email/password login**. The app is structured with two main components: `LoginActivity` and `MainActivity`.
    - **LoginActivity:** handles the entire authentication flow. It presents login options (email/password and Google Sign-In using the Credential Manager API), validates user input, and manages authentication state. Upon successful login, it securely communicates with Firebase Authentication to verify user credentials.
    - **MainActivity:** displays the authenticated user's basic profile information (name and email), which is retrieved from Firebase after login. This screen represents the post-login state and confirms that authentication was successful.
  - Overall, this app serves as a practical introduction to **secure authentication in modern Android apps**, helping you understand how to integrate and manage Firebase login workflows using Jetpack Compose.

# APP 1: Material Design

## Objective

This app serves as a **comprehensive showcase of Material Design 3 components** using **Jetpack Compose**. It helps you understand how to implement and style modern UI elements following Material Design principles.



## App Features

This app includes the following Material 3 UI elements:

- **Navigation Components:**
  - Navigation Drawer
  - Top App Bar
  - Bottom Navigation Bar
  - Floating Action Button
- **UI Components:**
  - Cards: Elevated, Outlined
  - Buttons: Filled, Outlined, Tonal, Elevated
  - Text Fields: Outlined, Password

- Chips: Suggestion, Filter, Assist
- Sliders, Switches
- Dialog Boxes
- **Surface Examples:**
  - Surface containers styled with primary and secondary colour schemes

## Styling and Layout

- All components are styled using **Material Design 3 principles**
- Proper theming and typography are applied
- Layout uses `LazyColumn` for vertical scrolling and spacing

## CODE Structure

`MainActivity`

- Entry point: `onCreate()`
  - Calls `enableEdgeToEdge()` to allow full-screen layout
  - Sets the Jetpack Compose UI using `setContent { ... }`
  - Applies the app theme
  - Invokes the main UI composable: `MaterialDesignShowcase()`

`MaterialDesignShowcase()`



`MaterialDesignShowcase()`

```

|   State variables using remember:
|   |   drawerState → Navigation drawer open/close
|   |   scope → CoroutineScope for drawer control
|   |   selectedTab → Currently selected tab in bottom bar
|   |   showDialog → Controls visibility of AlertDialog
|   |   textFieldValue, passwordValue → For name and password fields
|   |   sliderValue → For slider component
|   |   switchChecked → For toggle switch

|   Conditional Dialog:
|   |   if(showDialog) → shows AlertDialog with confirm/dismiss buttons

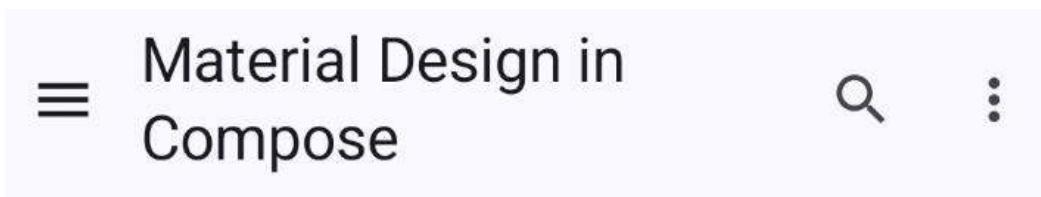
|   ModalNavigationDrawer:
|   |   drawerContent → List of NavigationDrawerItem components
|   |   content → Scaffold {
|   |   |   topBar → TopAppBar (with title, menu, search, more)
|   |   |   bottomBar → NavigationBar (with 3 dynamic tabs)
|   |   |   floatingActionButton → Standard FAB with add icon
|   |   |   content → LazyColumn {
```

```
|   |- Section: App title  
|   |- Section: Cards (ElevatedCard, OutlinedCard)  
|   |- Section: Buttons (Filled, Outlined, Text, Tonal, Elevated)  
|   |- Section: TextFields (Outlined, Password)  
|   |- Section: Chips (Suggestion, Filter, Assist)  
|   |- Section: Sliders & Switches (with labels)  
|   |- Section: Dialog trigger button  
|- Section: Surfaces (Primary and Secondary themed containers)
```

## Top App Bar

This section demonstrates how to build a Material Design Top App Bar in Jetpack Compose using `TopAppBar()`. The `TopAppBar` provides users with access to navigation and actions related to the current screen. In this example, it includes:

- A **menu icon** to open the navigation drawer
- A **search icon** and **more options icon** on the right

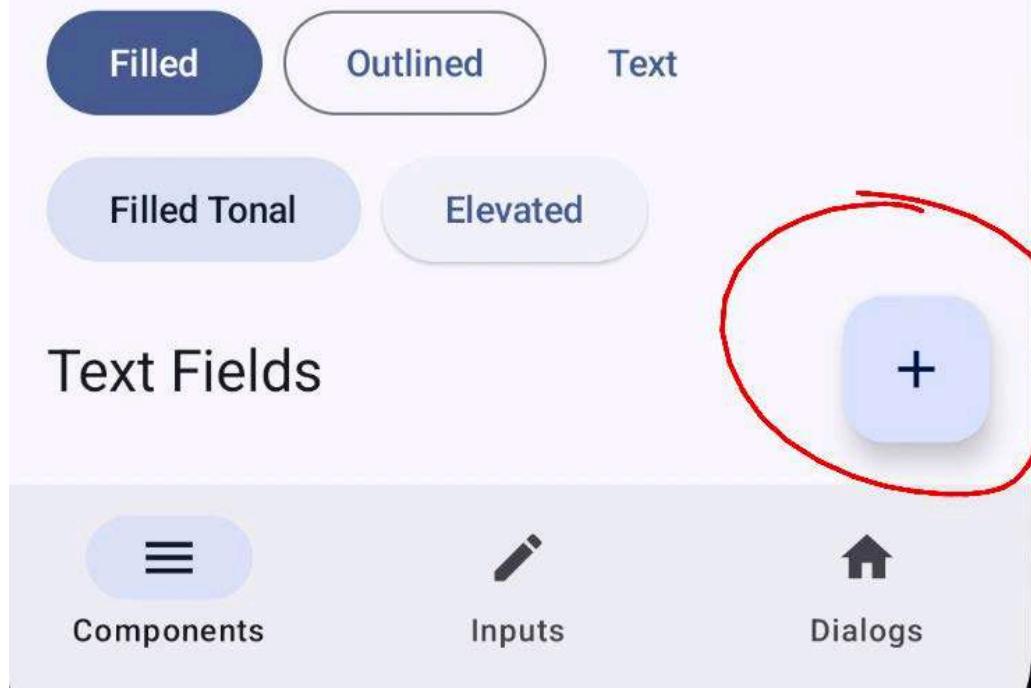


```
topBar = {
    TopAppBar(
        title = { Text("Material Design in Compose") },
        // Menu button that opens the navigation drawer
        navigationIcon = {
            IconButton(onClick = { scope.launch { drawerState.open() } })
            Icon(
                imageVector = Icons.Default.Menu,
                contentDescription = "Menu"
            )
        },
        // Action buttons displayed on the right side of the top bar
        actions = {
            // Search action button
            IconButton(onClick = { /* Search action */ })
            Icon(Icons.Default.Search, contentDescription = "Search")
        }
        // More options button
        IconButton(onClick = { /* More action */ })
        Icon(Icons.Default.MoreVert, contentDescription = "More")
    )
},
```

codesnap.dev

## Floating Action Button

## Buttons



```
// Floating action button displayed on top of the content
floatingActionButton = {
    FloatingActionButton(
        onClick = { /* fab click */ },
    ) {
        Icon(Icons.Default.Add, contentDescription = "Add")
    }
},
```

## Bottom navigation bar with tabs

This section demonstrates how to implement a **Material 3 Bottom Navigation Bar** using `NavigationBar()` in Jetpack Compose. The `Bottom Navigation Bar` allows users to switch between primary destinations in the app with a single tap.



Components



Inputs



Dialogs

### ***Key Features from the Code***

- Dynamically creates items using `tabs.forEachIndexed`
- Each item contains:
  - An **icon** (based on the tab index)
  - A **label** (tab title)
- The currently selected tab is **highlighted**
- Tapping a tab updates the selected state

```
// Define the available tabs for bottom navigation
val tabs = listOf("Components", "Inputs", "Dialogs")
```

```
bottomBar = {
    NavigationBar {
        // Create a navigation item for each tab
        tabs.forEachIndexed { index, title ->
            NavigationBarItem(
                // Icon for the navigation item (different for each tab)
                icon = {
                    Icon(
                        when (index) {
                            0 -> Icons.Default.Menu
                            1 -> Icons.Default.Edit
                            else -> Icons.Default.Home
                        },
                        contentDescription = title
                    )
                },
                // Text label shown below the icon
                label = { Text(title) },
                // Highlight the currently selected tab
                selected = selectedTab == index,
                // Update selected tab when clicked
                onClick = { selectedTab = index }
            )
        }
    }
}
```

codesnap.dev

## Elevated Card with shadow effect

This section showcases how to create an **Elevated Card** in Jetpack Compose using the `ElevatedCard()` composable. Elevated cards are used to group related information with a subtle shadow to distinguish them from the background.

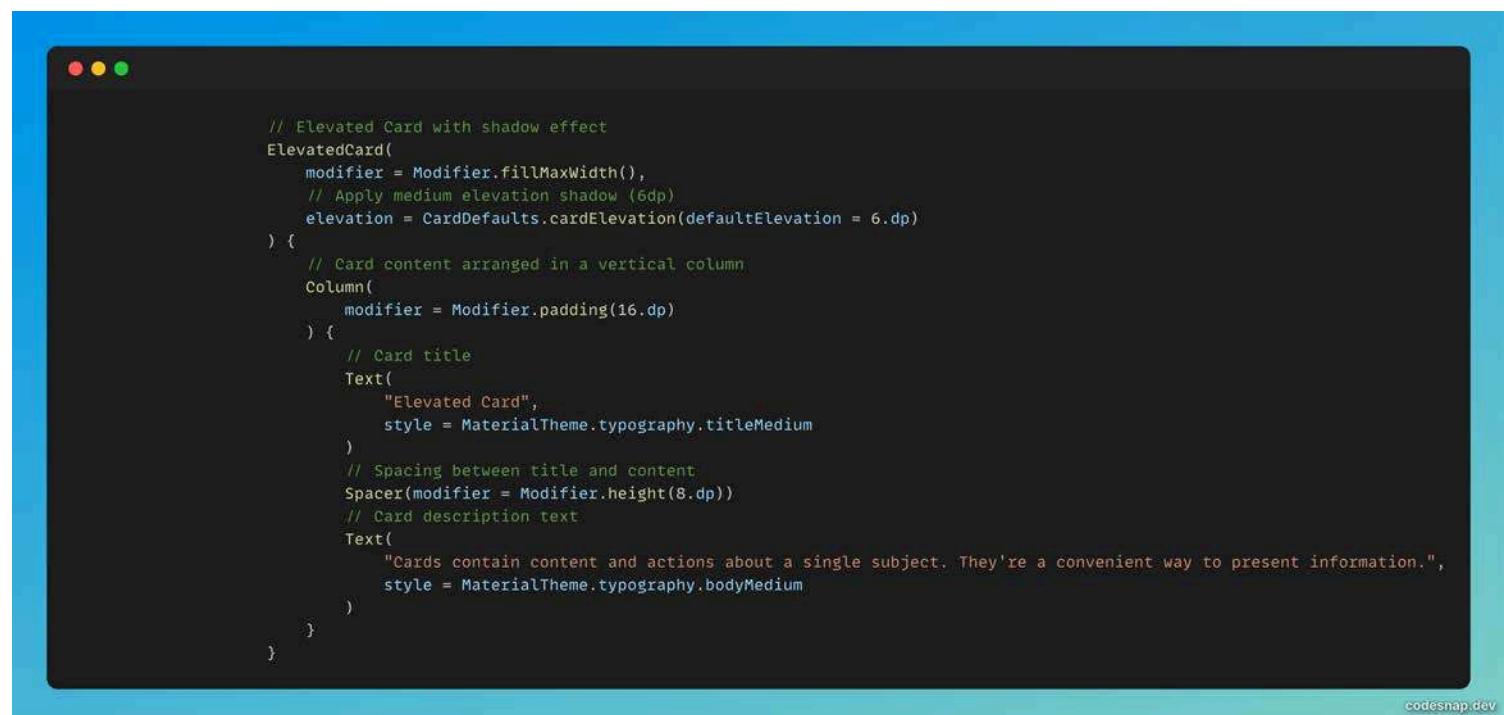
### Cards

#### Elevated Card

Cards contain content and actions about a single subject. They're a convenient way to present information.

## Key Features from the Code

- Uses `ElevatedCard()` with **medium elevation** (6.dp)
- The card content is structured in a **vertical Column**
- Includes:
  - A **title** styled with `titleMedium`
  - A **description** styled with `bodyMedium`
  - Proper **spacing** between title and content



## Outlined Card with border instead of elevation

This section shows how to use an **Outlined Card** in Jetpack Compose. Unlike `ElevatedCard`, it uses a **border** to separate the content instead of elevation. Outlined cards are used to emphasize content boundaries with a subtle border, without using shadows.

