



Participant Handbook

Sector
Telecom

Sub-Sector
Handset

Occupation
**Terminal Equipment Application
Developer**

Reference ID: **TEL/Q2300**, Version **6.0**
NSQF Level **4**



**Android Application
Technician - Telecom
Devices**

This book is sponsored by

Telecom Sector Skill Council

Estel House, 3rd Floor, Plot No: - 126, Sector-44

Gurgaon, Haryana 122003

Phone: 0124-2222222

Email: tssc@tsscindia.com

Website: www.tsscindia.com

All Rights Reserved

Second Edition, October 2025

Under Creative Commons License: CC BY-NC-SA

Copyright © 2025

Attribution-Share Alike: CC BY-NC-SA



Disclaimer

The information contained herein has been obtained from sources reliable to Telecom Sector Skill Council. Telecom Sector Skill Council disclaims all warranties to the accuracy, completeness or adequacy of such information. Telecom Sector Skill Council shall have no liability for errors, omissions, or inadequacies, in the information contained herein, or for interpretations thereof. Every effort has been made to trace the owners of the copyright material included in the book. The publishers would be grateful for any omissions brought to their notice for acknowledgements in future editions of the book. No entity in Telecom Sector Skill Council shall be responsible for any loss whatsoever, sustained by any person who relies on this material. The material in this publication is copyrighted. No parts of this publication may be reproduced, stored or distributed in any form or by any means either on paper or electronic media, unless authorized by the Telecom Sector Skill Council.





Shri Narendra Modi
Prime Minister of India

“ Skilling is building a better India.
If we have to move India towards
development then Skill Development
should be our mission. ”



Certificate

COMPLIANCE TO QUALIFICATION PACK– NATIONAL OCCUPATIONAL STANDARDS

is hereby issued by the

TELECOM SECTOR SKILL COUNCIL

for

SKILLING CONTENT : PARTICIPANT HANDBOOK

Complying to National Occupational Standards of

Job Role/ Qualification Pack: "Android Application Technician - Telecom Devices"

QP No. "TEL/Q2300, NSQF level 4.0"

Date of Issuance:

Valid up to*:

**Valid up to the next review date of the Qualification Pack or the
'Valid up to' date mentioned above (whichever is earlier)*

Authorised Signatory
(Telecom Sector Skill Council)

Acknowledgements

Telecom Sector Skill Council would like to express its gratitude to all the individuals and institutions who contributed in different ways towards the preparation of this “Participant Handbook”. Without their contribution it could not have been completed. Special thanks are extended to those who collaborated in the preparation of its different modules. Sincere appreciation is also extended to all who provided peer review for these modules.

The preparation of this handbook would not have been possible without the Telecom Industry’s support. Industry feedback has been extremely encouraging from inception to conclusion and it is with their input that we have tried to bridge the skill gaps existing today in the industry.

This participant handbook is dedicated to the aspiring youth who desire to achieve special skills which will be a lifelong asset for their future endeavours.

About this book

In the last few years, the growth of India's telecommunications sector has continued to outpace many other segments of the economy. The sector is now one of the major drivers of digital connectivity, technological innovation, and new employment opportunities.

According to the Telecom Regulatory Authority of India (TRAI), the total number of telephone subscribers (wireless + wireline) in India reached approximately 1,199.28 million as on 31 March 2024. India thus remains the second-largest telecommunications market in the world.

Market-size projections indicate that India's telecom services market (in terms of services revenue) is expected to grow at a compound annual growth rate (CAGR) of around 9.2 % during 2024-30.

Key growth drivers for the sector include:

- Rapid uptake of mobile broadband and data services (e.g., wireless data usage volume in India increased by ~17.46 % in 2024-25)
- The rollout and expansion of 5G infrastructure and services, and the shift towards fibre-based fixed broadband.
- Strong government support including policy reforms, manufacturing incentives, and investment in digital infrastructure.
- Changing consumer preferences: higher smartphone penetration, more data usage, value-added services, and digital ecosystems.
- The government's upcoming policy outlines a target to create one million new jobs by 2030 in the telecom sector.

This Participant Handbook is designed to impart theoretical and practical skill training to students for becoming an Android Application Technician - Telecom Devices in the telecom sector.

An Android Application Technician – Telecom Devices is responsible for assisting in the setup, configuration and basic implementation of Android-based applications on telecom devices. The individual assists in activities such as preparing the development environment, configuring user interfaces using standard layouts, application testing and providing technical assistance during application deployment. The role focuses on implementing and maintaining existing application components under supervision.

Key responsibilities include:

- Assist in Setting Up Development Environment: Prepare and configure the Android development environment, including software tools, SDKs, emulators, and device connectivity for telecom applications.
- Support User Interface Configuration: Assist in creating and modifying user interfaces using standard Android layouts, controls, and design principles as per project or telecom device requirements.
- Configure Value Added Services (VAS): Support the integration and configuration of VAS features such as messaging, location-based services, data synchronization, and other telecom-specific functionalities.
- Assist in Application Testing and Debugging: Conduct functional and compatibility testing of Android applications on various telecom devices; identify, report, and help resolve technical issues or bugs.
- Assist Application Deployment: Assist in packaging and publishing Android applications for telecom devices, ensuring compliance with device specifications and organizational standards.
- Follow Quality and Sustainability Practices: Adhere to organizational quality standards, sustainable development practices, and data security protocols during all development and deployment activities.
- Maintain Documentation and Reports: Record test results, configuration details, and updates; assist in maintaining project documentation for reference and future enhancements.

- **Provide Technical Support:** Offer basic technical assistance to team members and users during installation, configuration, or troubleshooting of Android applications.
- **Collaborate with Development Teams:** Work under supervision to support developers and testers, ensuring timely completion of assigned modules and adherence to project timelines.

Aligned with the latest and approved version of Android Application Technician – Telecom Devices Qualification Pack (TEL/Q2300), the handbook includes the following National Occupational Standards (NOSs):

1. TEL/N2300: Assist in Configuring Android Development Environment and User Interface for Telecom Devices
2. TEL/N2301: Assist in Configuring Value Added Services (VAS) in Android Applications for Telecom Devices
3. TEL/N2302: Assist in Testing and Publishing Android Applications for Telecom Devices
4. TEL/N9110: Follow sustainability practices in the development of mobile applications
5. DGT/VSQ/N0101: Employability Skills (30 Hours)

We trust this Participant Handbook will offer strong learning support and help budding professionals carve out engaging and rewarding careers in India's dynamic telecom industry.

Symbols Used



Key Learning
Outcomes



Steps



Notes



Practical



Unit
Objectives

Table of Contents

1.	Role and Responsibilities of an Android Application Technician – Telecom Devices (TEL/N2300)	1
	Unit 1.1 - Introduction to the Program	3
	Unit 1.2 - History of Communication	5
	Unit 1.3 - Signals	9
	Unit 1.4 - Networks	11
	Unit 1.5 - Channel Access Methods	14
	Unit 1.6 - Mobile Operating Systems	17
	Unit 1.7 - Legacy Mobile Operating Systems and Concepts	22
	Unit 1.8 - Android Operating System and Version History	26
	Unit 1.9 - Android Device Configuration and Development Requirements	31
	Unit 1.10 - Android Studio Installation and Setup	35
2.	Setting up Android framework/ Development Environment and creating user interface (TEL/N2300)	43
	Unit 2.1 - Creating Simple Android Project in Android Studio	46
	Unit 2.2 - Running the Project in Android Studio	49
	Unit 2.3 - Creating and Running a Simple User Interface	52
3.	Configuring Value Added Services (VAS) in Android Applications for Telecom Devices (TEL/N2301)	59
	Unit 3.1 - Managing Data within Android Applications	61
	Unit 3.2 - Messaging and Networking Service Integration on Android	89
	Unit 3.3 - Fundamentals of Google Maps Integration and Location-Based Services	98
	Unit 3.4 - Background Android Services	107
4.	Testing and Publishing Android Applications for Telecom Devices (TEL/N2302)	127
	Unit 4.1 - Testing and Validating Android Applications	129
	Unit 4.2 - Publishing Android Applications	143
5.	Sustainably Practices in the Development of Mobile Applications (TEL/N9110)	155
	Unit 5.1 - Sustainable Coding Practices	157
	Unit 5.2 - Application Performance Optimization	162
	Unit 5.3 - Network and Data Usage Reduction	168
	Unit 5.4 - Environmental Standards Compliance	172
	Unit 5.5 - Sustainable UI/UX Design	176
	Unit 5.6 - Sustainable Development Practices	180
6.	Employability Skills (30 Hours) (DGT/VSQ/N0101)	186
<p>It is recommended that all trainings include the appropriate Employability skills Module. Content for the same is available here: https://www.skillindiadigital.gov.in/content/list</p>		
7.	Annexure	188
	Annexure- I	189

It is recommended that all trainings include the appropriate Employability skills Module. Content for the same is available here: <https://www.skillindiadigital.gov.in/content/list>





1. Role and Responsibilities of an Android Application Technician – Telecom Devices



- Unit 1.1 - Introduction to the Program
- Unit 1.2 - History of Communication
- Unit 1.3 - Signals
- Unit 1.4 - Networks
- Unit 1.5 - Channel Access Methods
- Unit 1.6 - Mobile Operating Systems
- Unit 1.7 - Legacy Mobile Operating Systems and Concepts
- Unit 1.8 - Android Operating System and Version History
- Unit 1.9 - Android Device Configuration and Development Requirements
- Unit 1.10 - Android Studio Installation and Setup

Key Learning Outcomes



By the end of this module, the participants will be able to:

1. Define and classify the means and types of communication systems.
2. Illustrate and explain the various propagation methods used in communication systems.
3. Differentiate and describe the role of networking and the Internet in communication systems.
4. List and apply the common access methods (protocols) for data flow in networks.
5. Describe and distinguish the key features of the different generations of advanced communication systems (e.g., 3G, 4G, 5G).
6. Compare and contrast the architecture and features of major Mobile Operating Systems and their versions (e.g., Android vs. iOS).
7. Identify and summarize the key features and evolution of the Windows Mobile operating system.
8. Identify and distinguish the key features across at least three successive versions of the Android OS.
9. List, justify, and execute the procedure for installing Android based on its hardware requirements on a smartphone.

UNIT 1.1: Introduction to the Program

Unit Objectives



By the end of this unit, the participants will be able to:

1. Define and describe the basic principles and working process of Telecommunication.
2. Differentiate and compare the architecture and key features of major Mobile Operating Systems.
3. Identify and distinguish the key features across at least three successive versions of the Android OS.
4. List and justify the minimum hardware and software requirements for Android installation.
5. Successfully install, configure,

1.1.1 Introduction

Telecommunication is the electronic transmission of information—such as text, images, sound, or data—over significant distances. This process relies on signals propagated through two primary channels: physical media, including wires and optical fibers, and radiation media, primarily through electromagnetic waves.

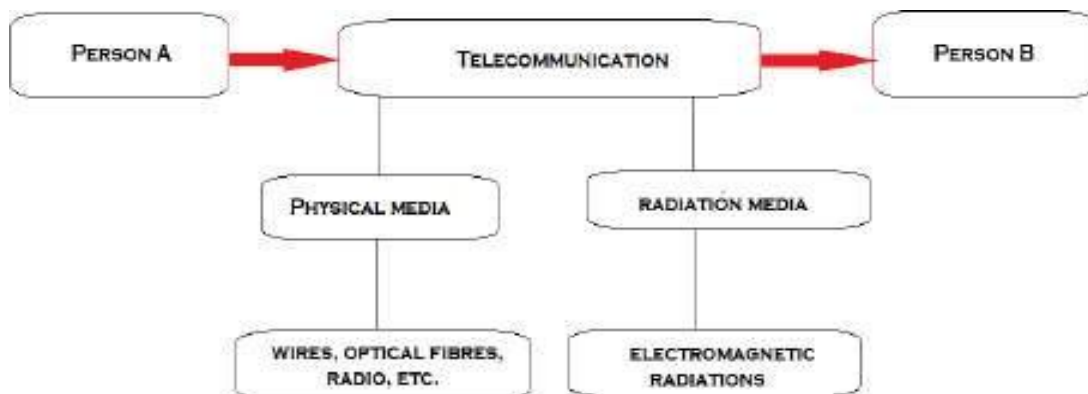


Fig. 1.1.1 Telecommunication process

The above diagram shows how a telecommunication system can be generated between two persons who may be living in two different places.

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 1.2: History of Communication

Unit Objectives



By the end of this unit, the participants will be able to:

1. Classify and describe the major types of communication systems (e.g., analog, digital, simplex, duplex).
2. Illustrate and explain the core networking principles and data flow essential to the working of the Internet.

1.2.1 History of Communication

- **Early Beginnings:** Communication methods date back to the earliest human civilizations, utilizing visual (smoke, flags) and acoustic (drums, shouting) signals.
- **Organized Postal Systems:** The establishment of efficient postal services, notably by the Persians as early as 540 B.C., marked a significant advancement in organized message delivery.
- **Carrier Pigeons:** By the 11th century, domesticated pigeons were widely used as reliable messengers for transmitting information over distances.

1.2.2 Telegraph and Telephone

- **Electric Telegraph:** The concept of the electric telegraph began with early work, such as that by English inventor Francis Ronalds in 1816 using static electricity. This invention was significantly advanced by subsequent scientists, leading to widespread commercial use.
- **The Telephone:** The telephone was successfully demonstrated and patented in 1876 by Alexander Graham Bell (and Elisha Gray, who filed a caveat), enabling real-time voice communication.
- **Foundation for Mobile:** The invention of the telephone laid the crucial groundwork for the development of cellular and mobile phone technology in the 20th century

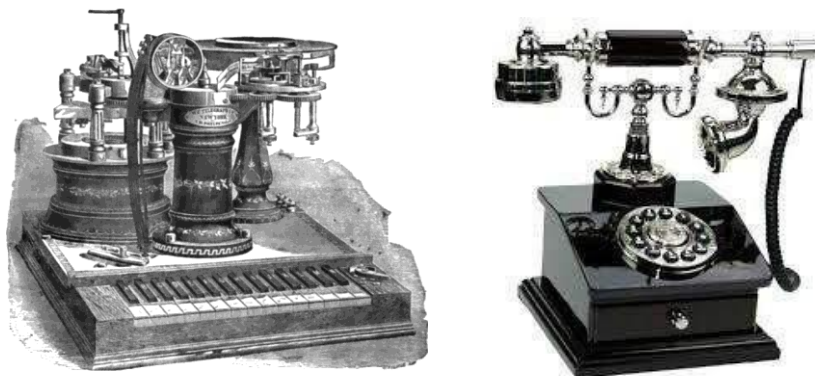


Fig. 1.2.2 A Telegraph and Telephone



Fig. 1.2.3 Evolution of phones

1.1.3 Major Service Players in Telecom Industry

Wireless Operators

Market Share in 2022 (Wireless Subscribers)

As of February 2022, with ~ 1,145 million (114.5 crore) wireless subscribers (including inactive):

- Jio: 35.4 % (\approx 402.7 million users)
- Airtel: 31.5 % (\approx 358.1 million)
- Vodafone-Idea (Vi): 23.2 % (\approx 263.6 million)
- BSNL: 10.0 % (\approx 113.8 million)

These figures sum to ~ 100 % across those four players in the wireless space in that period.

The below graph shows each of these telecom giants' market share as of 2022.

1.2.3 Computers and the Internet

- Computers are electronic devices designed to perform logical and arithmetic operations automatically and at high speed, primarily for processing and storing data.
- The Internet is a global network of interconnected computer networks that communicate using the Internet Protocol (IP).
- The fundamental communication method is packet switching, where digital data (like emails or videos) is broken down into small, standardized units called packets.
- These data packets travel independently across various wired (e.g., optical fibers buried underground) and wireless network connections until they reach their destination, where they are reassembled by the receiving computer.
- This architecture allows data to flow reliably and efficiently between billions of devices worldwide without needing a single centralized mainframe computer for communication.
- The Internet is essential for modern telecommunications, enabling the rapid transfer of complex media—such as emails, documents, songs, and videos—by converting them into signals for transmission.

Fundamental Parts of a Telecommunication System

A telecommunication system fundamentally comprises three essential parts:

1. **Transmitter** : This device takes the original information (e.g., voice, text, data) and converts it into an electrical or electromagnetic signal suitable for transmission. This process is often called encoding or modulation.
2. **Transmission Medium**: This is the physical or non-physical pathway the signal uses to travel from the transmitter to the receiver. Common examples include:
 - Physical Media: Wires (copper) or optical fibers
 - Radiation Media: Electromagnetic waves (radio, microwave, satellite)
3. **Receiver**: Located at the destination, the receiver captures the incoming signal and processes it to convert it back into the original, usable information (data, audio, video) for the end-user. This process is often called decoding or demodulation.

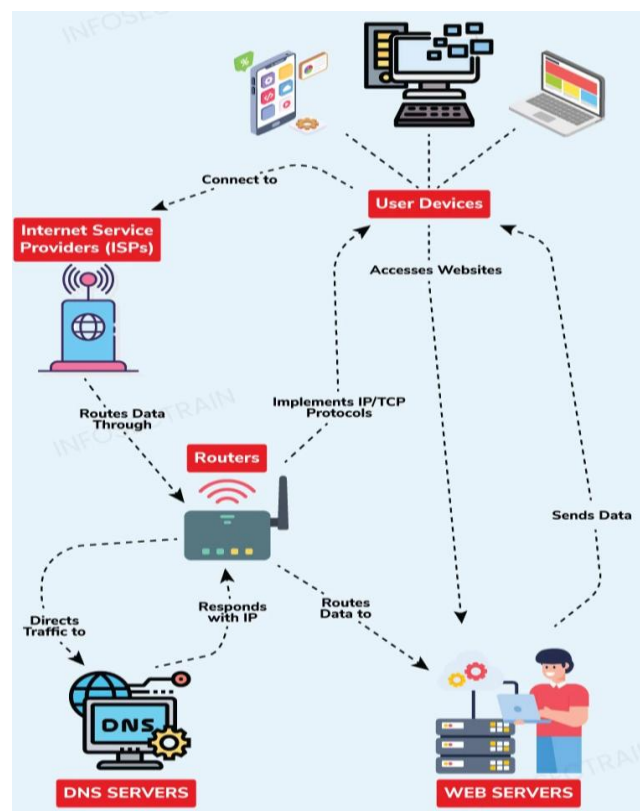


Fig. 1.2.4 Working process of the Internet

Notes



Lined area for taking notes, consisting of 30 horizontal lines.

UNIT 1.3: Signals

Unit Objectives



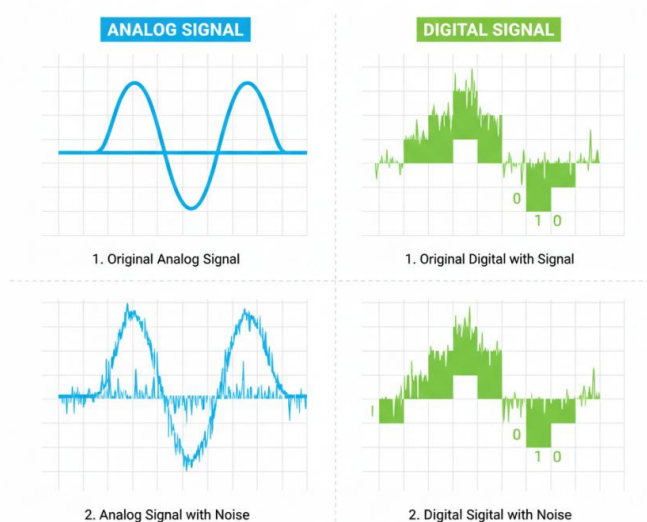
By the end of this unit, the participants will be able to:

1. Classify and describe the major types of communication systems (e.g., analog, digital, simplex, duplex).
2. Illustrate and explain the core networking principles and data flow essential to the working of the Internet.

1.3.1 Signals

- Signals are the critical components in a telecommunication system, serving as the physical or electromagnetic carriers of information (data, voice, video).
- Signals are fundamentally categorized into two types:
 - Analog Signals: These are continuous signals whose properties (amplitude, frequency, or phase) continuously vary in direct proportion to the changes in the original information being transmitted.
 - Digital Signals: These are discrete, non-continuous signals that represent information in a binary form, using distinct, finite values, typically combinations of 0s and 1s (high and low voltage states).
- Signal Degradation (Noise): The energy and fidelity of a transmitted signal are never 100% due to degradation from noise.
- Noise is defined as any unwanted, random energy introduced into the transmission medium. This interference randomly alters the signal's properties (adding or subtracting energy), making it difficult for the receiver to accurately interpret the original information.