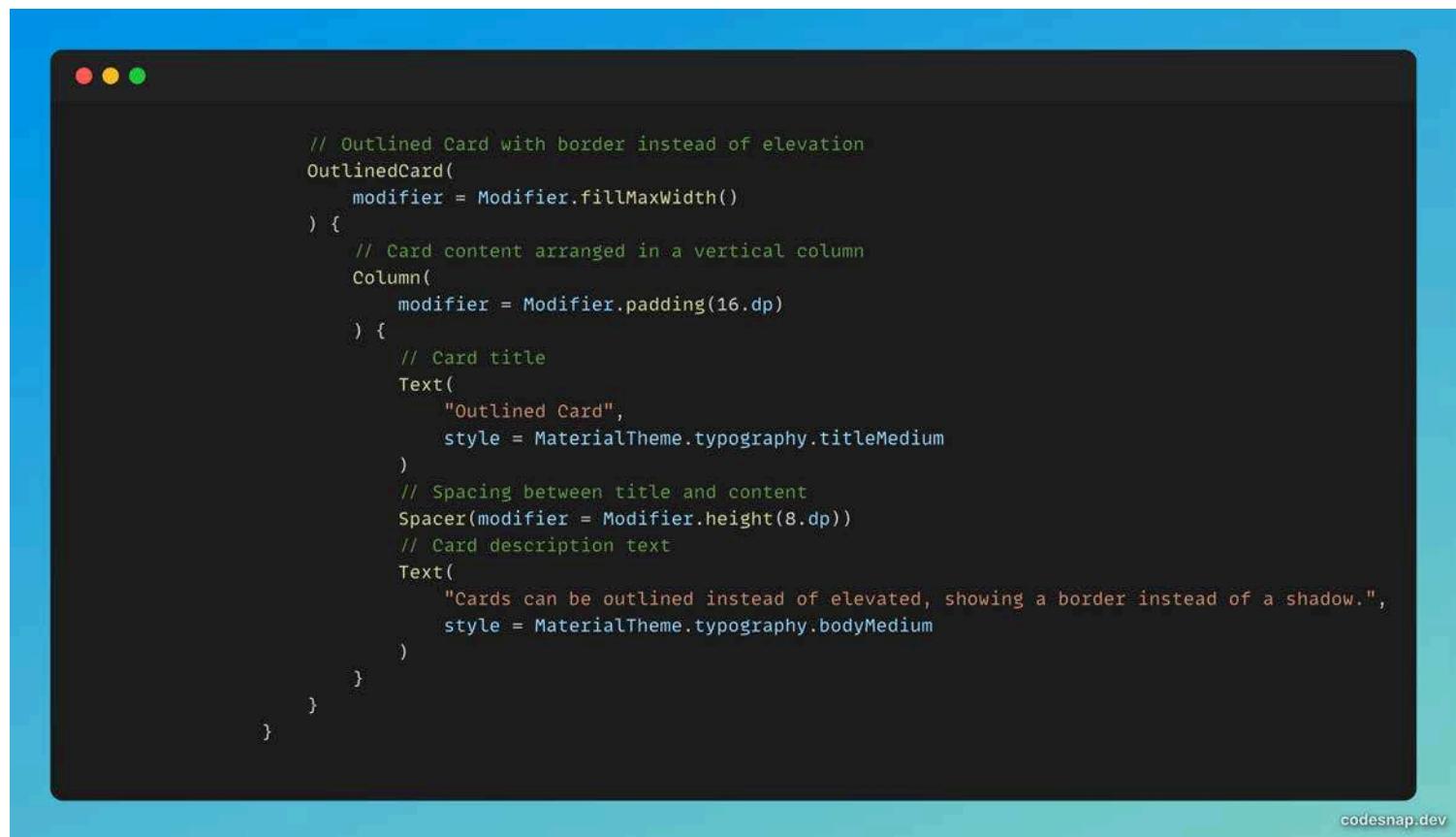
### Outlined Card

Cards can be outlined instead of elevated, showing a border instead of a shadow.

## Key Features from the Code

- Uses `OutlinedCard()` to define the card layout
- Card content is arranged in a vertical `Column`
- Includes:

- A **title** styled with `titleMedium`
- A **description** styled with `bodyMedium`
- Vertical **spacing** between elements using `Spacer`



```
// Outlined Card with border instead of elevation
OutlinedCard(
    modifier = Modifier.fillMaxWidth()
) {
    // Card content arranged in a vertical column
    Column(
        modifier = Modifier.padding(16.dp)
    ) {
        // Card title
        Text(
            "Outlined Card",
            style = MaterialTheme.typography.titleMedium
        )
        // Spacing between title and content
        Spacer(modifier = Modifier.height(8.dp))
        // Card description text
        Text(
            "Cards can be outlined instead of elevated, showing a border instead of a shadow.",
            style = MaterialTheme.typography.bodyMedium
        )
    }
}
```

codesnap.dev

## Material Buttons

This section demonstrates how to use different types of **Material Design buttons** in Jetpack Compose. Each button conveys a different level of emphasis based on its style. Buttons allow users to take actions. Material 3 provides multiple button variants to communicate priority, emphasis, and style.

- Use `Row` and `Arrangement.spacedBy()` to layout buttons horizontally with spacing
- Choose button types based on UI hierarchy:
  - Use `Button` for primary actions
  - Use `OutlinedButton` for secondary actions
  - Use `TextButton` for optional or less emphasized actions

## Buttons

Filled

Outlined

Text

Filled Tonal

Elevated

```
// First row of buttons with different styles
Row(
    modifier = Modifier.fillMaxWidth(),
    // Add space between buttons
    horizontalArrangement = Arrangement.spacedBy(8.dp)
) {
    // Standard filled button with high emphasis
    Button(onClick = { /* Filled button click */ }) {
        Text("Filled")
    }

    // Outlined button with medium emphasis
    OutlinedButton(onClick = { /* Outlined button click */ }) {
        Text("Outlined")
    }

    // Text-only button with low emphasis
    TextButton(onClick = { /* Text button click */ }) {
        Text("Text")
    }
}

// Space between button rows
Spacer(modifier = Modifier.height(8.dp))

// Second row of buttons with additional styles
Row(
    modifier = Modifier.fillMaxWidth(),
    // Add space between buttons
    horizontalArrangement = Arrangement.spacedBy(8.dp)
) {
    // Filled tonal button (softer filled style)
    FilledTonalButton(onClick = { /* Filled tonal button click */ }) {
        Text("Filled Tonal")
    }

    // Elevated button with shadow effect
    ElevatedButton(onClick = { /* Elevated button click */ }) {
        Text("Elevated")
    }
}
```

codesnap.dev

## Text Fields

This section demonstrates how to implement **text input fields** using `OutlinedTextField` and `TextField` in Jetpack Compose. Text fields allow users to input and edit text. Compose supports

multiple styles and keyboard behaviors, as well as visual transformations for passwords.

- Use `OutlinedTextField` for fields requiring visual separation
- Use `PasswordVisualTransformation()` to hide password text
- Set `keyboardOptions` to `KeyboardType.Password` for secure input keyboards

## Text Fields



```
item {
    // Section header for text input fields
    Text(
        "Text Fields",
        style = MaterialTheme.typography.titleLarge,
        modifier = Modifier.padding(vertical = 8.dp)
    )

    // Outlined text field for name input
    OutlinedTextField(
        // Bind to the text state variable
        value = textFieldValue,
        // Update state when text changes
        onValueChange = { textFieldValue = it },
        // Floating label text
        label = { Text("Name") },
        // Icon at the beginning of the text field
        leadingIcon = { Icon(Icons.Default.Person, contentDescription = "Person Icon") },
        // Make the field full width
        modifier = Modifier.fillMaxWidth()
    )

    // Space between text fields
    Spacer(modifier = Modifier.height(8.dp))

    // Filled text field for password input
    TextField(
        // Bind to the password state variable
        value = passwordValue,
        // Update state when text changes
        onValueChange = { passwordValue = it },
        // Floating label text
        label = { Text("Password") },
        // Hide the actual password characters
        visualTransformation = PasswordVisualTransformation(),
        // Show password keyboard with secure input
        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Password),
        // Icon at the beginning of the text field
        leadingIcon = { Icon(Icons.Default.Lock, contentDescription = "Lock Icon") },
        // Make the field full width
        modifier = Modifier.fillMaxWidth()
    )
}
```

codesnap.dev

## Chips

This section demonstrates how to use three types of **Material 3 Chip components** in Jetpack Compose: `SuggestionChip`, `FilterChip`, and `AssistChip`. Chips are compact elements that allow users to enter information, make selections, or trigger actions.

```
// == CHIPS SECTION ==
item {
    // Section header for chips
    Text(
        "Chips",
        style = MaterialTheme.typography.titleLarge,
        modifier = Modifier.padding(vertical = 8.dp)
    )

    // Row container for displaying chips horizontally
    Row(
        modifier = Modifier.fillMaxWidth(),
        // Add space between chips
        horizontalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        // Suggestion chip - for displaying suggestions to the user
        SuggestionChip(
            onClick = { /* Chip click */ },
            label = { Text("Suggestion") }
        )

        // Filter chip - for filtering content, can be selected/unselected
        FilterChip(
            // This chip is in selected state
            selected = true,
            onClick = { /* Filter chip click */ },
            label = { Text("Filter") }
        )

        // Assist chip - for initiating actions with an icon
        AssistChip(
            onClick = { /* Assist chip click */ },
            label = { Text("Assist") },
            // Icon to display next to text
            leadingIcon = {
                Icon(
                    Icons.Filled.Info,
                    contentDescription = "Info",
                    // Set icon size to be smaller
                    Modifier.size(16.dp)
                )
            }
        )
    }
}
```

# Chips

Suggestion

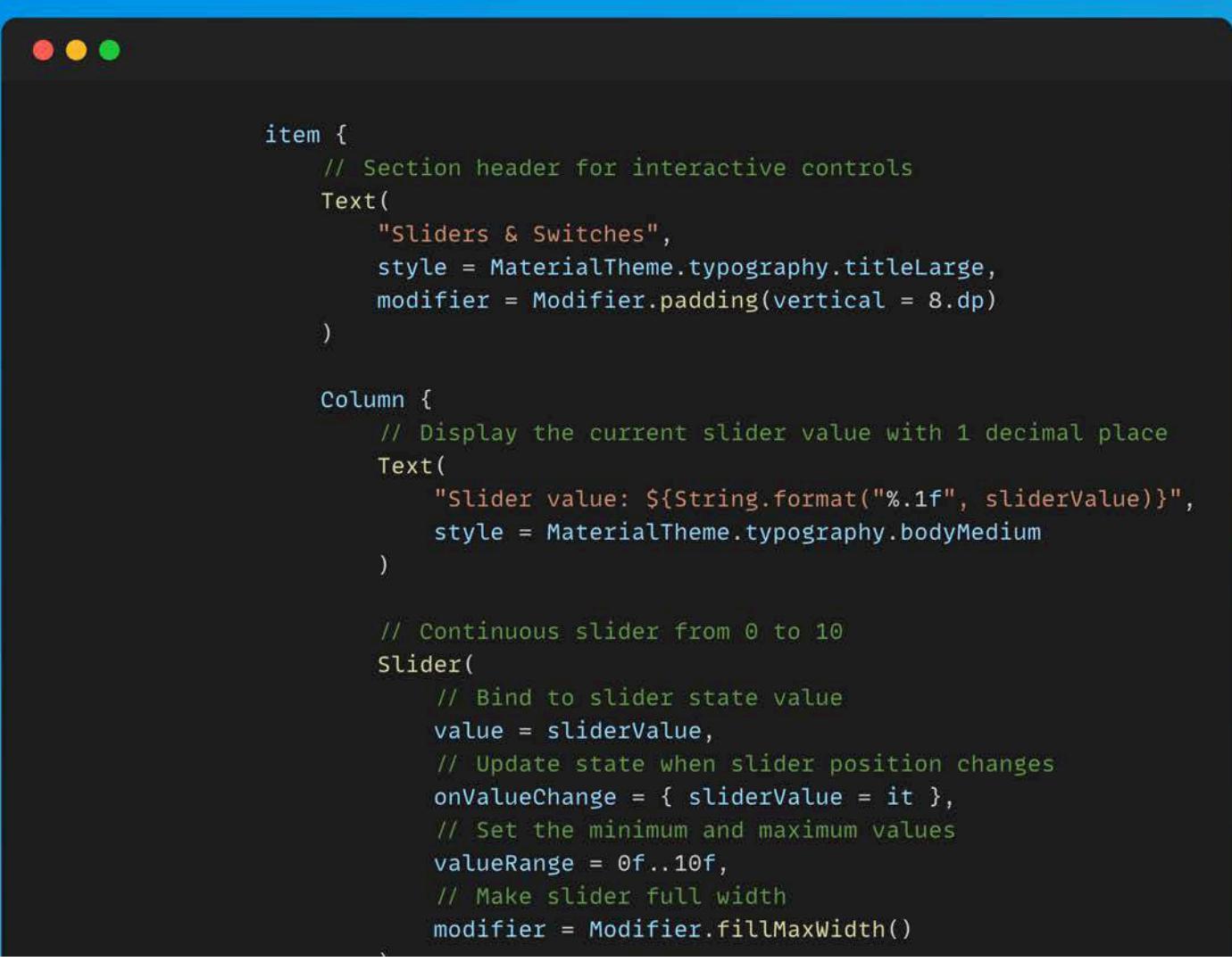
Filter

i Assist

## Sliders and Switches

This section demonstrates how to implement **interactive controls** including a `Slider` (for continuous values) and a `Switch` (for binary toggle states) using Jetpack Compose.