ANALOG VS. DIGITAL SIGNALS WITH NOISE



NOISE: Unwanted, random interference that degrades signal quality.

Fig. 1.3.1 Types of Signals

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 1.4: Networks

Unit Objectives

By the end of this unit, the participants will be able to:

1. Define and describe the fundamental concepts and components of networking.
2. Describe and distinguish the key features of the major generations of communication networks (e.g., 2G, 3G, 4G, 5G).

1.4.1 Networks and Signal Integrity

- A telecommunication network is formed by the cooperative operation of transmitters, transmission media, and receivers over large distances to exchange information.
- Switching is necessary for data transfer:
 - In digital communication systems, devices called routers and switches control the path of data packets.
 - In analog systems, physical switches connect and disconnect dedicated circuits between users (known as circuit switching).
- Signal Degradation and Amplification: As a signal travels over long distances, it naturally loses power (attenuation).
- To restore the signal's strength, amplification is required, which is achieved using devices called repeaters (for analog signals) or regenerators (for digital signals).

1.4.2 Mobile Telecommunication Fundamentals

Generations of Mobile Services: Mobile services began with Zero Generation (0G), which were early, limited car phones. Advancements followed with 1G, 2G, 3G, 4G, and 5G.

Basic Mobile Phone Components:

- **Battery:** The essential power source that enables the phone's operation.
- **User Input:** The interface (e.g., keypad or touchscreen) that allows the user to interact with and control the phone's functions.
- **SIM Card (Subscriber Identity Module):** A removable chip that securely stores the user's network identity (phone number) and enables access to network services (calls, texts, data).
- **Storage/Memory Card:** External storage (e.g., microSD) that allows the user to expand the internal memory for storing personal data (images, videos, songs, etc.).

1.4.3: Mobile Communication Standards (Generations)

- The global standards for mobile communication are defined by various bodies, including the 3rd Generation Partnership Project (3GPP), which is a collaboration of telecommunications standards organizations across Asia, Europe, and North America.
- The following are key technologies and their corresponding generations:

Technology	Generation	Description
GSM (Global System for Mobile Communications)	2G	Developed by ETSI and launched in Finland in 1991. It became the first global digital standard. Initially used circuit-switching; later incorporated GPRS (General Packet Radio Service) and EDGE (Enhanced Data rates for GSM Evolution) for basic packet-switched data transfer.
UMTS (Universal Mobile Telecommunications System)	3G	Introduced by 3GPP to offer significantly higher data rates and better voice quality. It utilized W-CDMA (Wideband Code Division Multiple Access) as its radio access technology.
HSPA (High-Speed Packet Access)	Enhanced 3G	A combination of HSDPA (Downlink) and HSUPA (Uplink) introduced to dramatically improve the speed and efficiency of the UMTS (3G) network.
LTE (Long-Term Evolution)	4G	A 3GPP standard specifically developed to achieve true high-speed wireless mobile broadband. It provides a fully IP-based network architecture, resulting in faster data and low latency.
5G	5G	The successor to 4G, defined by 3GPP, designed for extremely high speeds, ultra-low latency, and massive device connectivity.

Notes

This image shows a full page of blank white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page, providing a template for writing or drawing. There are no margins, text, or other markings on the paper.

UNIT 1.5: Channel Access Methods

Unit Objectives

By the end of this unit, the participants will be able to:

1. Explain the concept and principles of multiplexing.
2. Describe the fundamentals of Code Division Multiple Access (CDMA) systems.

1.5.1 Multiplexing

Multiplexing is a technique that combines multiple separate data streams or signals into one composite signal over a shared communication medium. This allows a single physical channel (like a cable or radio frequency) to carry several logical channels simultaneously.

- It is the process of combining multiple independent signals into one signal for transmission over a shared medium.
- The communication channel is logically divided to accommodate the different data streams.
- The device that performs this process is called a Multiplexer (MUX).
- At the receiving end, the original individual signals are separated by a process called demultiplexing, which is performed by a Demultiplexer (DEMUX).

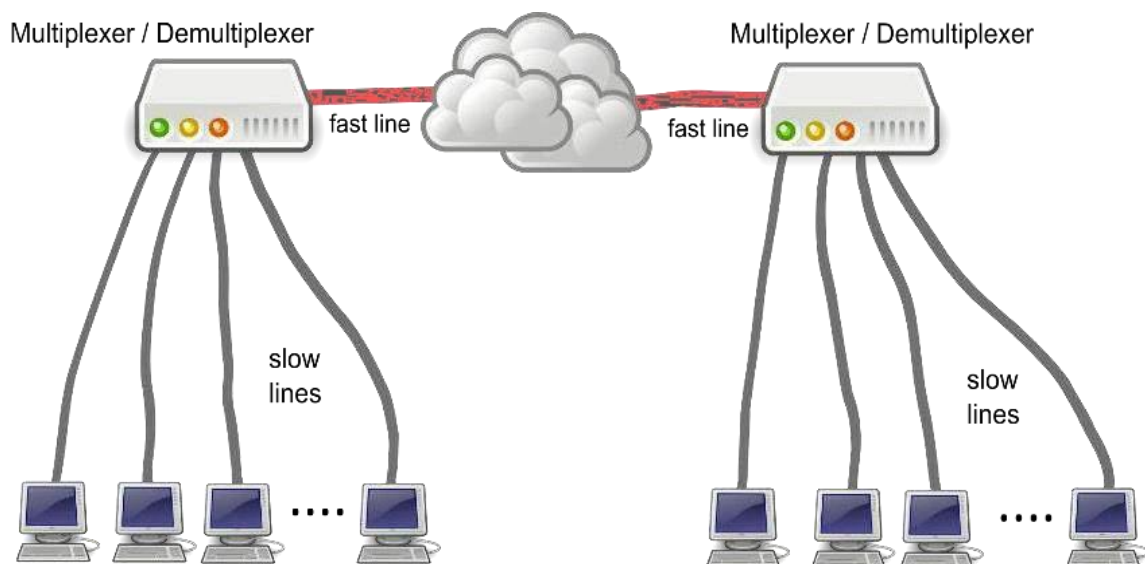


Fig. 1.5.1 Multiplexing/ De-multiplexing systems

1.5.2 Channel Access Methods

A Channel Access Method (or Multiple Access Scheme) determines how multiple users or devices can efficiently share a common transmission medium (the channel) without interfering with each other. These schemes are often based on multiplexing principles.

- One widely used category of channel access schemes is Code Division Multiple Access (CDMA).
- CDMA allows multiple users to share the same channel (both time and frequency) simultaneously. Each user's data is spread across a wider spectrum using a unique spreading code or chip sequence. This unique coding allows the receiver to isolate the intended signal from the composite signal, offering better resistance to interference and increased system capacity compared to simpler methods like Time Division Multiple Access (TDMA) or Frequency Division Multiple Access (FDMA).

Direct Sequence-CDMA (DS-CDMA)

- DS-CDMA is a core technology used in 3rd Generation (3G) mobile phone systems (e.g., UMTS and W-CDMA).
- In DS-CDMA, each information bit or symbol is multiplied by a long, high-rate sequence of pulses called chips (the unique spreading code). This process spreads the signal power over a much wider bandwidth, making it resemble noise to unauthorized receivers.

Frequency Hopping-CDMA (FH-CDMA)

- FH-CDMA (or FHSS - Frequency Hopping Spread Spectrum) involves the rapid and repeated changing of the carrier frequency according to a pseudo-random sequence known only to the transmitter and receiver.
- Bluetooth is a well-known example of a technology that utilizes frequency hopping.

The primary purpose of frequency hopping is to enhance security, reduce interference (both intentional jamming and unintentional interference from other systems), and mitigate multipath fading (signal cancellation caused by reflections). While it can help avoid collisions, its core function is spectrum spreading and interference resilience.

Notes



A large rectangular area with a thin orange border, containing numerous horizontal lines for writing notes.

UNIT 1.6: Mobile Operating Systems

Unit Objectives

By the end of this unit, the participants will be able to:

1. Discuss the features and types of mobile operating systems.
2. Explain the fundamentals of Android application development.

1.6.1 Definition: Operating System

An Operating System (OS) is a critical collection of programs that manages the hardware and software resources of a computer. It serves as an intermediary between the user and the computer hardware.

- The OS performs fundamental tasks such as handling user input (from devices like a keyboard or mouse), managing memory, processing commands, and delivering output (like displaying information on a monitor or printing a document).
- Every general-purpose computer, whether a desktop, laptop, or server, requires an operating system to function.

Mobile Operating Systems Modern mobile phones are also sophisticated computing devices, and thus require a mobile operating system. The mobile OS:

- Manages the device's resources (CPU, memory, battery).
- Runs mobile applications (apps).
- Interprets input from touchscreens, sensors, and physical keys.
- Handles communication processes, including voice calls, text messages, and mobile data networking.

Historical Context Early personal computers primarily used OSs like MS-DOS. These were later superseded by graphical user interfaces (GUIs) found in systems like Microsoft Windows and Apple's macOS (not iOS, which is for mobile devices). Similarly, the evolution of mobile phones from simple "brick" phones to modern smartphones was driven by the introduction of new, feature-rich mobile operating systems.

1.6.2 Symbian OS and User Interface

- **Status Update:** Symbian is now obsolete. It was eventually surpassed by more modern and efficient mobile operating systems like Apple iOS and Google Android, and its development was officially halted. Devices still running Symbian are now considered legacy systems.
- **User Interface:** A User Interface (UI) is the point where human-machine interaction occurs. Effective UI design considers ergonomics (efficiency in the working environment) and psychology to make the device intuitive.

- Symbian utilized various graphical toolkits, such as Series 60. Development later heavily shifted towards the Qt framework (pronounced "cute") to enable faster cross-platform application creation using a Graphical User Interface (GUI). Qt Quick with QML (a declarative language) was recommended for visually rich touchscreen interfaces.