- **Slider:** Used for selecting a numeric value from a defined continuous range
- **Switch:** Used for toggling a setting between on/off states



```
item {  
    // Section header for interactive controls  
    Text(  
        "Sliders & Switches",  
        style = MaterialTheme.typography.titleLarge,  
        modifier = Modifier.padding(vertical = 8.dp)  
    )  
  
    Column {  
        // Display the current slider value with 1 decimal place  
        Text(  
            "Slider value: ${String.format("%.1f", sliderValue)}",  
            style = MaterialTheme.typography.bodyMedium  
        )  
  
        // Continuous slider from 0 to 10  
        Slider(  
            // Bind to slider state value  
            value = sliderValue,  
            // Update state when slider position changes  
            onValueChange = { sliderValue = it },  
            // Set the minimum and maximum values  
            valueRange = 0f..10f,  
            // Make slider full width  
            modifier = Modifier.fillMaxWidth()  
        )  
    }  
}
```

```
        )  
  
        // Space between slider and switch  
        Spacer(modifier = Modifier.height(16.dp))  
  
        // Row for switch and its label  
        Row(  
            verticalAlignment = Alignment.CenterVertically  
        ) {  
            // Toggle switch component  
            Switch(  
                // Bind to switch state  
                checked = switchChecked,  
                // Update state when switch is toggled  
                onCheckedChange = { switchChecked = it }  
            )  
  
            // Space between switch and label  
            Spacer(modifier = Modifier.width(8.dp))  
  
            // Label describing the switch function  
            Text("Enable feature")  
        }  
    }  
}
```

codesnap.dev

# Sliders & Switches

Slider value: 0.0



## Dialogs

This section demonstrates how to implement a **Material 3 AlertDialog** using Jetpack Compose, including title, content, and two action buttons. Dialogs are used to inform users about tasks or

request decisions. This dialog appears when `showDialog` is `true`.

- Wrap your `AlertDialog` in a conditional (`if (showDialog)`) to control visibility
- Use both confirm and dismiss buttons for flexibility
- Always provide an `onDismissRequest` to support back press and outside click behavior



```
// === DIALOG DEMONSTRATION SECTION ===
item {
    // Section header for dialogs
    Text(
        "Dialogs",
        style = MaterialTheme.typography.titleLarge,
        modifier = Modifier.padding(vertical = 8.dp)
    )

    // Button that triggers the dialog display
    Button(
        onClick = { showDialog = true },
        modifier = Modifier.fillMaxWidth()
    ) {
        Text("Show Alert Dialog")
    }
}
```

```
// Display an AlertDialog when showDialog is true
if (showDialog) {
    AlertDialog(
        // Close dialog when clicking outside
        onDismissRequest = { showDialog = false },
        // Dialog title component
        title = { Text("Material Dialog") },
        // Main content text of the dialog
        text = { Text("This is a Material 3 dialog example. " +
            "Dialogs inform users about tasks and can contain " +
            "critical information or require decisions.") },
        // Primary action button
        confirmButton = {
            TextButton(onClick = { showDialog = false }) {
                Text("Confirm")
            }
        },
        // Secondary action button
        dismissButton = {
            TextButton(onClick = { showDialog = false }) {
                Text("Dismiss")
            }
        }
    )
}
```

Slider value: 0.0

# Material Dialog

This is a Material 3 dialog example.

Dialogs inform users about tasks and can contain critical information or require decisions.

Dismiss

Confirm

## Surfaces

This section demonstrates how to use the `Surface` composable in Jetpack Compose with **Material 3 color theming**. Surfaces are used as containers for content, applying elevation, color, and shape. Surfaces provide a background layer with a theme-defined color. You can use them to wrap content and apply elevation, shape, and semantic meaning.

- Use `onPrimary` or `onSecondary` for contrast text color
- Are styled with rounded corners (`RoundedCornerShape(8.dp)`)
- Center the label text using `Box` and `contentAlignment`

```
// == SURFACES SECTION ==
item {
    // Section header for surface demonstrations
```

```
Text(
    "Surfaces",
    style = MaterialTheme.typography.titleLarge,
    modifier = Modifier.padding(vertical = 8.dp)
)

// Primary colored surface with text
Surface(
    modifier = Modifier
        .fillMaxWidth()
        .height(100.dp),
    // Use the theme's primary color
    color = MaterialTheme.colorScheme.primary,
    // Use the on-primary color for content (ensuring contrast)
    contentColor = MaterialTheme.colorScheme.onPrimary,
    // Apply rounded corners to the surface
    shape = RoundedCornerShape(8.dp)
) {
    // Center the text in the surface
    Box(
        modifier = Modifier.fillMaxSize(),
        contentAlignment = Alignment.Center
    ) {
        Text(
            "Primary Surface",
            style = MaterialTheme.typography.bodyLarge
        )
    }
}

// Space between surfaces
Spacer(modifier = Modifier.height(8.dp))

// Secondary colored surface with text
Surface(
    modifier = Modifier
        .fillMaxWidth()
        .height(100.dp),
    // Use the theme's secondary color
    color = MaterialTheme.colorScheme.secondary,
    // Use the on-secondary color for content (ensuring contrast)
    contentColor = MaterialTheme.colorScheme.onSecondary,
    // Apply rounded corners to the surface
    shape = RoundedCornerShape(8.dp)
) {
    // Center the text in the surface
    Box(
        modifier = Modifier.fillMaxSize(),
        contentAlignment = Alignment.Center
    ) {
        Text(
            "Secondary Surface",
            style = MaterialTheme.typography.bodyLarge
        )
    }
}
```

# Surfaces

Primary Surface

Secondary Surface



## APP 2: Animation Showcase App

This app is an interactive educational app designed to learn Jetpack Compose animation techniques in Android development. The Animation Showcase presents a comprehensive collection of modern UI animation patterns implemented using Kotlin and Jetpack Compose's declarative UI framework. You can explore, interact with, and study the code implementation of each animation type to understand how to create fluid, engaging user interfaces in their own Android applications.

### Features:

- **Fade In/Out Animation** - Demonstrates how to make UI elements appear and disappear with smooth transitions using AnimatedVisibility.
- **Color Change Animation** - Shows smooth color transitions between different states using animateColorAsState.
- **Size Animation with Spring Physics** - Illustrates how to animate an element's size with natural-feeling spring animations that include bounce effects.
- **Position Animation with Spring Physics** - Demonstrates movement animations with configurable spring parameters for controlling stiffness and dampening.
- **Content Transition Animation** - Showcases how to smoothly transition between different content screens with sliding animations.
- **Infinite Animation** - Presents continuous, looping animations that run indefinitely without user interaction, including rotation and pulsing effects.
- **Layout Change Animation** - Demonstrates how containers can smoothly resize as their content expands or collapses using animateContentSize.

Each animation example includes interactive elements that allow students to trigger the animations and observe their behaviour in real-time, providing an engaging hands-on learning experience for mobile UI animation concepts.

```
1  /**
2   * Main Activity class that serves as the entry point for the animation examples app
3   * This class demonstrates various animation techniques available in Jetpack Compose
4  */
5  class MainActivity : ComponentActivity() {
6      /**
7       * Called when the activity is first created
8       * Sets up the Compose UI through setContent
9       * @param savedInstanceState Bundle containing the activity's previously saved state
10      */
11     override fun onCreate(savedInstanceState: Bundle?) {
12         super.onCreate(savedInstanceState)
13         setContent {
14             // Apply the app theme to all composables
15             Week10_animationTheme {
16                 // Create a full-screen surface with the theme's background color
17                 Surface(
18                     modifier = Modifier.fillMaxSize(),
19                     color = MaterialTheme.colorScheme.background
20                 ) {
21                     // Display the main screen containing all animation demos
22                     AnimationDemoScreen()
23                 }
24             }
25         }
26     }
27 }
```

## AnimationDemoScreen

This is the **main composable function** that acts as the container for all individual animation demos. Each of these functions (like `FadeAnimationDemo()`) is assumed to be a **self-contained composable** that demonstrates one animation type, with interactive UI controls to trigger and observe the animation.

### How to Read & Understand This App

- Start at `AnimationDemoScreen()`, just like `MaterialDesignShowcase()` in APP 1
- Understand the layout:
  - Everything is placed in a vertically scrollable `Column`
  - There's consistent spacing and padding
- Explore each animation demo function one by one:
  - `FadeAnimationDemo()` : likely uses `AnimatedVisibility`
  - `SpringAnimationDemo()` : likely uses `Animatable` or `animateDpAsState` with `spring()`
  - ...

```
1
2 /**
3  * Main composable function that displays all animation examples in a scrollable column
4  * This screen serves as the container for all individual animation demos
5 */
6 @Composable
7 fun AnimationDemoScreen() {
8     // Create a scroll state to enable vertical scrolling through demos
9     val scrollState = rememberScrollState()
10
11    // Main column container for all animation demos
12    Column(
13        modifier = Modifier
14            .fillMaxSize() // Use the entire available screen space
15            .verticalScroll(scrollState) // Enable vertical scrolling for content
16            .padding(16.dp), // Add padding around all content
17        verticalArrangement = Arrangement.spacedBy(24.dp) // Add vertical space between each demo
18    ) {
19        // Title for the animation examples screen
20        Text(
21            text = "FIT2081-Animation Examples",
22            fontSize = 20.sp,
23            fontWeight = FontWeight.Bold,
24            modifier = Modifier.padding(bottom = 16.dp)
25        )
26
27        // Display all animation demo components
28        FadeAnimationDemo()
29
30        // Color Animation Example
31        ColorChangeAnimationDemo()
32
33        // Size Animation Example
34        SizeAnimationDemo()
35
36        // Spring Animation Example
37        SpringAnimationDemo()
38
39        // Content Animation Example
40        ContentAnimationDemo()
41
42        // Infinite Animation Example
43        InfiniteAnimationDemo()
44
45        // Layout Change Animation Example
46        LayoutChangeAnimationDemo()
47
48        // Add extra space at the bottom for better scrolling experience
49        Spacer(modifier = Modifier.height(32.dp))
50    }
51 }
```

## FadeAnimationDemo

The FadeAnimationDemo function is a Jetpack Compose @Composable function designed to showcase a fade-in and fade-out animation.

Its role is to provide a visual example of how UI elements can be made to appear and disappear smoothly using AnimatedVisibility.

Structurally, the function is organized as follows:

1. State Management: It initializes a visible boolean state variable using remember { mutableStateOf(true) }. This state determines whether the animated content should be visible or hidden.
2. Layout: It uses a Column to arrange its child elements vertically.  
Informational Text: A Text composable displays the title "Fade In/Out Animation".
3. Control Element: A Button is provided. When clicked (onClick), it toggles the visible state (visible = !visible). The button's text dynamically changes between "Hide" and "Show" based on the current visible state.
4. Animated Content: The AnimatedVisibility composable is the core of the animation.
  1. It takes the visible state as a parameter to control its content's visibility.
  2. The enter parameter is configured with fadeIn(animationSpec = tween(1000)), specifying that the content should fade in over 1000 milliseconds (1 second) when visible becomes true.
  3. The exit parameter is configured with fadeOut(animationSpec = tween(1000)), specifying that the content should fade out over 1000 milliseconds when visible becomes false.
  4. The content that will be animated (i.e., the UI element that fades in and out) is defined within the trailing lambda of AnimatedVisibility (though this part is not fully shown in your selection).

```
1  /**
2   * Demonstrates fade in/out animation using AnimatedVisibility
3   * This example shows how to make UI elements appear and disappear with smooth transitions
4   */
5  @Composable
6  fun FadeAnimationDemo() {
7      // State to track visibility of the animated element
8      var visible by remember { mutableStateOf(true) }
9
10     Column {
11         // Section title
12         Text("Fade In/Out Animation", fontWeight = FontWeight.Bold)
13
14         Spacer(modifier = Modifier.height(8.dp))
15
16         // Button to toggle visibility state
17         Button(onClick = { visible = !visible }) {
18             Text(if (visible) "Hide" else "Show")
19         }
20
21         Spacer(modifier = Modifier.height(8.dp))
22
23         // AnimatedVisibility controls the fade animation based on the visible state
24         AnimatedVisibility(
25             visible = visible,
26             enter = fadeIn(animationSpec = tween(1000)), // 1 second fade in animation
27             exit = fadeOut(animationSpec = tween(1000)) // 1 second fade out animation
28         ) {
29             // Blue box that will fade in/out
30             Box(
31                 modifier = Modifier
32                     .size(100.dp)
33                     .background(Color.Blue)
34             )
35         }
36     }
37 }
38 }
```