1.6.3 Application Development and Architecture

Development Tools :

- Qt Framework: A cross-platform application development framework. It became the recommended Software Development Kit (SDK) for Symbian, primarily using C++ with Qt extensions. Developers used simulators on a computer to test the applications.
- Symbian C++: The Symbian operating system itself was largely written in C++. Early development was complex and was done using commercial Integrated Development Environments (IDEs) like CodeWarrior, later replaced by Nokia's Eclipse-based IDE called Carbide.c++.
- Eclipse: A popular, open-source IDE (mostly written in Java) that supports multiple programming languages, including C, C++, and Java.
- Other Supported Languages: Symbian also supported application development using languages and runtimes such as Java ME, Python, Flash Lite, Ruby, and Web Runtime.
- Symbian Operating System Architecture

Symbian employed a layered architecture:

1. User Interface Layer
2. Application Services Layer (e.g., Java ME)
3. Operating System Services Layer (Includes Generic OS, Communication, Multimedia/Graphics, and Connectivity services)
4. Base Services Layer (Lowest user-accessible layer; includes File Server, Database Management, and Cryptographic Services)
5. Kernel Services and Hardware Interface Layer (The core)
 - Microkernel Architecture: Symbian utilizes a microkernel. The kernel is the central component of an OS, controlling all operations. A microkernel contains only the minimal software necessary for core OS functions—such as the scheduler (for task management), memory management, and device drivers. Other services (like networking and file systems) are run as user-level processes in the Operating System Services Layer.
 - Networking Subsystem: The large networking and communication subsystem includes key servers with plug-in schemes:

- **ETEL** (Telephony)
- **ESOCK** (Networking)
- **C32** (Serial communication) These subsystems supported short-range links like Bluetooth, IrDA, and USB.
- **Legacy Name:** Symbian was originally called EPOC (Electronic Piece Of Cheese).

Symbian UI Variants/Platforms

Symbian was customized for different devices and markets, resulting in various UI platforms:

- **S60 (Series 60):** Used primarily on Nokia, Samsung, and LG smartphones; supported both keypads and limited touch.
- **Series 80 and Series 90:** Used for business communicators with full keyboards and larger screens.
- **UIQ:** Developed for stylus-based touchscreen devices (used by Sony Ericsson and Motorola).
- **MOAP (Mobile Oriented Applications Platform) and OPP (its successor):** Primarily used in the Japanese market.

The global mobile operating system landscape is dominated by two primary players: Android and Apple's iOS, which together account for over 99% of the worldwide market share. While the market is stable, both systems release significant annual updates focused on enhancing user experience, security, and especially Artificial Intelligence (AI) integration.

1.6.4 Current Mobile Operating Systems

1. Android (Google)

Latest Stable Version: Android 15 (Released in late 2024) Global Market Share: Approximately 70-75% of the global market. Key Characteristics:

- **Open Source (AOSP):** Based on a modified Linux kernel and the open-source Android Open Source Project (AOSP), allowing manufacturers (Samsung, Google, Xiaomi, etc.) to customize it extensively.
- **Fragmentation:** Due to manufacturer and carrier customizations, users often run different versions of the OS, leading to slower adoption of the latest features and security updates across all devices.
- **Customization and Flexibility:** Offers high levels of customization, including custom launchers, widgets, and deep settings control.
- **Ecosystem:** The Google Play Store hosts millions of apps and seamlessly integrates with Google services (Gmail, Drive, Google Assistant).

Highlights of Android 15

- Android 15 heavily focuses on security, privacy, and optimizations for large-screen devices (foldables and tablets).
- **Advanced Theft Protection:** New AI-powered features like Theft Detection Lock automatically lock the phone if it detects a snatch-and-run motion. It also adds new authentication requirements to prevent thieves from easily removing the SIM or turning off Find My Device.

- **Private Space:** A dedicated, hidden, and password-protected area where users can store sensitive apps (like banking or social media) and keep their notifications and data completely separate from the main user profile.
- **Foldable & Tablet Enhancements:** Improvements to multitasking, including the ability to pin and unpin the taskbar and App Pairing to quickly launch two apps side-by-side in split-screen mode.
- **Satellite Connectivity:** Extends satellite support beyond emergency services, allowing carrier-level messaging apps to send and receive text messages without a cellular or Wi-Fi connection.

2. iOS (Apple)

- **Latest Stable Version:** iOS 18 (Released in late 2024) Global Market Share: Approximately 25-30% of the global market. Key Characteristics:
- **Closed Ecosystem:** Used exclusively on Apple's iPhone and iPod Touch devices, which ensures tight integration between hardware and software.
- **Consistency:** The user experience is highly consistent across all supported devices, leading to rapid adoption of new OS versions and a high level of security.
- **Ecosystem:** The App Store maintains a strict vetting process for apps. It integrates seamlessly with other Apple products (iPadOS, macOS, watchOS) through features like Handoff and AirDrop.

Highlights of iOS 18

- iOS 18 focuses on personalization and the introduction of Apple's generative AI features, branded as Apple Intelligence.
- **Enhanced Customization:** For the first time, users can place app icons anywhere on the Home Screen (not restricted to a grid), change app icon color tint, and customize the shortcuts on the Lock Screen.
- **Control Center Redesign:** The Control Center is modular, scrollable, and customizable, allowing users to add or remove controls across multiple pages.
- **Apple Intelligence (AI):** Deep integration of generative models for features like:
- **Writing Tools:** Rewriting, proofreading, and summarising text across Mail, Notes, and other apps.
- **Image Playground:** Creating playful images in Messages and Notes.
- **Siri Overhaul:** A more contextually aware and capable Siri with on-screen awareness and the ability to process more complex, multi-step requests.
- **Photos Redesign:** The largest overhaul of the Photos app ever, with a new single-view layout and AI-powered collections to quickly surface memories.

Emerging Platforms

While Android and iOS are dominant, a few minor players exist, often targeting niche markets:

- **HarmonyOS (Huawei):** Developed by Huawei, it has a significant presence in the Chinese market, effectively replacing Google's proprietary Android services on their newer devices due to geopolitical factors.
- **KaiOS:** A light-weight operating system designed for smart feature phones. It brings smartphone capabilities (like 4G, GPS, and apps like WhatsApp and Google Assistant) to low-cost devices, primarily targeting emerging markets.
- **Custom Android Distributions (e.g., GrapheneOS):** These are security and privacy-focused operating systems built on the Android Open Source Project (AOSP), offering enhanced sandboxing and security features for users who prioritize digital privacy over full feature sets.

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 1.7: Legacy Mobile Operating Systems and Concepts

Unit Objectives

By the end of this unit, the participants will be able to:

1. Discuss the history and eventual discontinuation of the Windows Mobile platform.
2. Explain the key features and evolution of the BlackBerry OS and its hardware innovations.
3. Discuss the significance of the Mobile Information Device Profile (MIDP) in early mobile application development.

1.7.1 Windows Mobile (Discontinued)

Windows Mobile was a family of mobile operating systems developed by Microsoft for smartphones and Personal Digital Assistants (PDAs), often referred to as Pocket PCs. The platform spanned several generations under different names:

- **Origins:** The foundation began with Windows CE (Compact Embedded) in 1996.
- **Evolution:** It was branded as Pocket PC in 2000 and then officially renamed Windows Mobile in 2003.
- **Status Update:** The Windows Mobile platform was superseded by Windows Phone 7 in 2010, which was a complete re-write and incompatible with the older OS. Microsoft effectively exited the mobile OS market with the eventual failure of the Windows Phone and subsequent Windows 10 Mobile initiatives. Windows Mobile is now considered obsolete and is no longer supported.

Standard Features (During Active Life):

- **Core Applications:** Included Internet Explorer Mobile (the default browser), Windows Media Player, and mobile versions of Microsoft Office applications (Word, Excel, PowerPoint, and Outlook).
- **Networking:** Supported features like Internet Connection Sharing (tethering) via USB or Bluetooth, and Virtual Private Networking (VPN) over protocols like PPTP.

Virtual Private Networking (VPN)

A VPN is a technology that creates a secure, encrypted connection (a "tunnel") over a less secure network, such as the public internet. It allows a user to send and receive data securely, as if their computing device were directly connected to a private network.

- **Mechanism:** VPNs are established using protocols (like PPTP, L2TP, IKEv2, or the more modern OpenVPN/WireGuard) that tunnel data traffic through the network.
- **Classification:** VPNs can be classified based on the protocols used, where the secure connection terminates (e.g., user device or a corporate network firewall), the security levels employed, and whether they offer remote access (user-to-network) or site-to-site (network-to-network) connectivity.

Point-to-Point Tunneling Protocol (PPTP)

PPTP is one of the oldest and most basic networking protocols used to implement VPNs. While it was widely supported by older Windows Mobile devices and desktop versions, it is now considered largely insecure and outdated due to known vulnerabilities.

Windows Mobile Version History (Key Releases)

Year	Version Name	Notes
1996	Windows CE	The initial embedded kernel.
2000	Pocket PC 2000	First branded release for handhelds.
2003	Windows Mobile 2003	Official rebranding; supported touchscreens and non-touch phones.
22005	Windows Mobile 5.0	Significant updates to the operating system and platform.
2007	Windows Mobile 6.0	Focused on better integration with desktop Microsoft Office applications.
2009	Windows Mobile 6.5	The last version of the traditional Windows Mobile line, featuring a more finger-friendly interface.

1.7.2 Hardware and Platform Variations

During its peak, Microsoft tailored its mobile OS versions to match different hardware form factors:

- Windows Mobile Professional: Developed for smartphones and Pocket PCs equipped with touchscreens.
- Windows Mobile Standard: Developed for non-touchscreen devices (often called "smartphones" at the time) that primarily relied on physical keypads and D-pads for navigation.
- Windows Mobile Classic: The version designated for Personal Digital Assistants (PDAs) and legacy Pocket PC devices.

1.7.3 BlackBerry OS (Discontinued)

BlackBerry OS was a proprietary mobile operating system developed by BlackBerry Limited (formerly Research In Motion - RIM) for its line of smartphones.

Key Features: It was renowned for its robust corporate email integration (via the BlackBerry Enterprise Server - BES), excellent multitasking capabilities, and strong focus on security and instant communication (BBM).

Hardware Innovation: BlackBerry devices were famous for their specialized input methods, including the signature full QWERTY keyboard, and navigation tools like the track wheel, trackball, and later the optical trackpad. These were considered best-in-class for rapid communication.

Status Update: BlackBerry OS was discontinued. The company later released BlackBerry 10 (a new, QNX-based OS) and then shifted to producing Android devices before fully exiting the hardware/OS business. BlackBerry OS is now obsolete and no longer receives official support.

Email Support: It provided strong support for corporate email and data synchronization with major platforms like Microsoft Exchange, Notes, Calendar, and Contacts.

1.7.2 Mobile Information Device Profile (MIDP)

MIDP stands for Mobile Information Device Profile.

- **Definition:** It was a specification published by the Java Community Process designed to allow Java applications (called MIDlets) to run on mobile phones and personal digital assistants (PDAs).
- **Framework:** MIDP was a core part of the Java Micro Edition (Java ME) framework. It defined the APIs (Application Programming Interfaces) for user interface, persistent storage, networking, and other lower-level functions necessary for applications to execute in the resource-constrained environment of early mobile devices.
- **Relevance:** It was crucial for developing the first generation of mobile games and simple applications before the rise of modern smartphone operating systems.

Software Updates

Mobile operating systems, including these older platforms and modern ones like Android and iOS, can be updated remotely by wireless carriers or manufacturers using the Over-The-Air Software Loading (OTASL) service, now commonly known simply as OTA updates.

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 1.8: Android Operating System and Version History

Unit Objectives

By the end of this unit, the participants will be able to:

1. Analyze and differentiate the features, architecture, and historical context of major Android versions.

1.8.1 Android OS Fundamentals

Android is a Linux-based, open-source operating system developed by Google and the Open Handset Alliance. It is the world's most widely used mobile OS, initially designed for touchscreen devices but now serving a massive ecosystem, including smart TVs, cars, smartwatches, and IoT devices.

- **Open Source Core:** The heart of the system is the Android Open Source Project (AOSP), released under a permissive license. This openness allows hundreds of manufacturers (Samsung, Xiaomi, etc.) to customize the software heavily for their specific hardware and markets, creating diverse user experiences.
- **Source Code Defined:** The Source Code consists of human-readable programming instructions (primarily in C, C++, Java, and Kotlin). It must be translated by a compiler into low-level machine code (binary instructions) before the computer's processor can execute it to generate the desired output.

1.8.2 Key Features and User Interface (UI)

Android's interface is rooted in Direct Manipulation, relying on gestures like tapping, swiping, pinching, and dragging for interaction.

- **Input and Interaction:**
 - **Virtual Keyboard:** The standard input method for text.
 - **Peripherals:** Supports external devices like keyboards, mice, and game controllers via Bluetooth or USB.
- **Sensors for Context:** Android devices utilize sophisticated sensors for enhanced application functionality, allowing the device to understand its environment and movement:
 - **Accelerometer and Gyroscope:** Essential for motion-based input (e.g., tilting the device to steer in a racing game).
 - **Proximity and Ambient Light:** Used to manage screen brightness and turn off the display during phone calls.
 - **Barometer, Magnetometer (Compass), and GPS:** Provide data for location services, navigation, and weather applications.
- **Home Screen and Widgets:** The home screen serves as the user's primary hub, featuring application icons and widgets—small, dynamic, and auto-updating application snippets (e.g., a music player controller or a scrolling news feed) that live directly on the home page.

- **Notification System:** A core strength of Android. The Status Bar can be swiped down to reveal the Notification Shade, which aggregates all alerts (messages, missed calls, updates). These are easily managed, allowing users to take action directly on a notification (e.g., replying to a text) or swipe away unwanted alerts individually or in batches. It also hosts the Quick Settings toggles for Wi-Fi, Bluetooth, etc.

1.8.3 Application Development and Ecosystem

- **App Development:** Android apps are built using the official Android Software Development Kit (SDK), primarily programmed in Java or, increasingly, Kotlin.
- **IDEs and Tools:** The current official IDE is Android Studio (which replaced the older Eclipse IDE). The SDK provides essential tools like a debugger, software libraries, and an emulator (a software replica of an Android device) for rigorous testing across various screen sizes and hardware specifications.
- **Distribution:** Apps are packaged as Android Application Package (APK) files.
 - **Google Play Store:** The official, curated platform for distribution, offering security screening, easy installation, and guaranteed updates. Many apps are distributed under open-source licenses, while others are paid.
 - **Third-Party Sideload:** Users can install apps from outside the Play Store via APKs. This offers flexibility but carries significant risks of installing malware or bug-ridden, unsupported software. Using the official store for installation and updates is strongly recommended for security.

1.8.3 Application Development and Ecosystem

- **App Development:** Android apps are built using the official Android Software Development Kit (SDK), primarily programmed in Java or, increasingly, Kotlin.
- **IDEs and Tools:** The current official IDE is Android Studio (which replaced the older Eclipse IDE). The SDK provides essential tools like a debugger, software libraries, and an emulator (a software replica of an Android device) for rigorous testing across various screen sizes and hardware specifications.
- **Distribution:** Apps are packaged as Android Application Package (APK) files.
 - **Google Play Store:** The official, curated platform for distribution, offering security screening, easy installation, and guaranteed updates. Many apps are distributed under open-source licenses, while others are paid.
 - **Third-Party Sideload:** Users can install apps from outside the Play Store via APKs. This offers flexibility but carries significant risks of installing malware or bug-ridden, unsupported software. Using the official store for installation and updates is strongly recommended for security.

1.8.4 Hardware Architecture

- **ARM Dominance:** The vast majority of Android devices utilize processors designed by ARM (Advanced RISC Machine). The Reduced Instruction Set Computing (RISC) architecture is key to its success in mobile:
- **Efficiency:** RISC processors require fewer transistors and simpler instruction sets, leading to significantly lower power consumption (critical for battery life) and less heat generation than the desktop-focused CISC architecture (used by Intel/AMD).
- **64-bit Transition:** Android fully adopted the 64-bit instruction set (ARMv8-A) starting with Android 5.0 "Lollipop." This shift was crucial for supporting modern mobile CPUs, addressing larger amounts of RAM (over 4 GB), and boosting security and performance.
- **Multi-Platform Support:** While ARM is standard, Android also supports x86 architectures (used by some Intel and AMD processors) and historically supported MIPS, ensuring compatibility across a broader range of specialized devices like industrial handhelds or smart displays.

1.8.5 Android Version History (The Dessert Era)

Since 2019, Google stopped using dessert names for major public releases, adopting simple numbering starting with Android 10, but the alphabetical dessert names remain useful for historical context.

Codename	Version Range	Release Year	Key Innovations and Features
Cupcake	1.5	2009	First on-screen virtual keyboard; support for third-party widgets; video recording and playback.
Donut	1.6	2009	Quick Search Box (universal search); support for different screen resolutions (vital for early fragmentation); CDMA network support.
Eclair	2.0 – 2.1	2009	Multi-touch support; global search within SMS/MMS; turn-by-turn navigation (Google Maps); live wallpapers.
Froyo	2.2 – 2.2.3	2010	Speed, memory, and performance optimizations (V8 JIT compiler); USB Tethering and Wi-Fi Hotspot; Adobe Flash support in the browser.
Gingerbread	2.3 – 2.3.7	2010	Refined UI design; NFC (Near Field Communication) support; improved power management; native support for multiple cameras.
Honeycomb	3.0 – 3.2.6	2011	Tablet-only OS (later merged with phones); new "Holo" UI theme; virtual navigation buttons; optimized multitasking.
Ice Cream Sandwich	4.0 – 4.0.4	2011	Unification of phone and tablet interfaces; introduced "Holo" design language across all devices; Face Unlock; swipe gestures for dismissing notifications and recent apps.

Jelly Bean	4.1 – 4.3.1	2012	Project Butter (major focus on performance and frame rate smoothness); expandable and actionable notifications; introduction of Google Now (intelligent personal assistant).
KitKat	4.4 – 4.4.4	2013	Optimization for entry-level devices (low RAM usage); "Immersive Mode" for full-screen apps; native printing framework; translucent navigation bars.
Lollipop	5.0 – 5.1.1	2014	Major design overhaul with Material Design (paper-like interface with shadows and motion); lock screen notifications; multi-user support for phones; native 64-bit support.
Marshmallow	6.0 – 6.0.1	2015	Runtime Permissions (apps ask for permission when needed); Doze Mode (deep sleep for power saving); native Fingerprint sensor support; USB-C support.
Nougat	7.0 – 7.1.2	2016	Split-screen multitasking; Vulkan API for improved 3D graphics; "Direct Reply" for notifications; seamless system updates.
Oreo	8.0 – 8.1	2017	Notification Channels (fine-grained control over alerts); Picture-in-Picture (PiP) mode; Autofill Framework; Project Treble for modular system updates.
Pie	9.0	2018	Gesture Navigation (introduced modern swiping); Digital Wellbeing controls; Adaptive Battery (uses machine learning to prioritize app resources).
Android 10	10.0	2019	First numeric version; system-wide Dark Mode; improved sharing shortcuts; dedicated permissions for location access.
Android 11	11.0	2020	Conversation Notifications (grouping chats at the top); built-in Screen Recorder; enhanced privacy controls for location and microphone.
Android 12	12.0	2021	Material You (dynamic color theming based on wallpaper); redesigned widgets; new privacy dashboard and indicators for camera/mic use.
Android 13	13.0	2022	Per-app language settings; refined Material You colors; expanded theme support for app icons; enhanced privacy for media access.
Android 14	14.0	2023	Improved battery life; new security protections (blocking installation of older apps); enhanced customization options for the lock screen.
Android 15	15.0	2024	Private Space; AI-powered Theft Detection Lock; satellite connectivity for messaging; App Pairing for improved multitasking on foldables/tablets.

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 1.9: Android Device Configuration and Development Requirements

Unit Objectives



By the end of this unit, the participants will be able to:

1. Explain the architecture and components of the Android operating system.
2. Describe the basic configuration and setup of Android handsets.

1.9.1 Android Handset Configuration (Historical Baseline)

Android is a Linux-based, open-source operating system developed by Google to support a wide spectrum of device configurations. The following requirements represent the historical minimums for early Android releases (around versions 1.x through 4.x, or the "pre-modern" era). Modern Android (currently Android 15) requires significantly more powerful hardware.