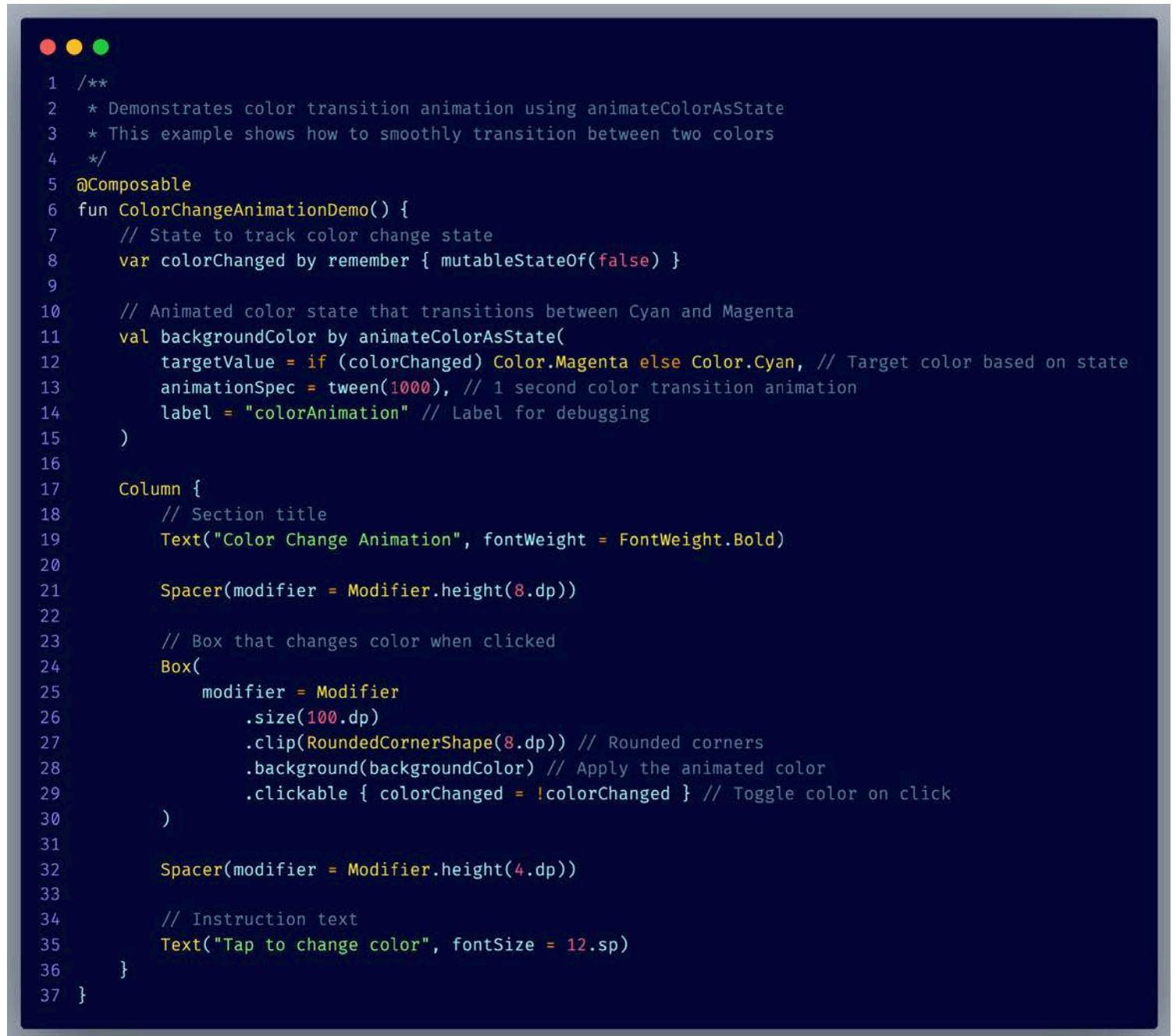
## ColorChangeAnimationDemo

The ColorChangeAnimationDemo function is a Composable function designed to showcase a smooth colour transition.

Structurally, it initialises a boolean state variable colorChanged using remember { mutableStateOf(false) }. This state determines the target colour for the animation.

The core of the animation is handled by animateColorAsState. This function takes a targetValue which dynamically switches between Color.Cyan and Colour.Magenta based on the colorChanged state.

The animationSpec = tween(1000) dictates that the colour transition will take 1000 milliseconds (1 second) to complete, creating a smooth visual effect. The label parameter is useful for debugging and identifying this specific animation in development tools. The resulting animated colour is stored in the backgroundColor\_ \_variable, which can then be applied to UI elements within the Column that follows.



```
1  /**
2   * Demonstrates color transition animation using animateColorAsState
3   * This example shows how to smoothly transition between two colors
4   */
5  @Composable
6  fun ColorChangeAnimationDemo() {
7      // State to track color change state
8      var colorChanged by remember { mutableStateOf(false) }
9
10     // Animated color state that transitions between Cyan and Magenta
11     val backgroundColor by animateColorAsState(
12         targetValue = if (colorChanged) Color.Magenta else Color.Cyan, // Target color based on state
13         animationSpec = tween(1000), // 1 second color transition animation
14         label = "colorAnimation" // Label for debugging
15     )
16
17     Column {
18         // Section title
19         Text("Color Change Animation", fontWeight = FontWeight.Bold)
20
21         Spacer(modifier = Modifier.height(8.dp))
22
23         // Box that changes color when clicked
24         Box(
25             modifier = Modifier
26                 .size(100.dp)
27                 .clip(RoundedCornerShape(8.dp)) // Rounded corners
28                 .background(backgroundColor) // Apply the animated color
29                 .clickable { colorChanged = !colorChanged } // Toggle color on click
30         )
31
32         Spacer(modifier = Modifier.height(4.dp))
33
34         // Instruction text
35         Text("Tap to change color", fontSize = 12.sp)
36     }
37 }
```

## ColorChangeAnimationDemo

The ColorChangeAnimationDemo function is a Jetpack Compose @Composable function designed to showcase a smooth colour transition.

Structurally, it initialises a boolean state variable colorChanged using remember and mutableStateOf to track whether the colour should be toggled. It then defines backgroundColor using animateColorAsState, which automatically animates the colour from Color.Cyan to Color.Magenta.

(or vice-versa) whenever colorChanged changes. This animation is configured to last 1000 milliseconds (1 second) using a tween. The UI itself is a Column containing a title Text, a Spacer for layout, and a Box. This Box has its background set to the animated backgroundColor and becomes clickable, toggling the colorChanged state upon interaction, thereby triggering the colour animation.

```
1  /**
2   * Demonstrates color transition animation using animateColorAsState
3   * This example shows how to smoothly transition between two colors
4   */
5  @Composable
6  fun ColorChangeAnimationDemo() {
7      // State to track color change state
8      var colorChanged by remember { mutableStateOf(false) }
9
10     // Animated color state that transitions between Cyan and Magenta
11     val backgroundColor by animateColorAsState(
12         targetValue = if (colorChanged) Color.Magenta else Color.Cyan, // Target color based on state
13         animationSpec = tween(1000), // 1 second color transition animation
14         label = "colorAnimation" // Label for debugging
15     )
16
17     Column {
18         // Section title
19         Text("Color Change Animation", fontWeight = FontWeight.Bold)
20
21         Spacer(modifier = Modifier.height(8.dp))
22
23         // Box that changes color when clicked
24         Box(
25             modifier = Modifier
26                 .size(100.dp)
27                 .clip(RoundedCornerShape(8.dp)) // Rounded corners
28                 .background(backgroundColor) // Apply the animated color
29                 .clickable { colorChanged = !colorChanged } // Toggle color on click
30         )
31
32         Spacer(modifier = Modifier.height(4.dp))
33
34         // Instruction text
35         Text("Tap to change color", fontSize = 12.sp)
36     }
37 }
```

## SizeAnimationDemo

The SizeAnimationDemo function is a Composable in Jetpack Compose designed to showcase a size animation with a spring-like, bouncy effect.

Structurally, it's a self-contained UI component:

- State Management: It uses remember { mutableStateOf(false) } to create an expanded boolean state. This state determines whether the animated element should be in its larger or smaller size.
- Animation Logic: The animateDpAsState function is the heart of the animation. It takes a targetValue which changes based on the expanded state (either 150.dp or 100.dp). The animationSpec is configured to use spring physics, with dampingRatio controlling the bounciness and stiffness influencing the speed of the animation. The result, size, is a State that smoothly transitions between the target values.
- UI Layout: A Column is used to arrange a title, Text, and the animated Box vertically. A Spacer adds some vertical separation. The Box is the visual element that animates. Its size modifier is directly bound to the animated size value. The Box is made interactive through a clickable modifier (not fully shown in the selection, but implied by the expanded state logic), which would toggle the expanded state, triggering the animation. It also has a clip modifier to give it rounded corners.



```
1 /**
2  * Demonstrates size animation using animateDpAsState with spring physics
3  * This example shows how to animate the size of a UI element with a bouncy effect
4 */
5 @Composable
6 fun SizeAnimationDemo() {
7     // State to track expanded/contracted status
8     var expanded by remember { mutableStateOf(false) }
9
10    // Animate size with spring physics for a bouncy effect
11    val size by animateDpAsState(
12        targetValue = if (expanded) 150.dp else 100.dp, // Toggle between two sizes
13        animationSpec = spring(
14            dampingRatio = Spring.DampingRatioMediumBouncy, // Controls bounciness
15            stiffness = Spring.StiffnessLow // Controls animation speed
16        ),
17        label = "sizeAnimation" // Label for debugging
18    )
19
20    Column {
21        // Section title
22        Text("Size Animation with Spring", fontWeight = FontWeight.Bold)
23
24        Spacer(modifier = Modifier.height(8.dp))
25
26        // Box that changes size with animation when clicked
27        Box(
28            modifier = Modifier
29                .size(size) // Apply the animated size
30                .clip(RoundedCornerShape(8.dp))
31                .background(Color.Green)
32                .clickable { expanded = !expanded } // Toggle size on click
33        )
34
35        Spacer(modifier = Modifier.height(4.dp))
36
37        // Instruction text
38        Text("Tap to resize", fontSize = 12.sp)
39    }
40 }
```

## SpringAnimationDemo

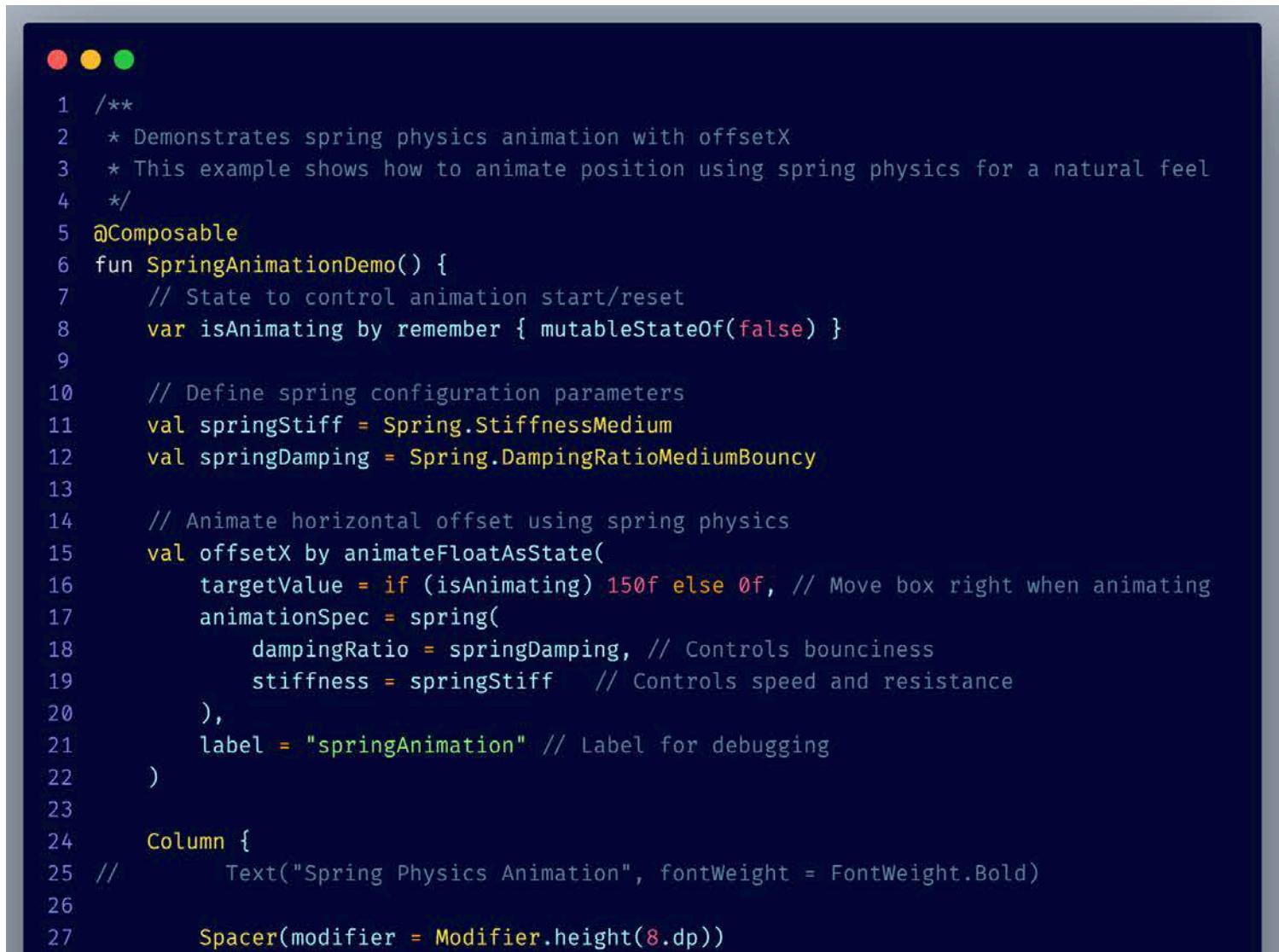
The SpringAnimationDemo function is a Composable in Jetpack Compose designed to showcase a spring physics-based animation.

Its primary role is to visually demonstrate how an element's horizontal position (offsetX) can be

animated with a natural, bouncy feel, characteristic of a spring.