1. Chipset (Processor Architecture)

- **Definition:** A chipset is a collection of circuits (or integrated circuits) that coordinate the flow of data between the various components of a computer or mobile system (CPU, memory, storage, peripherals).
- **Architecture:** Android's initial and primary architecture is ARM (Advanced RISC Machine).
 - **RISC Advantage:** RISC stands for Reduced Instruction Set Computing. This architecture uses simpler, faster instructions, allowing processors to be built with fewer transistors. This design reduces cost, heat generation, power consumption, and the overall physical size of the processor, making it ideal for mobile devices.

2. Memory (RAM)

- **Historical Minimum:** The absolute minimum requirement for very early, low-density screen devices (pre-KitKat) was around $\mathbf{128\text{ MB}}$ RAM (Random Access Memory).
- **Modern Reality:** Modern versions of Android (currently Android 15) require a minimum of 2 GB to 4 GB of RAM for basic, smooth operation, with high-end devices exceeding 12 GB or 16 GB .

3. Storage

- **Historical Need:** Early Android devices had very limited internal flash storage (e.g., $\mathbf{256\text{ MB}}$), necessitating the use of external storage like Mini-SD or Micro-SD cards to store user data and system updates.
- **Modern Reality:** Today, most device storage is internal flash memory (UFS or eMMC), with a recommended minimum of 64 GB for the OS and applications. External SD card support is now an optional feature.

4. Primary Display

- **Historical Baseline:**
 - Early Android targeted displays with \mathbf{HVGA} (Half-size Video Graphics Array, 480×320 pixels) resolution.

- An acceptable minimum visual experience was a \mathbf{QVGA} (Quarter Video Graphics Array, 320×240 pixels) resolution using TFT LCD (Thin Film Transistor Liquid Crystal Display) technology with 16-bit color or better.
- Minimum physical size was around $\mathbf{2.8\text{ inches}}$ with a touch interface.

Modern Reality: Modern smartphones use much higher resolutions, typically HD (1280×720), Full HD (1920×1080), or higher, using OLED or advanced LCD technology.

5. Navigation and Control Keys

- **Historical Requirement:** Basic Android devices were required to have dedicated physical keys for essential functions: Power, Volume Control (up/down), and often a dedicated Camera key.
- **Modern Reality:** While Power and Volume remain physical, the Camera key is largely replaced by a screen icon. Mandatory physical navigation buttons (Home, Back, Menu) were deprecated in favor of on-screen virtual keys and modern gesture navigation.

6. Camera

- **Historical Requirement:** The camera was initially an optional feature, with a basic $\mathbf{2\text{ MP}}$ (Megapixel) sensor being the benchmark if included.
- **Modern Reality:** A camera is now a standard and expected feature, with modern devices featuring multiple rear and front lenses (main, ultra-wide, telephoto) often exceeding $\mathbf{50\text{ MP}}$.

7. Bluetooth

- **Purpose:** Bluetooth is the standard wireless technology for creating a Personal Area Network (PAN) for short-range data transfer and connecting peripherals (headphones, keyboards, etc.).
- **Historical Requirement:** Early Android required $\mathbf{Bluetooth\text{ 1.2}}$ or $\mathbf{Bluetooth\text{ 2.0}}$.
- **Modern Reality:** Modern Android supports current standards, including Bluetooth $\mathbf{5.x}$ and LE Audio, for improved speed, range, and power efficiency.

8. Universal Serial Bus (USB)

- **Function:** USB enables high-speed data transfer between the Android device and a Personal Computer (PC). It is also essential for debugging (using $\text{Android Debug Bridge}$ or ADB) and charging.
- **Historical Requirement:** The standard interface used was the Standard mini-B USB interface.
- **Modern Reality:** The global standard is now the USB Type-C connector, supporting features like USB $\mathbf{3.1}$ / $\mathbf{3.2}$ and Power Delivery for fast charging.

1.9.2 Android Studio System Requirements

Google has significantly increased the demand for development tools, particularly with the transition to 64-bit systems and the use of the powerful IntelliJ IDEA platform. The following table provides the current recommended requirements for development as of 2024/2025.

Component	Minimum Requirement (for basic development)	Recommended Requirement (for modern development with Android 15 and Emulators)
OS Version	Microsoft Windows 10/11 (64-bit), macOS 11.0 (Big Sur) or higher, or a modern 64-bit Linux distribution (e.g., Ubuntu 20.04/22.04)	Windows 10/11 (64-bit), macOS 14 (Sonoma) or higher
CPU	Any recent 64-bit processor with support for Intel VT-x , Intel EM64T (Intel) or AMD V (AMD) for emulator acceleration.	8^{th} Generation Intel Core i5/i7 or newer, or AMD Ryzen 5/7 or newer (or Apple Silicon M series)
RAM	8^{GB} minimum	16^{GB} or more
Disk Space	8^{GB} of available disk space, plus space for IDE (5^{GB})	50^{GB} SSD (Solid State Drive) minimum for OS, IDE, and development tools
Android SDK/Tools	At least 10^{GB} for the Android SDK, system images, NDK , and caches.	20^{GB} to 30^{GB} SSD space recommended
JDK Version	Java Development Kit (JDK) 17 or higher.	$\text{JDK } 21$ (latest LTS version)
Screen Resolution	1280×800 minimum screen resolution	1920×1080 or higher recommended

Key Modern Change: The requirement for RAM has drastically increased to ensure smooth performance when running the IDE , multiple virtual Android Emulators, and the compiler simultaneously. Using an SSD is now considered mandatory for a productive development experience.

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 1.10: Android Studio Installation and Setup

Unit Objectives



By the end of this unit, the participants will be able to:

1. Set up the required Java Development Kit (JDK) environment.
2. Successfully install and configure Android Studio on Windows, Linux, and macOS.
3. Use the SDK Manager to download and manage the essential Android SDK packages and tools.

1.10.1 Java Development Kit (JDK) Setup (Prerequisite)

Android Studio requires the Java Development Kit (JDK) to compile Android apps, although modern Android Studio versions bundle a compatible JDK . However, explicitly verifying or setting up a compatible version is still a crucial first step.

- **Current Requirement:** Android Studio officially uses and bundles a version of OpenJDK (an open-source implementation of the Java Platform). For current stable development, the recommended version is $\text{JDK}\text{ 17}$ or newer ($\text{JDK}\text{ 21}$).
- **Installation:**
 - Go to the official Oracle Java website or the OpenJDK distribution page (e.g., Adoptium or $\text{Microsoft's OpenJDK build}$).
 - Download the latest LTS (Long-Term Support) version of the JDK compatible with your operating system (Windows, macOS, or Linux).
 - Run the installer and follow the prompts. The installer usually handles setting up the necessary environment variables.

1.10.2 Installing Android Studio (Windows, macOS, and Linux)

Android Studio is the official Integrated Development Environment (IDE) provided by Google.

Download: Always download the latest stable version of Android Studio from the official Google Developers page: <https://developer.android.com/studio>

Installation on Windows

- **Run Installer:** Locate the downloaded .exe file and run it to start the $\text{Android Studio Setup Wizard}$.
- **Configuration:** Click Next through the setup prompts. Ensure that both Android Studio and Android Virtual Device (AVD) are selected for installation.
- **Location:** Choose an installation location (the default is usually fine).
- **Install:** Click Install. Once finished, the setup may prompt you to Start Android Studio.
- **Settings Import:** If this is your first install, select the option to "Do not import settings." If upgrading, you may import settings from a previous version.

- **Setup Wizard:** The `$\text{Setup}\text{ Wizard}` will guide you through downloading essential components. Select the Standard installation type for beginners.
- **Finalize:** The wizard will download components like the latest `$\text{SDK}\text{ Platform}` and `$\text{Platform}\text{ Tools}`. Click Finish to complete the setup.
- Installation on Linux
 - **Extract:** Download the .zip file. Extract it to your preferred installation location, often in the user's home directory or a system-wide location like /opt.
 - **Example using the terminal:** `sudo unzip android-studio-ide-xxxx-linux.zip -d /opt`
 - Launch Studio: Navigate to the android-studio/bin directory within the installation path. Launch the executable script:
 - `cd /opt/android-studio/bin`
 - `./studio.sh`
 - Setup Wizard: Follow the `$\text{Setup}\text{ Wizard}` instructions, similar to the Windows installation, to download and extract required `$\text{SDK}` components.
 - Optional (Recommended): To easily launch Android Studio from any terminal location, add the /opt/android-studio/bin directory to your system's `$\text{PATH}` environment variable.
 - Note for 32-bit Libraries: If you are running a 64-bit version of Debian-based Linux (e.g., Ubuntu), you may need to install specific 32-bit libraries for the `$\text{Android}\text{ Emulator}` to function correctly: `sudo apt-get install lib32z1 lib32ncurses5 lib32bz2-1.0 lib32stdc++6`
- Installation on macOS
 - Run DMG: Download the .dmg file and open it.
 - Install: Drag the Android Studio application icon into your Applications folder.
 - Launch & Setup: Open `$\text{Android}\text{ Studio}` from the Applications folder. The `$\text{Setup}\text{ Wizard}` will launch automatically, prompting you to download the necessary `$\text{SDK}` components, following the standard installation path.

1.10.3 Installing and Managing the Software Development Kit (SDK)

The `$\text{Android}\text{ SDK}` is essential; it contains the libraries, documentation, tools, and system images required to build and test apps for specific Android versions. All `$\text{SDK}` management is done through the SDK Manager within `$\text{Android}\text{ Studio}`.