Structurally, the function:

1. Declares a mutable state variable `isAnimating` using `remember { mutableStateOf(false) }`. This state acts as a trigger: when `isAnimating` changes, the animation's target value updates.
2. Defines constants `springStiff` and `springDamping` for the spring's physical properties (stiffness and damping ratio), which dictate the animation's speed and bounciness.
3. Uses `animateFloatAsState` to create an animatable float value `offsetX`.
  1. The `targetValue` for `offsetX` switches between `150f` (when `isAnimating` is true, moving the element) and `0f` (when `isAnimating` is false, returning it to the original position).
  2. The `animationSpec` is configured using `spring()`, applying the defined `dampingRatio` and stiffness. This tells Compose to use spring physics for the transition between target values.
  3. A label is provided for debugging purposes.
4. It then sets up a `Column` layout to arrange UI elements vertically. The `Spacer` adds some vertical spacing. Although not shown in this snippet, the `offsetX` value would typically be used with a `Modifier.offset(x = offsetX.dp)` on a visual element (like a `Box` or `Text`) inside this `Column` to actually move it horizontally.



```
1  /**
2   * Demonstrates spring physics animation with offsetX
3   * This example shows how to animate position using spring physics for a natural feel
4   */
5  @Composable
6  fun SpringAnimationDemo() {
7      // State to control animation start/reset
8      var isAnimating by remember { mutableStateOf(false) }
9
10     // Define spring configuration parameters
11     val springStiff = Spring.StiffnessMedium
12     val springDamping = Spring.DampingRatioMediumBouncy
13
14     // Animate horizontal offset using spring physics
15     val offsetX by animateFloatAsState(
16         targetValue = if (isAnimating) 150f else 0f, // Move box right when animating
17         animationSpec = spring(
18             dampingRatio = springDamping, // Controls bounciness
19             stiffness = springStiff // Controls speed and resistance
20         ),
21         label = "springAnimation" // Label for debugging
22     )
23
24     Column {
25         // Text("Spring Physics Animation", fontWeight = FontWeight.Bold)
26
27         Spacer(modifier = Modifier.height(8.dp))
```

```

28
29      // Button to trigger animation
30      Button(onClick = { isAnimating = !isAnimating }) {
31          Text(if (isAnimating) "Reset" else "Animate")
32      }
33
34      Spacer(modifier = Modifier.height(16.dp))
35
36      // Container for the animated box
37      Box(modifier = Modifier
38          .fillMaxWidth()
39          .height(120.dp)
40      ) {
41          // Box that will move horizontally with spring animation
42          Box(
43              modifier = Modifier
44              .offset(x = offsetX.dp) // Apply the animated offset
45              .size(70.dp)
46              .clip(RoundedCornerShape(8.dp))
47              .background(Color(0xFFEC407A))
48          ) {
49              // Text inside the animated box
50              Text(
51                  "Spring",
52                  color = Color.White,
53                  modifier = Modifier
54                      .align(Alignment.Center)
55                      .padding(8.dp)
56              )
57          }
58      }
59
60      // Description of the spring physics parameters
61      Text(
62          "Spring physics: DampingRatio=${springDamping}, Stiffness=${springStiff}",
63          fontSize = 12.sp,
64          color = Color.Gray
65      )
66  }
67 }
```

## ContentAnimationDemo

The ContentAnimationDemo function is a Jetpack Compose @Composable designed to showcase how animations can be applied when content changes, specifically when switching between different "pages" or views within an app.

Structurally, it's set up as follows:

1. State Management: It initialises and remembers a currentPage state variable. This variable will

determine which content (Page 1 or Page 2) is currently active.

2. Layout: It uses a Column to arrange its child elements vertically.
3. Title: A Text composable displays the title "Content Change Animation".
4. Navigation Controls: A Row contains two Buttons ("Page 1" and "Page 2"). Clicking these buttons updates the currentPage state.
5. Spacing: Spacer composables are used to add vertical spacing between UI elements for better visual separation.
6. Animated Content (Implied): The comment // AnimatedContent handles smooth transitions between different content indicates that the subsequent code (not shown in the selection) will use the AnimatedContent composable. This composable will observe changes to the currentPage state and animate the transition between the content associated with the old page and the new page.



```
1  /**
2   * Demonstrates content transition animation when switching between pages
3   * Uses ExperimentalAnimationApi for AnimatedContent
4  */
5 @OptIn(ExperimentalAnimationApi::class)
6 @Composable
7 fun ContentAnimationDemo() {
8     // State to track current page
9     var currentPage by remember { mutableStateOf(0) }
10
11    Column {
12        // Section title
13        Text("Content Change Animation", fontWeight = FontWeight.Bold)
14
15        Spacer(modifier = Modifier.height(8.dp))
16
17        // Navigation buttons for page switching
18        Row(
19            modifier = Modifier.fillMaxWidth(),
20            horizontalArrangement = Arrangement.SpaceBetween
21        ) {
22            // Page 1 button
23            Button(onClick = { currentPage = 0 }) {
24                Text("Page 1")
25            }
26
27            // Page 2 button
28            Button(onClick = { currentPage = 1 }) {
29                Text("Page 2")
30            }
31        }
32
33        Spacer(modifier = Modifier.height(8.dp))
34
35        // AnimatedContent handles smooth transitions between different content
36        AnimatedContent(
```

```

37     targetState = currentPage,
38     transitionSpec = {
39         // Slide in from right or left depending on navigation direction
40         slideInHorizontally(animationSpec = tween(300)) { fullWidth →
41             fullWidth * (if (targetState > initialState) 1 else -1)
42         } with
43         // Slide out in the opposite direction
44         slideOutHorizontally(animationSpec = tween(300)) { fullWidth →
45             -fullWidth * (if (targetState > initialState) 1 else -1)
46         }
47     },
48     label = "contentAnimation" // Label for debugging
49 ) { page →
50     // Display different content based on current page
51     when (page) {
52         // Page 1 content - red card
53         0 → Card(
54             modifier = Modifier.size(200.dp),
55             colors = CardDefaults.cardColors(containerColor = Color(0xFFE57373))
56         ) {
57             Box(
58                 contentAlignment = Alignment.Center,
59                 modifier = Modifier.fillMaxSize()
60             ) {
61                 Text("Page 1 Content")
62             }
63         }
64         // Page 2 content - green card
65         1 → Card(
66             modifier = Modifier.size(200.dp),
67             colors = CardDefaults.cardColors(containerColor = Color(0xFF81C784))
68         ) {
69             Box(
70                 contentAlignment = Alignment.Center,
71                 modifier = Modifier.fillMaxSize()
72             ) {
73                 Text("Page 2 Content")
74             }
75         }
76     }
77 }
78 }
79 }
```

## InfiniteAnimationDemo

The InfiniteAnimationDemo function is a Jetpack Compose @Composable function designed to showcase how to create animations that run continuously and indefinitely.

Its primary role is to demonstrate the use of InfiniteTransition for animations that don't stop, such as loading spinners or attention-grabbing visual effects that loop forever without user interaction.

Structurally, the function is set up as follows:

1. It initialises an infiniteTransition using rememberInfiniteTransition(). This infiniteTransition object acts as a manager for one or more child animations that will run indefinitely. The label parameter is useful for debugging and animation inspection tools.
  2. It then defines a specific animation using an extension function on infiniteTransition. In this snippet, animateFloat is used to create a floating-point value that animates continuously.
    - o initialValue: The starting value of the animation (0f for 0 degrees).
    - o targetValue: The ending value of the animation (360f for a full circle).
    - o animationSpec: This defines how the animation progresses.
      - infiniteRepeatable: Specifies that the animation should repeat forever.
      - animation = tween(2000, easing = LinearEasing): Defines the core animation behavior – a "tween" (short for in-between) animation that lasts 2000 milliseconds (2 seconds) and progresses at a constant rate ( LinearEasing).
      - repeatMode = RepeatMode.Restart: Dictates that when the animation reaches its targetValue, it should jump back to the initialValue and start over.
    - o label: Again, a descriptive label for this specific animation, helpful for debugging.
- The animated value (in this case, rotation) is then typically used to modify a Composable's properties, like its rotation, color, or size, to create the visual effect. The rest of this function (not shown in the snippet) would use this rotation value.

```
1  /**
2   * Demonstrates continuous animations that run indefinitely using InfiniteTransition
3   * Shows how to create animations that continue running without user interaction
4   */
5  @Composable
6  fun InfiniteAnimationDemo() {
7      // Create an infinite transition object that manages ongoing animations
8      val infiniteTransition = rememberInfiniteTransition(label = "infiniteTransition")
9
10     // Continuous rotation animation (0 to 360 degrees)
11     val rotation by infiniteTransition.animateFloat(
12         initialValue = 0f,
13         targetValue = 360f,
14         animationSpec = infiniteRepeatable(
15             animation = tween(2000, easing = LinearEasing), // 2 second linear rotation
16             repeatMode = RepeatMode.Restart // Start from beginning after each cycle
17         ),
18         label = "rotation" // Label for debugging
19     )
20
21     // Continuous pulsing animation (scaling between 0.8 and 1.2)
22     val scale by infiniteTransition.animateFloat(
23         initialValue = 0.8f,
24         targetValue = 1.2f,
25         animationSpec = infiniteRepeatable(
26             animation = tween(1000), // 1 second transition
27             repeatMode = RepeatMode.Reverse // Reverse direction after each cycle
28         ),
29     )
```

```

29         label = "scale" // Label for debugging
30     )
31
32     Column {
33         // Section title
34         Text("Infinite Animation", fontWeight = FontWeight.Bold)
35
36         Spacer(modifier = Modifier.height(8.dp))
37
38         // Container for centered animation
39         Box(
40             contentAlignment = Alignment.Center,
41             modifier = Modifier.fillMaxWidth()
42         ) {
43             // Box that rotates and pulses indefinitely
44             Box(
45                 modifier = Modifier
46                     .size(100.dp)
47                     .graphicsLayer {
48                         rotationZ = rotation // Apply rotation animation
49                         scaleX = scale      // Apply horizontal scale animation
50                         scaleY = scale      // Apply vertical scale animation
51                     }
52                     .clip(RoundedCornerShape(8.dp))
53                     .background(Color(0xFFFFA000)) // Orange/amber color
54             )
55         }
56     }
57 }
```

## LayoutChangeAnimationDemo

The LayoutChangeAnimationDemo function is a Composable function in Jetpack Compose designed to showcase how layout size changes can be animated smoothly.

Its primary role is to demonstrate the animateContentSize modifier. This modifier, when applied to a Composable, automatically animates any changes to its size that occur due to its content expanding or collapsing.

Structurally, the function is organised as follows:

1. State Management: It initialises a boolean state variable expanded using remember { mutableStateOf(false) }. This state determines whether the content within the card is visible (expanded) or hidden (collapsed).
2. Outer Layout: A Column acts as the main container for the demo's UI elements.

Title and Spacer: A Text composable displays the title "Layout Change Animation", and a Spacer adds some vertical spacing.

### 3. Animated Card:

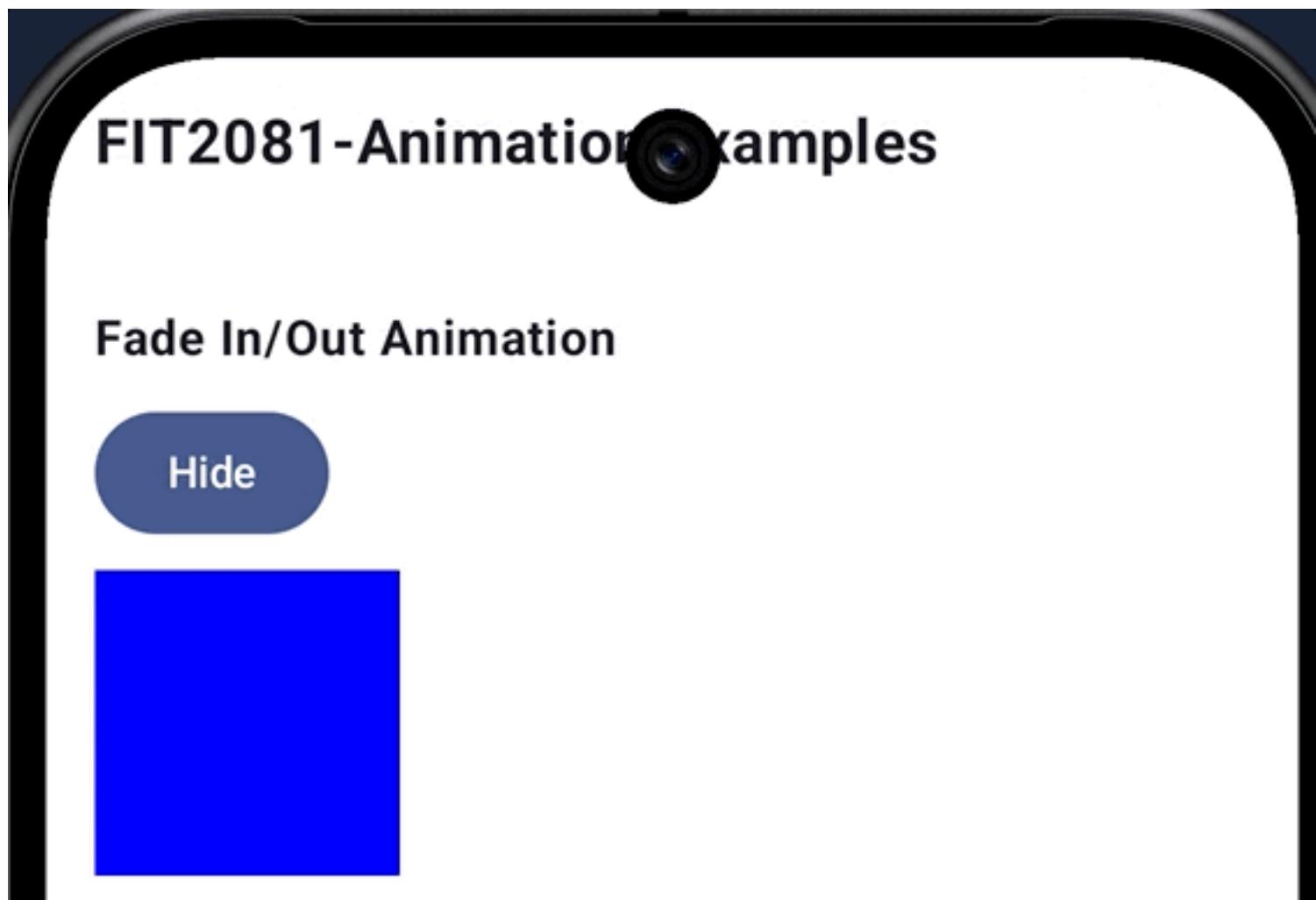
- A Card composable is the core element that demonstrates the animation.
- The animateContentSize modifier is applied directly to this Card. It's configured with a spring animation, where dampingRatio controls the bounciness and stiffness controls the speed of the animation.
- The clickable modifier on the Card toggles the expanded state when the card is tapped.
- The colors property customizes the card's background color.

### 4. Card Content Layout: Inside the Card, another Column is used to arrange the content. This inner Column will contain elements that appear or disappear based on the expanded state, thereby triggering the size change and the animation of the parent Card. The snippet ends before showing the conditional content, but it typically includes a title and then other elements that are shown or hidden.

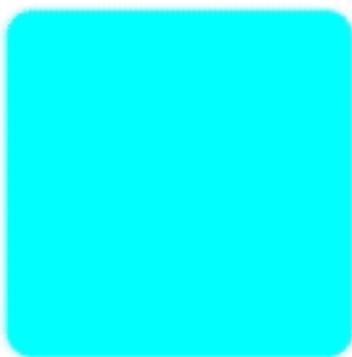
```
1 /**
2  * Demonstrates layout size changes with smooth animation using animateContentSize
3  * Shows how to animate the container size when content expands or collapses
4 */
5 @Composable
6 fun LayoutChangeAnimationDemo() {
7     // State to track expanded/collapsed status
8     var expanded by remember { mutableStateOf(false) }
9
10    Column {
11        // Section title
12        Text("Layout Change Animation", fontWeight = FontWeight.Bold)
13
14        Spacer(modifier = Modifier.height(8.dp))
15
16        // Card that animates its size when content changes
17        Card(
18            modifier = Modifier
19                .fillMaxWidth()
20                .animateContentSize() // Animates size changes when content changes
21                .animationSpec = spring(
22                    dampingRatio = Spring.DampingRatioMediumBouncy, // Controls bounciness
23                    stiffness = Spring.StiffnessLow // Controls animation speed
24                )
25        )
26        .clickable { expanded = !expanded }, // Toggle expanded state on click
27        colors = CardDefaults.cardColors(containerColor = Color(0xFF90CAF9)) // Light blue color
28    ) {
29        Column(
30            modifier = Modifier.padding(16.dp)
31        ) {
32            // Card title
33            Text(
34                text = "Animate Layout Changes",
35                fontWeight = FontWeight.Bold
36            )
37
38            Spacer(modifier = Modifier.height(8.dp))
39
40            // Conditional content that appears/disappears with animation
41            if (expanded) {
42                // Additional text that appears when expanded
43                Text(
44                    text = "This is an example of animating layout changes with animateContentSize modifier. " +
45                        "This paragraph appears and disappears with a smooth animation that adjusts the card's size. " +
46                        "Click anywhere on this card to toggle the expanded state.",
47                    modifier = Modifier.padding(top = 8.dp)
48                )
49
50            Spacer(modifier = Modifier.height(8.dp))
51        }
```

```
52         // Buttons that appear when expanded
53         Row(
54             modifier = Modifier.fillMaxWidth(),
55             horizontalArrangement = Arrangement.spacedBy(8.dp)
56         ) {
57             // First action button
58             Button(
59                 onClick = { }, // No action defined
60                 modifier = Modifier.weight(1f),
61                 colors = ButtonDefaults.buttonColors(containerColor = Color(0xFF5C6BC0))
62             ) {
63                 Text("Action 1")
64             }
65
66             // Second action button
67             Button(
68                 onClick = { }, // No action defined
69                 modifier = Modifier.weight(1f),
70                 colors = ButtonDefaults.buttonColors(containerColor = Color(0xFF5C6BC0))
71             ) {
72                 Text("Action 2")
73             }
74         } else {
75             // Simple instruction text when collapsed
76             Text("Tap to expand", fontSize = 12.sp)
77         }
78     }
79 }
80 }
81 }
82 }
```

## Output

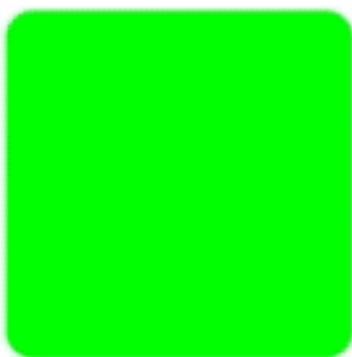


## Color Change Animation



Tap to change color

## Size Animation with Spring



Tap to resize

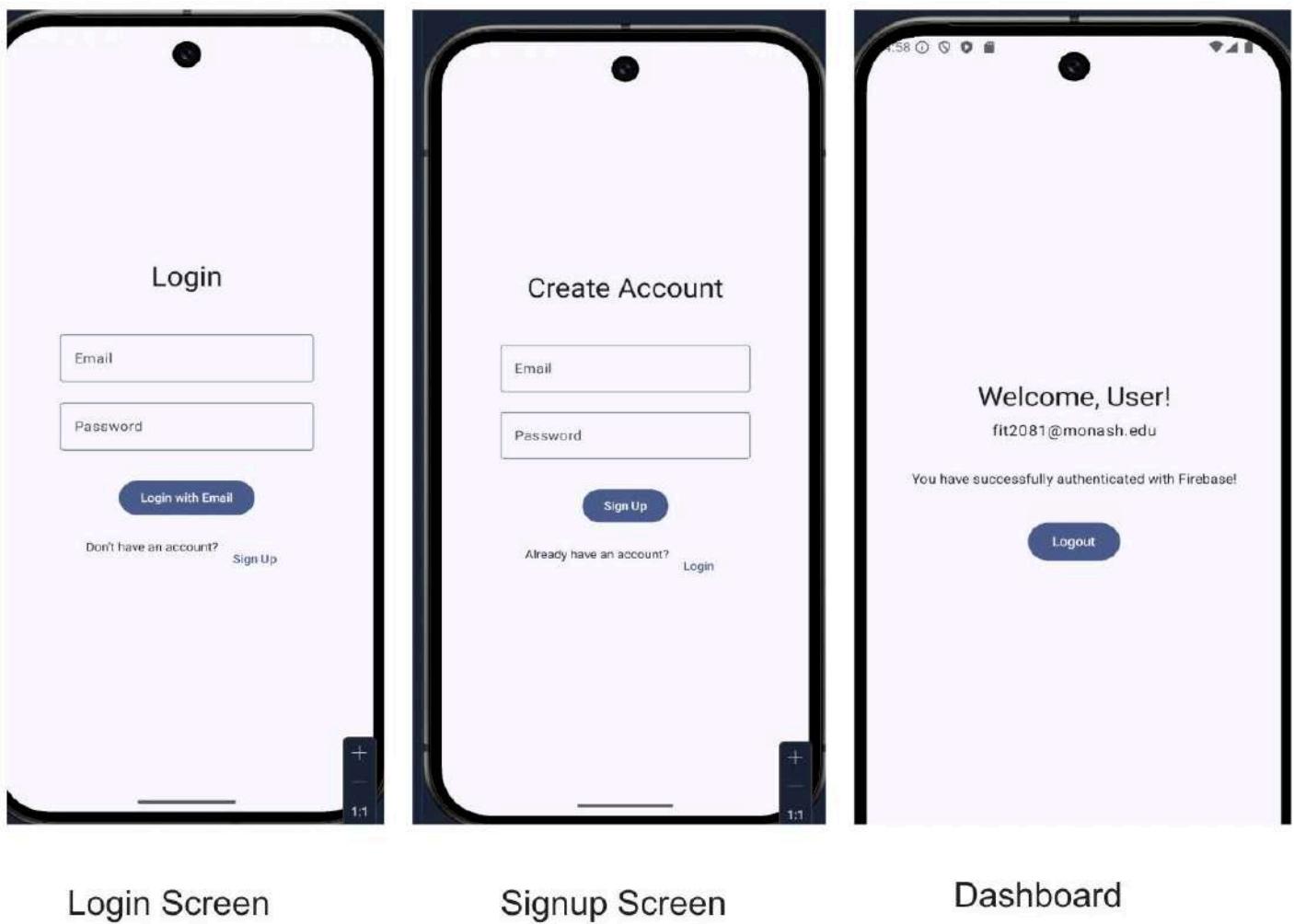
Animate

Spring

Spring physics: DampingRatio=0.5, Stiffness=1500.0

# APP 3: Authentication with Google - Part I - MainActivity

This app introduces you to **user authentication using Firebase**, focusing specifically on implementing **Google Sign-In and email/password login** in a Compose-based Android application. The goal is to understand secure authentication flows, session handling, and profile data retrieval through Firebase Authentication.



Login Screen

Signup Screen

Dashboard

## What You'll Build

A modern Android app that allows users to sign in using either:

- **Email and password**
- **Google account** (via Credential Manager API)

The app includes two screens:

- **MainActivity** – Displays the signed-in user's profile (name + email) (**Part I**)
- **LoginActivity** – Handles login and user input validation (**Part II**)

## By completing this lab, you will:

- Set up Firebase Authentication in an Android project
- Implement **Google Sign-In** using modern APIs
- Handle secure **email/password login** with proper validation
- Manage **authenticated vs unauthenticated states** in app navigation
- Retrieve and display **user profile information** using Firebase

## Class MainActivity

The MainActivity class serves as the primary screen displayed to the user after a successful login. It's an Android Activity, meaning it represents a single, focused thing that the user can do.

Structurally, MainActivity inherits from ComponentActivity, which is a base class for activities that use Jetpack Compose for their UI. It declares a private property auth of type FirebaseAuth. This instance is used to interact with Firebase Authentication services, such as checking the current user's login status. The onCreate method is a standard Android lifecycle callback that's invoked when the activity is being created. Inside onCreate:

1. enableEdgeToEdge() is called, likely to allow the app's UI to draw under the system bars (status bar and navigation bar) for a more immersive look.
2. auth is initialized by getting an instance of FirebaseAuth.
3. It then checks if a user is currently signed in (auth.currentUser). If currentUser is null, it means no user is logged in, and the intention (though the code is incomplete in the snippet) is to redirect the user to a LoginActivity. If a user is logged in, this activity would typically proceed to display the main content, such as a user profile.

```
1  /**
2  * MainActivity displays the user profile after successful authentication.
3  * It checks if the user is logged in and redirects to LoginActivity if not.
4  */
5  class MainActivity : ComponentActivity() {
6      // Firebase Authentication instance for user management
7      private lateinit var auth: FirebaseAuth
8
9      /**
10     * Called when the activity is first created.
11     * Initializes Firebase Auth and sets up the UI.
12     */
13    override fun onCreate(savedInstanceState: Bundle?) {
14        super.onCreate(savedInstanceState)
15        // Enable edge-to-edge display mode for better UI experience
16        enableEdgeToEdge()
17
18        // Initialize Firebase Auth instance to manage user authentication
19        auth = FirebaseAuth.getInstance()
```

```
1>     auth = FirebaseAuth.getInstance()
2>
3>     // Check if user is signed in by getting the current user object
4>     val currentUser = auth.currentUser
5>     if (currentUser == null) {
6>         // If no user is logged in, redirect to LoginActivity
7>         navigateToLogin()
8>         return // Exit onCreate to prevent setting content for this activity
9>     }
10>
11>     // Set up the UI using Jetpack Compose
12>     setContent {
13>         // Apply the app theme
14>         Week10_authTheme {
15>             // Create a full-screen surface with the theme's background color
16>             Surface(
17>                 modifier = Modifier.fillMaxSize(),
18>                 color = MaterialTheme.colorScheme.background
19>             ) {
20>                 // Scaffold provides the basic Material Design visual layout structure
21>                 Scaffold { innerPadding ->
22>                     // Display the user profile composable with user information
23>                     UserProfile(
24>                         // Use displayName if available, otherwise default to "User"
25>                         displayName = currentUser.displayName ?: "User",
26>                         // Use email if available, otherwise provide a default message
27>                         email = currentUser.email ?: "No email available",
28>                         // Pass the logout function to be called when logout button is clicked
29>                         onLogout = { logout() },
30>                         // Apply padding from the scaffold
31>                         modifier = Modifier.padding(innerPadding)
32>                     )
33>                 }
34>             }
35>         }
36>     }
37>
38>     /**
39>      * Signs out the current user from Firebase Auth and navigates back to login screen.
40>      */
41>     private fun logout() {
42>         // Sign out the current user from Firebase Authentication
43>         auth.signOut()
44>         // Display a toast message to confirm successful logout
45>         Toast.makeText(this, "Logged out successfully", Toast.LENGTH_SHORT).show()
46>         // Navigate back to the login screen
47>         navigateToLogin()
48>     }
49>
50>     /**
51>      * Starts the LoginActivity and finishes the current activity.
52>      */
53>     private fun navigateToLogin() {
54>         // Create an intent to launch LoginActivity
55>         val intent = Intent(this, LoginActivity::class.java)
56>         // Start the LoginActivity
57>         startActivity(intent)
58>         // Close MainActivity to prevent returning to it using the back button
59>         finish()
60>     }
61> }
62>
```

# UserProfile

The UserProfile function is a Composable function, a core building block in Jetpack Compose for creating UI elements. Its primary role is to display user-specific information like their displayName and email, and to provide a way for the user to log out via the onLogout action.