- **Launch SDK Manager:** In `Android Studio`, click the SDK Manager icon (often a small box with a downward-pointing arrow and a cog, located in the toolbar) or go to Tools \rightarrow SDK Manager.
- **SDK Platforms (Target Android Versions):** Select the SDK Platforms tab.
 - Choose the latest stable Android Platform (e.g., Android 15.0) to target the newest devices.
 - Select any older platforms you need to support (e.g., Android 10.0 or 11.0 to ensure compatibility with older device market share).
- **SDK Tools (Core Components):** Select the SDK Tools tab. Ensure the following are installed:
 - Android SDK Build-Tools (latest version)
 - Android SDK Platform-Tools (`ADB`, `fastboot`, etc.)
 - Android SDK Tools (legacy, usually superseded by `Platform-Tools`)
 - Android Emulator
- **API Support and Repository (Extras):** Open the SDK Tools tab and expand the "Extras" or "Support" sections (depending on the `Studio` version).
- **Android Support Repository / Google Repository:** Essential for development using modern Android Architecture Components and Google services.
- **Google Play Services:** Needed if your app uses Google APIs for features like maps, location, or cloud messaging (`Firebase`).
- **Install Packages:**
 - Check the boxes for all required packages.
 - Click "Apply" or "Install X packages."
 - A dialog will appear requiring you to accept the license agreement for all selected packages. Accept the agreement and click Install to download the files.

Completion: Once the installation finishes, the system is fully configured to create a new Android project and begin development.

Exercise

Short Questions:

1. Define telecommunication and name its two main transmission media.
2. Differentiate between analog and digital signals with suitable examples.
3. Explain the role of a transmitter, transmission medium, and receiver in a telecommunication system.
4. What is multiplexing? Name and explain the device used for multiplexing.
5. Briefly describe how CDMA allows multiple users to share the same communication channel.
6. List any three major generations of mobile networks and their key characteristics.
7. What is the function of a mobile operating system?
8. State any two reasons why Symbian OS became obsolete.

Multiple Choice Questions:

1. Which of the following is not a physical transmission medium?
 - a) Optical fiber
 - b) Copper wire
 - c) Electromagnetic waves
 - d) Coaxial cable
2. The device that combines multiple signals into one for transmission is called:
 - a) Router
 - b) Multiplexer
 - c) Demultiplexer
 - d) Amplifier
3. The first commercially successful telephone was patented in which year?
 - a) 1844
 - b) 1876
 - c) 1890
 - d) 1905
4. Which of the following statements about analog and digital signals is true?
 - a) Analog signals are discrete; digital signals are continuous.
 - b) Digital signals are continuous; analog signals are discrete.
 - c) Analog signals are continuous; digital signals are discrete.
 - d) Both are continuous.
5. Which generation of mobile technology first introduced packet-switched data?
 - a) 1G
 - b) 2G
 - c) 3G
 - d) 4G
6. CDMA works by assigning each user a unique _____.
 - a) Frequency band
 - b) Time slot
 - c) Spreading code
 - d) Channel number

7. The main function of an Operating System is to:
 - a) Manage hardware and software resources
 - b) Increase battery life
 - c) Connect devices to Wi-Fi
 - d) Compress data files

8. Symbian OS was primarily written in which programming language?
 - a) Java
 - b) Python
 - c) C++
 - d) Kotlin

9. The Android operating system is based on which kernel?
 - a) UNIX
 - b) Linux
 - c) DOS
 - d) Windows

10. The current official IDE for Android development is:
 - a) Eclipse
 - b) Android Studio
 - c) Visual Studio Code
 - d) CodeWarrior

Fill in the Blanks:

1. Telecommunication involves the electronic transmission of _____ over long distances.
2. The process of separating combined signals at the receiver is known as _____.
3. In packet switching, data is divided into small units called _____.
4. A _____ is used to boost or regenerate signals over long transmission distances.
5. The global body that develops mobile communication standards like LTE and 5G is _____.
6. In DS-CDMA, each information bit is multiplied by a unique _____ sequence.
7. The two dominant mobile operating systems today are _____ and _____.
8. Android applications are packaged in files with the extension _____.
9. The Reduced Instruction Set Computing architecture used in Android devices is known as _____.
10. The software that allows developers to build Android apps is called the _____.

Application-Based Questions

1. Suppose a signal transmitted over 100 km weakens due to noise. What device would you use to restore it, and why?
2. If two users are transmitting simultaneously using CDMA, how does the receiver identify the correct signal?
3. A developer is setting up Android Studio on a Windows 11 system with 8 GB RAM. What additional hardware upgrades would you recommend for optimal performance?
4. Discuss how Android's open-source model contributes to the diversity of mobile devices in the market.

Notes



Lined area for taking notes, consisting of multiple horizontal lines.



2. Setting up Android Framework/ Development Environment and Creating user Interface



Unit 2.1 - Creating Simple Android Project in Android Studio

Unit 2.2 - Running the Project in Android Studio

Unit 2.3 - Creating and Running a Simple User Interface



Key Learning Outcomes



By the end of this module, the participants will be able to:

1. Explain the principles of radio access technology (4G/5G) and their deployment frameworks, highlighting key differences and implementation strategies.
2. Discuss the fifth-generation (5G) access domain, including 3GPP specifications and standards across L1, L2, and L3 layers, and their significance in modern telecommunications.
3. Demonstrate coordination with cross-functional teams to translate high-level architecture into deployment deliverables.
4. Explain the role of cloud technologies, Open Edge Servers, and xHaul deployments within cloud environments and their impact on 5G network efficiency.
5. Show how to implement orchestration between teams, NFVI, and cloud-native network functions to optimize site deployment.
6. Show how to review and interpret 3GPP standards, project budgets, architectural blueprints, and client-specific design documents.
7. Demonstrate coordination with cross-functional teams to translate high-level architecture into deployment deliverables.
8. Describe the key concepts of VoLTE, VoWiFi, Virtualized RAN (vRAN), O-RAN, and Management and Orchestration (MANO), and their roles in advanced mobile networks.
9. Demonstrate evaluation of MIMO antenna radiation patterns and implementation of network slicing and NFV strategies.
10. Elucidate the message flows, parameters, and signaling procedures used in 5G networking, and their importance in maintaining seamless communication.
11. Demonstrate validation of utilities, such as command centers, alarm managers, PM reports, and alarm correlation tools.
12. Explain system integration principles, focusing on API management, virtualized network function (VNF) compatibility, and interoperability testing in telecom infrastructures.
13. Show how to assess and validate MIMO antenna parameters, including diversity gain, MIMO capacity, and beamforming requirements.
14. Demonstrate evaluation of MIMO antenna radiation patterns for performance optimization.
15. Describe network slicing and its role in providing flexible, programmable, and efficient network management in 5G networks.
16. Demonstrate implementation of network slicing and NFV strategies to ensure optimal utilization of network resources.
17. Describe the equipment safety procedures, compliance guidelines, and industry best practices for hardware deployment in a telecom environment.
18. Demonstrate how to verify the availability of passive equipment (battery banks, power plants, antennas, feeder cables, mounting accessories, etc.) ensuring operational readiness.
19. Show how to ensure readiness of active equipment including gNodeB, fiber transmission networks, microwave links, and edge computing units for deployment.
20. Discuss troubleshooting methodologies for common hardware failures, network configuration issues, and software bugs, including best practices for rapid issue resolution.

21. Demonstrate installation, configuration, and commissioning of equipment at designated locations while ensuring safety compliance.
22. Show how to conduct pre-activation checks to validate equipment readiness and network synchronization.
23. Explain lifecycle management processes and their relevance in maintaining and upgrading 5G networks for optimal performance.
24. Demonstrate impact assessment and risk analysis of solution lifecycle activities, including revisions and system upgrades.
25. Explain the concept and purpose of Proof of Concepts (PoCs) in validating new 5G solutions.
26. Show how to execute Proof of Concepts (PoC) and present findings to stakeholders.
27. Explain methods of analyzing signal strength and understanding antenna orientation parameters (tilt, zenith, azimuth).
28. Demonstrate signal strength analysis and adjustment of antenna tilt, zenith, and azimuth angles for optimal coverage.
29. Explain system integration and orchestration concepts across teams, NFVI, and cloud-native environments.
30. Show how to implement orchestration between teams, NFVI, and cloud-native network functions to optimize site deployment and productivity.
31. Determine the appropriate escalation processes and incident management frameworks for reporting system failures, security threats, and environmental hazards.
32. Demonstrate reporting and handling of emergency incidents such as passive equipment failures, fire, and power loss in accordance with safety norms.
33. Explain the process of software testing and automation in 5G environments, and mapping of backhaul network structures with site configurations.
34. Show how to conduct pre-deployment software testing using automated scripts and verify backhaul network integration for 5G sites.

UNIT 2.1: Install 5G NR Site Hardware Equipment

Unit Objectives



By the end of this unit, the participants will be able to:

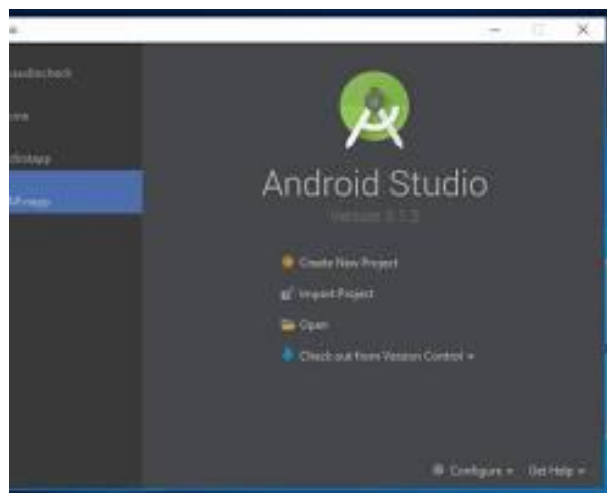
1. Follow guided steps to create a basic Android project using available tools.
2. Identify the key components of a Class definition and describe their basic functions.

2.1.1 Creating a Simple Android Project

Creating a simple Android project in Android Studio is a straightforward process. Here are the steps, complete with images for visual reference:

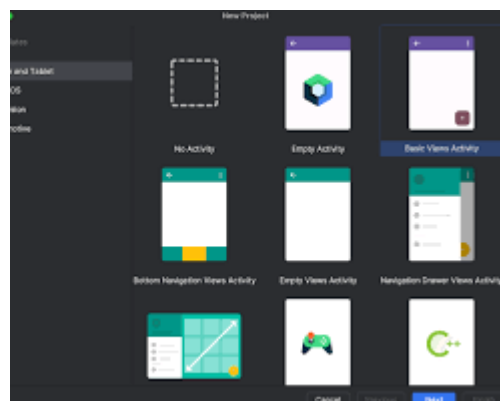
Step 1: Start a New Project

- Open Android Studio.
- In the welcome screen, click "New Project".