Structurally, it uses a Column to arrange UI elements vertically. Inside this Column, it displays two Text elements for the welcome message and email, and it's designed to include a logout button (though the button's code isn't fully visible in the provided snippet). The modifier parameter allows for customisation of the Column's layout and appearance from where UserProfile is called. The horizontalAlignment and verticalArrangement properties of the Column ensure that its contents are centred on the screen. Spacer elements are used to add visual separation between UI components.

```
1  /**
2   * Composable function that displays the user profile information.
3   *
4   * @param displayName The name of the user to display
5   * @param email The email address of the user
6   * @param onLogout Function to call when the logout button is clicked
7   * @param modifier Optional modifier for customizing the layout
8   */
9  @Composable
10 fun UserProfile(
11     displayName: String,
12     email: String,
13     onLogout: () -> Unit,
14     modifier: Modifier = Modifier
15 ) {
16     // Column layout to arrange items vertically
17     Column(
18         modifier = modifier
19             .fillMaxSize() // Use the entire available space
20             .padding(16.dp), // Add padding around the content
21         horizontalAlignment = Alignment.CenterHorizontally, // Center items horizontally
22         verticalArrangement = Arrangement.Center // Center items vertically
23     ) {
24         // Display the welcome message with the user's name
25         Text(
26             text = "Welcome, $displayName!",
27             style = MaterialTheme.typography.headlineMedium, // Use headline medium text style
28             textAlign = TextAlign.Center // Center-align the text
29         )
30
31         // Add vertical spacing between elements
32         Spacer(modifier = Modifier.height(8.dp))
33
34         // Display the user's email address
35         Text(
36             text = email,
```

```
37         style = MaterialTheme.typography.bodyLarge, // Use body large text style
38         textAlign = TextAlign.Center // Center-align the text
39     )
40
41     // Add more vertical spacing
42     Spacer(modifier = Modifier.height(32.dp))
43
44     // Display a success message
45     Text(
46         text = "You have successfully authenticated with Firebase!",
47         style = MaterialTheme.typography.bodyMedium, // Use body medium text style
48         textAlign = TextAlign.Center // Center-align the text
49     )
50
51     // Add more vertical spacing before the logout button
52     Spacer(modifier = Modifier.height(32.dp))
53
54     // Display a logout button that calls the onLogout function when clicked
55     Button(onClick = onLogout) {
56         Text("Logout") // Button text
57     }
58 }
59 }
```

# APP 3: Authentication with Google - Part II - LoginActivity

The LoginActivity class is designed to handle user authentication within your Android application. It acts as the entry point for users to sign in or create new accounts.

Structurally, it's an Activity, inheriting from ComponentActivity, which is a common base class for activities that use Jetpack Compose. It declares two key private properties:

1. auth: An instance of FirebaseAuth, which is the primary tool from Firebase for managing user accounts, including email/password and social logins.
2. credentialManager: An instance of CredentialManager, likely used here to facilitate Google Sign-In by securely handling user credentials.

The onCreate method, a standard Android lifecycle callback, is overridden to perform initial setup. This includes:

- o Initializing the auth instance by calling FirebaseAuth.getInstance().
- o Initializing the credentialManager using CredentialManager.create(this).
- o Checking if a user is already signed in (auth.currentUser). If a user session exists, the app would typically navigate to the main part of the application, bypassing the login screen.

```
1  /**
2   * LoginActivity manages user authentication using Firebase Authentication.
3   * It allows users to sign in with email/password or Google Sign-In, and also provides
4   * functionality for new users to create accounts.
5   */
6  class LoginActivity : ComponentActivity() {
7      // Firebase Authentication instance for user authentication operations
8      private lateinit var auth: FirebaseAuth
9      // Credential Manager to handle Google Sign-In
10     private lateinit var credentialManager: CredentialManager
11
12     /**
13      * Called when the activity is first created.
14      * Initializes Firebase Auth, Credential Manager, and checks if a user is already logged in.
15      * Sets up the UI for login/signup if no user is currently authenticated.
16      */
17     override fun onCreate(savedInstanceState: Bundle?) {
18         super.onCreate(savedInstanceState)
19
20         // Initialize Firebase Auth instance
21         auth = FirebaseAuth.getInstance()
22
23         // Initialize Credential Manager for Google Sign-In
24         credentialManager = CredentialManager.create(this)
25
26         // Check if a user is already logged in
27         val currentUser = auth.currentUser
28         if (currentUser != null) {
29             // User is already signed in, navigate to MainActivity
30             navigateToMainActivity()
31             return // Exit onCreate to prevent setting content for this activity
32         }
33
34         // Set up the UI using Jetpack Compose
35         setContent {
36             // Apply the app theme
37             Week10_authTheme {
38                 // Create a full screen surface with the theme's background color
39             }
40         }
41     }
42 }
```

```
39         Surface(
40             modifier = Modifier.fillMaxSize(),
41             color = MaterialTheme.colorScheme.background
42         ) {
43             // Display the login screen composable
44             LoginScreen(
45                 // Function to handle email/password login
46                 onEmailPasswordLogin = { email, password →
47                     loginWithEmailPassword(email, password)
48                 },
49                 // Function to handle user sign up
50                 onSignUp = { email, password →
51                     signUpWithEmailPassword(email, password)
52                 }
53             )
54         }
55     }
56 }
57 */
58
59 /**
60 * Attempts to authenticate a user with the provided email and password.
61 * Validates inputs and shows appropriate error messages if authentication fails.
62 *
63 * @param email The user's email address
64 * @param password The user's password
65 */
66 private fun loginWithEmailPassword(email: String, password: String) {
67     // Validate that email and password fields aren't empty
68     if (email.isBlank() || password.isBlank()) {
69         showToast("Email and password cannot be empty")
70         return
71     }
72
73     // Attempt to sign in with provided credentials using Firebase Auth
74     auth.signInWithEmailAndPassword(email, password)
75         .addOnCompleteListener(this) { task →
76             if (task.isSuccessful) {
77                 // Sign in success, navigate to MainActivity
78                 navigateToMainActivity()
79             } else {
80                 // If sign in fails, display the error message to the user
81                 showToast("Authentication failed: ${task.exception?.message}")
82             }
83         }
84 }
85 */
86 /**
87 * Creates a new user account with the provided email and password.
88 * Validates inputs and shows appropriate error messages if sign up fails.
89 *
90 * @param email The email address for the new account
91 * @param password The password for the new account
92 */
93 private fun signUpWithEmailPassword(email: String, password: String) {
94     // Validate that email and password fields aren't empty
95     if (email.isBlank() || password.isBlank()) {
96         showToast("Email and password cannot be empty")
97         return
98     }
99
100    // Validate password length - Firebase requires at least 6 characters
101    if (password.length < 6) {
102        showToast("Password should be at least 6 characters")
103        return
104    }
105
106    // Attempt to create a new user account with Firebase Auth
107    auth.createUserWithEmailAndPassword(email, password)
108        .addOnCompleteListener(this) { task →
109            if (task.isSuccessful) {
110                // Sign up success, notify user and navigate to MainActivity
111                showToast("Account created successfully")
112                navigateToMainActivity()
113            } else {
```

```
114             // If sign up fails, display the error message to the user
115             showToast("Sign up failed: ${task.exception?.message}")
116         }
117     }
118 }
119
120 /**
121 * Initiates the Google Sign-In flow using Credential Manager API.
122 * This function launches a coroutine to handle the asynchronous authentication process.
123 */
124 private fun loginWithGoogle() {
125     // Launch a coroutine for the asynchronous authentication process
126     lifecycleScope.launch {
127         // Create Google ID token request option with the server client ID
128         val googleIdOption = GetGoogleIdOption.Builder()
129             .setServerClientId("project-944555265112") // Replace with your web client ID from Google Cloud Console
130             .build()
131
132         // Build the credential request with the Google ID option
133         val request = GetCredentialRequest.Builder()
134             .addCredentialOption(googleIdOption)
135             .build()
136
137         try {
138             // Request credential from the Credential Manager
139             val result = credentialManager.getCredential(
140                 request = request,
141                 context = this@LoginActivity
142             )
143             // Handle the sign-in result
144             handleSignIn(result)
145         } catch (e: GetCredentialException) {
146             // Display error message if Google Sign-In fails
147             showToast("Google Sign-In failed: ${e.message}")
148         }
149     }
150 }
151
152 /**
153 * Processes the credential received from Google Sign-In and authenticates with Firebase.
154 *
155 * @param result The credential response from the Credential Manager
156 */
157 private fun handleSignIn(result: GetCredentialResponse) {
158     when (val credential = result.credential) {
159         is CustomCredential → {
160             // Check if the credential is a Google ID token
161             if (credential.type == GoogleIdTokenCredential.TYPE_GOOGLE_ID_TOKEN_CREDENTIAL) {
162                 try {
163                     // Parse the Google ID token credential from the raw data
164                     val googleIdTokenCredential = GoogleIdTokenCredential.createFrom(credential.data)
165
166                     // Extract the ID token string
167                     val idToken = googleIdTokenCredential.idToken
168
169                     // Create a Firebase credential from the Google ID token
170                     val firebaseCredential = GoogleAuthProvider.getCredential(idToken, null)
171
172                     // Sign in to Firebase with the Google credentials
173                     auth.signInWithCredential(firebaseCredential)
174                         .addOnCompleteListener(this) { task →
175                             if (task.isSuccessful) {
176                                 // Sign in success, navigate to MainActivity
177                                 navigateToMainActivity()
178                             } else {
179                                 // If Firebase authentication fails, display the error message
180                                 showToast("Firebase Authentication failed: ${task.exception?.message}")
181                             }
182                         }
183                     } catch (e: GoogleIdTokenParsingException) {
184                         // Handle error in parsing the Google ID token
185                         showToast("Error parsing Google ID token: ${e.message}")
186                     }
187                 } else {
188                     // Handle case where credential is not a Google ID token
189                 }
190             }
191         }
192     }
193 }
```

```

189             showToast("Unsupported credential type")
190         }
191     }
192     else -> {
193         // Handle unsupported credential types
194         showToast("Unsupported credential type")
195     }
196 }
197 }
198 /**
199 * Navigates to the MainActivity and closes the LoginActivity.
200 */
201 private fun navigateToMainActivity() {
202     // Create an intent to launch MainActivity
203     val intent = Intent(this, MainActivity::class.java)
204     // Start the MainActivity
205     startActivity(intent)
206     // Close LoginActivity to prevent returning to it using the back button
207     finish()
208 }
209 }
210 /**
211 * Displays a short toast message to the user.
212 *
213 * @param message The message to display
214 */
215 private fun showToast(message: String) {
216     Toast.makeText(this, message, Toast.LENGTH_SHORT).show()
217 }
218 }
219 }
220

```

## LoginScreen

The LoginScreen function is a Jetpack Compose composable, designed to build the user interface for a login and sign-up screen.

Its role is to:

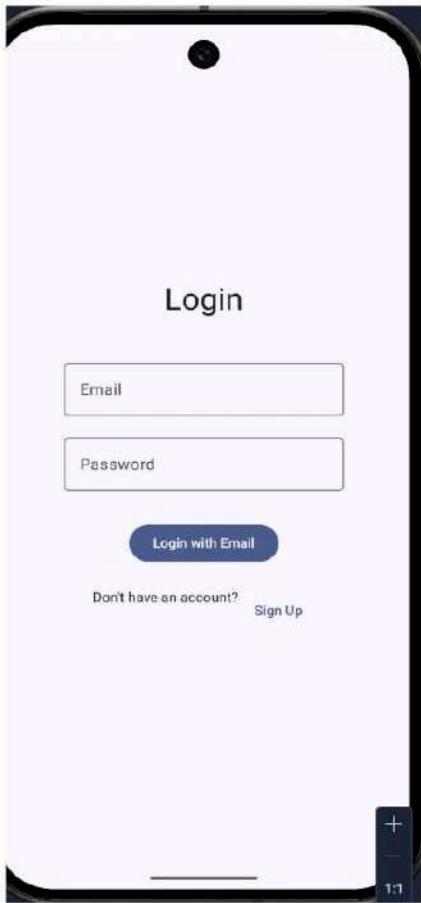
1. Display UI Elements: It renders input fields for email and password, and buttons for submitting login or sign-up attempts.  
Manage State: It uses remember with mutableStateOf to hold and manage the current text in the email and password fields (email, password) and to track whether the user is in "sign up" mode or "login" mode (isSignUp).
2. Handle User Interactions: It accepts two lambda functions as parameters:
  - o onEmailPasswordLogin: This function is invoked when the user attempts to log in, passing the entered email and password.
  - o onSignUp: This function is invoked when the user attempts to sign up, also passing the email and password.
3. Structure Layout: It uses a Column composable to arrange its child elements (like text fields and buttons) vertically. The Column is modified to fill the entire screen (fillMaxSize()) and center its contents both horizontally (Alignment.CenterHorizontally) and vertically (Arrangement.Center), with some padding around the edges (padding(16.dp)).

The `@OptIn(ExperimentalMaterial3Api::class)` annotation indicates that this function uses components from Material 3 that are still considered experimental, like `OutlinedTextField` which is likely used later in the function body. The `LocalContext.current` is used to get access to the Android Context, which might be needed for operations like showing Toasts or accessing resources.

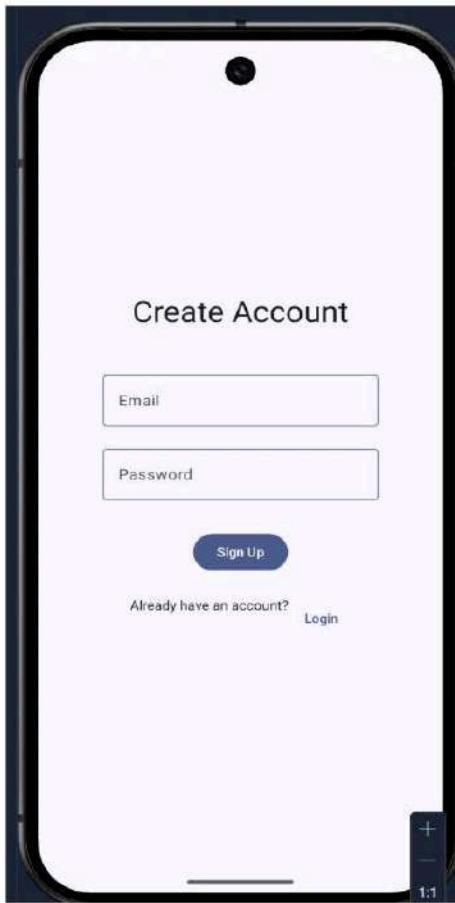
```
1  /**
2   * Composable function that displays a login screen with email and password fields.
3   * Provides options for both login and signup.
4   *
5   * @param onEmailPasswordLogin Function to call when user attempts to login with email/password
6   * @param onSignUp Function to call when user attempts to sign up with email/password
7   */
8  @OptIn(ExperimentalMaterial3Api::class) // Required for OutlinedTextField
9  @Composable
10 fun LoginScreen(
11     onEmailPasswordLogin: (email: String, password: String) → Unit,
12     onSignUp: (email: String, password: String) → Unit,
13 ) {
14     // Get access to the current context for UI operations
15     val context = LocalContext.current
16
17     // State variables to store user input and UI state
18     var email by remember { mutableStateOf("") } // Email input state
19     var password by remember { mutableStateOf("") } // Password input state
20     var isSignUp by remember { mutableStateOf(false) } // Toggle between login and signup mode
21
22     // Column layout to arrange items vertically
23     Column(
24         modifier = Modifier
25             .fillMaxSize() // Use the entire available space
26             .padding(16.dp), // Add padding around the content
27             horizontalAlignment = Alignment.CenterHorizontally, // Center items horizontally
28             verticalArrangement = Arrangement.Center // Center items vertically
29     ) {
30         // Display the title based on the current mode (login or sign up)
31         Text(
32             text = if (isSignUp) "Create Account" else "Login",
33             style = MaterialTheme.typography.headlineLarge // Use headline large text style
34         )
35
36         // Add vertical spacing between elements
37         Spacer(modifier = Modifier.height(32.dp))
38
39         // Email input field
40         OutlinedTextField(
41             value = email, // Current email value
42             onValueChange = { email = it }, // Update email when user types
43             label = { Text("Email") }, // Label for the field
44             modifier = Modifier.padding(vertical = 8.dp) // Add vertical padding
45         )
46
47         // Password input field with hidden text
48         OutlinedTextField(
49             value = password, // Current password value
50             onValueChange = { password = it }, // Update password when user types
51             label = { Text("Password") }, // Label for the field
52             visualTransformation = PasswordVisualTransformation(), // Hide password characters
53         )
54     }
55 }
```

```
53     modifier = Modifier.padding(vertical = 8.dp) // Add vertical padding
54 )
55
56     // Add more vertical spacing
57     Spacer(modifier = Modifier.height(16.dp))
58
59     // Login/Sign Up button
60     Button(
61         onClick = {
62             // Call appropriate function based on current mode
63             if (isSignUp) {
64                 onSignUp(email, password)
65             } else {
66                 onEmailPasswordLogin(email, password)
67             }
68         },
69         modifier = Modifier.padding(vertical = 8.dp) // Add vertical padding
70     ) {
71         // Button text changes based on mode
72         Text(if (isSignUp) "Sign Up" else "Login with Email")
73     }
74
75     // Add more vertical spacing
76     Spacer(modifier = Modifier.height(16.dp))
77
78     // Row layout for the toggle text and button
79     Row(
80         modifier = Modifier.fillMaxWidth(), // Use full width
81         horizontalArrangement = Arrangement.Center // Center items horizontally
82     ) {
83         // Toggle prompt text
84         Text(
85             text = if (isSignUp) "Already have an account? " else "Don't have an account? ",
86             style = MaterialTheme.typography.bodyMedium // Use body medium text style
87         )
88
89         // Toggle button to switch between login and signup modes
90         TextButton(onClick = { isSignUp = !isSignUp }) {
91             Text(if (isSignUp) "Login" else "Sign Up") // Button text
92         }
93     }
94 }
95 }
```

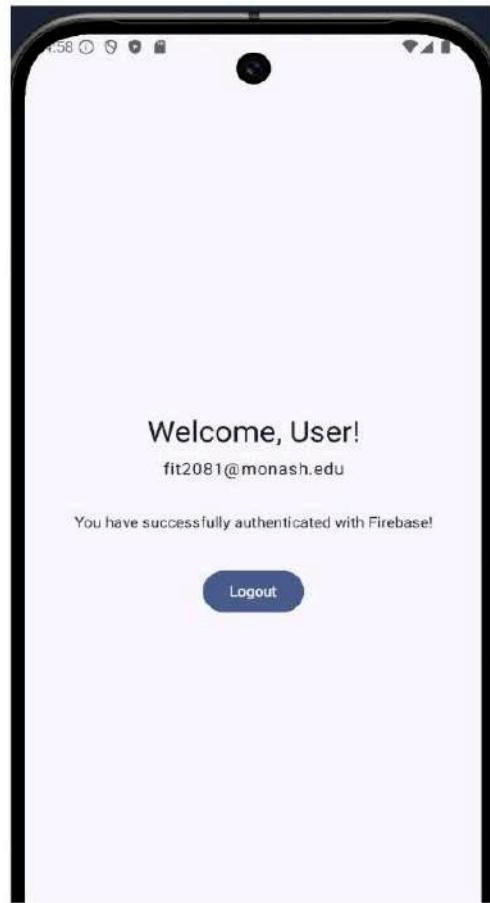
## Expected Output



Login Screen



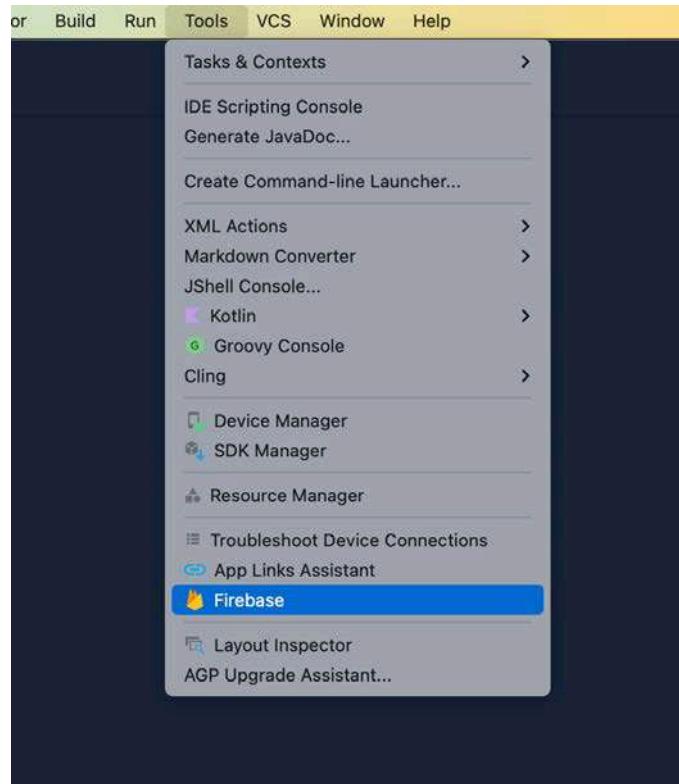
Signup Screen



Dashboard

# APP 3: How to connect your App to Google Firebase

**Step 1:** From Android Studio, select Tools-->Firebase



**Step 2:** Now, from the Firebase pane, click on "Authentication" - "Authentication using Google"

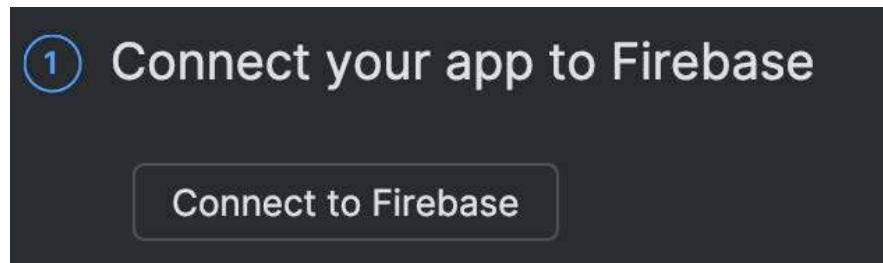
A screenshot of the Firebase console. At the top left is the yellow flame logo followed by the word 'Firebase'. Below this is a promotional message: 'Firebase gives you the tools and infrastructure from Google to help you develop, grow and earn money from your app.' with a 'Learn more' link. The main content area has two sections: 'Analytics' (with a description) and 'Authentication'. Under 'Authentication', there is a list of links: 'Authenticate using Google' (which is circled in red), 'Authenticate using Google [Java]', 'Authenticate using Facebook Login', 'Authenticate using Facebook Login [Java]', 'Authenticate using a custom authentication system', and 'Authenticate using a custom authentication system [Java]'.

- > **Analytics**  
Measure user activity and engagement with free, easy, and unlimited analytics. [More info](#)
- > **Authentication**  
Sign in and manage users with ease using popular login providers like Google, Facebook, and others. You can even use a custom authentication system. [More info](#)
  - (D) [Authenticate using Google](#)
  - (D) [Authenticate using Google \[Java\]](#)
  - (D) [Authenticate using Facebook Login](#)
  - (D) [Authenticate using Facebook Login \[Java\]](#)
  - (D) [Authenticate using a custom authentication system](#)
  - (D) [Authenticate using a custom authentication system \[Java\]](#)

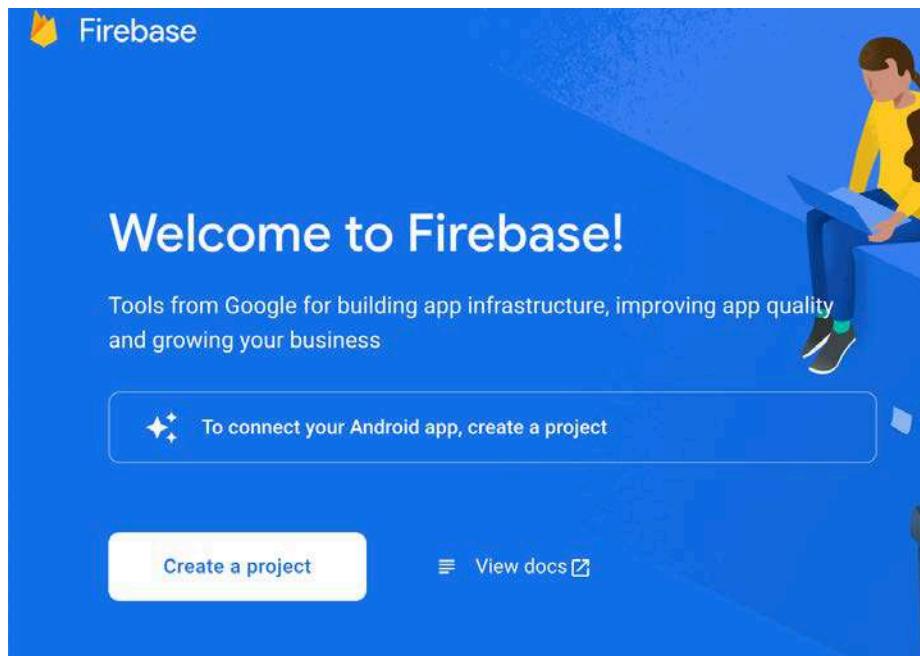


Before going to the next step, open you email (or Google account) using your Monash ID. Close all the tabs that use your private Google account (if possible)

**Step 3:** Click on 'Connect to Firebase',



**Step 4:** which should open a new tab in your browser. Click on 'Create a Project'.



Give a project name

**Let's start with a name for  
your project<sup>®</sup>**

Project name

**bookstoreapp**

bookstoreapp-58912

monash.edu

**Continue**

and disable the analytics (for simplicity)

## Google Analytics for your Firebase project

Google Analytics is a free and unlimited analytics solution that enables targeting, reporting and more in Firebase Crashlytics, Cloud Messaging, in-app messaging, Remote Config, A/B Testing and Cloud Functions.

Google Analytics enables:

- A/B testing ?
- Crash-free users ?
- User segmentation and targeting across Firebase products ?
- Event-based Cloud Functions triggers ?
- Free unlimited reporting ?

Enable Google Analytics for this project  
Recommended

[Previous](#)

[Create project](#)

Click 'Continue' and then 'Connect'



**Step 5:** Now go back to Android Studio and click "Add the Firebase Authentication SDK to your app".

You should get both steps done successfully.

## ② Add the Firebase Authentication SDK to your app

[Add the Firebase Authentication SDK to your app](#)