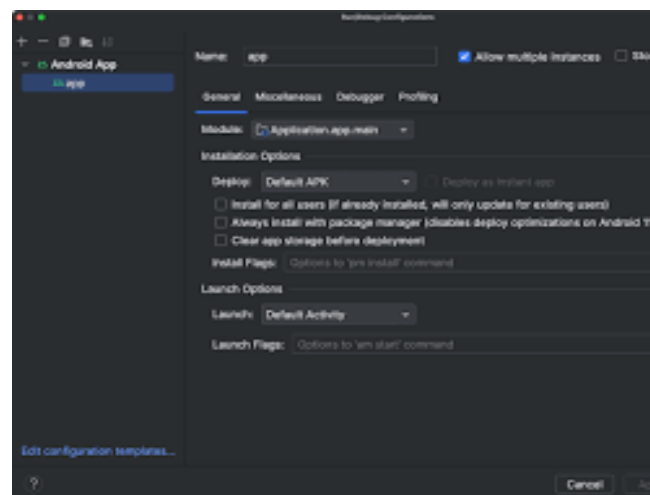
Step 2: Choose a Project Template

- The next screen will prompt you to choose a template for your project. Templates provide a starting point for your app's user interface.
- For a simple project, select the "Empty Activity" template. This gives you a minimal starting structure with a single screen (Activity).
- Click "Next".



Step 3: Configure Your Project

- On the "Configure your new project" screen, you'll need to set a few details:
- Name: Enter a name for your application (e.g., "MyFirstApp"). This is the name displayed to users.
- Package name: This is automatically generated based on the name you entered. It must be unique across all Android apps. A standard format is com.example.yourprojectname. You typically don't need to change this unless you're preparing for publishing.
- Save location: Choose where you want the project files to be saved on your computer.
- Language: Select either Kotlin (the preferred modern language for Android development) or Java.
- Minimum SDK: Choose the minimum Android version your app will support. Selecting a lower API level means your app can run on more devices, but you'll have fewer modern features available. For a beginner project, the default suggested value is often fine.
- Click "Finish".

**Step 4: Wait for Gradle Build**

Android Studio will now create the project structure and perform an initial Gradle build. Gradle is the system that manages your project's dependencies and builds the app.

- Wait for the process to complete (you'll usually see progress bars at the bottom of the screen). This may take a few moments.
- Once the build is finished, your simple Android project is ready. You will see two main files open in the editor:
- MainActivity.kt (or .java): This is the main code file for your app's logic.
- activity_main.xml: This is the layout file that defines the user interface (what the user sees) for your main screen.

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 2.2: Running the Project in Android Studio

Unit Objectives

By the end of this unit, the participants will be able to:

1. Follow instructions to enable developer options on an Android device.
2. Demonstrate how to run an application using an Android Emulator under supervision.
3. Describe the basic procedure to install and run an application on a real Android device.

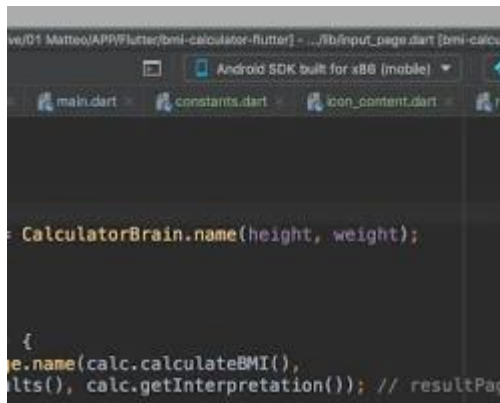
2.2.1 Running the Project

Running your Android project involves either setting up a Virtual Device (Emulator) or connecting a physical Android device to your computer.

Following are the steps for running your project in Android Studio, using an Emulator:

Step 1: Check the Run Configuration

- Look at the toolbar in Android Studio, near the top-center.
- You will see a dropdown menu that displays the currently selected device or the message "No devices."
- Next to the device selector, there is a green Run button (a triangle icon).

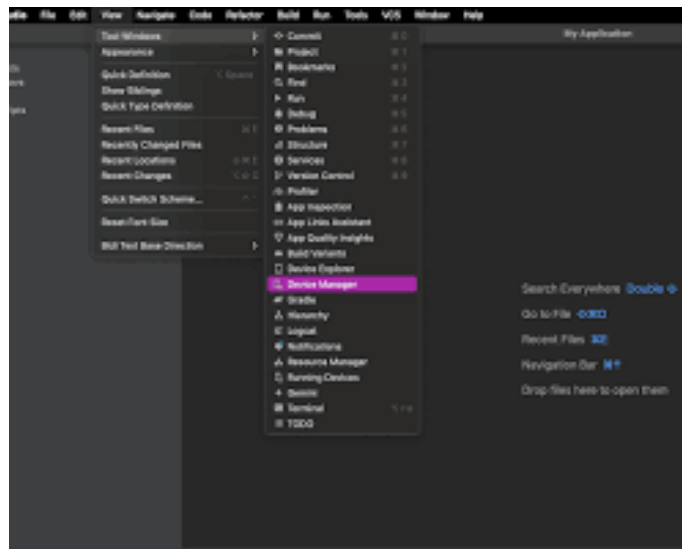


Step 2: Create a New Virtual Device (If needed)

If you don't have a device listed, you need to create an Android Virtual Device (AVD), which is an emulator.

- Click the device dropdown menu in the toolbar and select "Device Manager" (or go to Tools > Device Manager).
- In the Device Manager panel, click the "Create device" button.
- Select Hardware: Choose a device definition (e.g., Pixel 6 or Pixel 7). This determines the screen size and resolution of your virtual phone. Click "Next".

- **Select System Image:** Choose the Android version (System Image) you want to run on the emulator. You'll likely need to download the image first (look for the "Download" link next to the image name, like Tiramisu or UpsideDownCake).
- Click "Next" after selecting an image.
- **Verify Configuration:** Give your AVD a name and confirm the settings (orientation, memory, etc.). Click "Finish".



Step 3: Run the App

- Ensure your newly created AVD is selected in the device dropdown menu on the toolbar. If you don't see it, try restarting Android Studio.
- Click the green "Run" button (▶) on the toolbar.

Step 4: View the Result

- Android Studio will begin the Gradle build process again. Watch the "Build" output window at the bottom of the screen.
- Once the build is complete, the Emulator will start up (if it wasn't already running).
- Your app will be automatically installed and launched on the virtual device. You should see your simple app, which usually displays "Hello World!" by default.

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 2.3: Creating and Running a Simple User Interface

Unit Objectives

By the end of this unit, the participants will be able to:

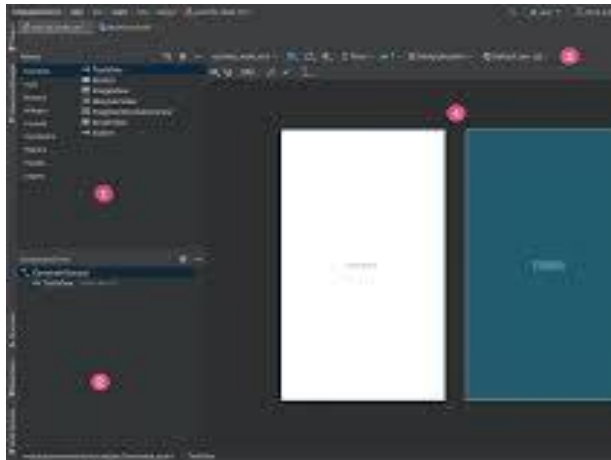
1. Create and execute a basic user interface.
2. Follow the process of adding and configuring text fields.
3. Learn to add and implement button functionality.

2.3.1 Creating and Running a Simple User Interface

Building a simple User Interface (UI) in Android Studio involves working with the Layout Editor and then running the app to see the changes. We'll add a Button next to the default "Hello World" text.

Step 1: Open the Layout Editor

- In your project, navigate to the res > layout folder.
- Double-click on the activity_main.xml file. This opens the Layout Editor.

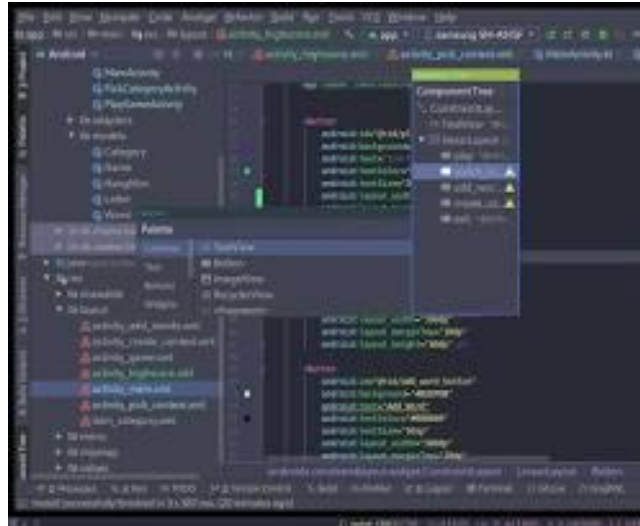


(Note: The default layout shows a "Hello World!" TextView)

Step 2: Use the Design View and Palette

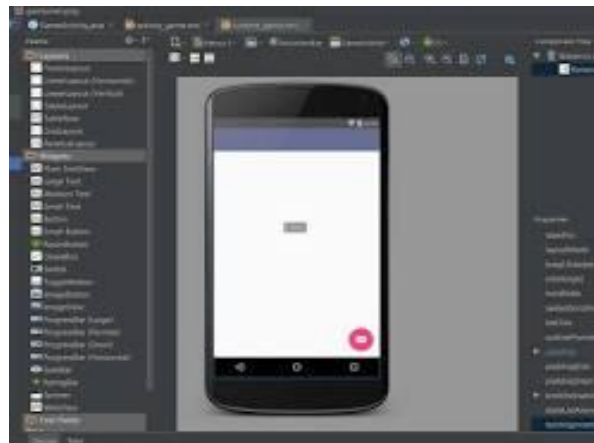
The Layout Editor has three tabs at the bottom-center: Code, Split, and Design.

- Make sure you are in the Design view.
- On the left side, you'll see the Palette, which contains various UI elements (widgets and layouts).
- On the right side, you have the Component Tree (a hierarchy of all elements in your layout) and the Attributes window.



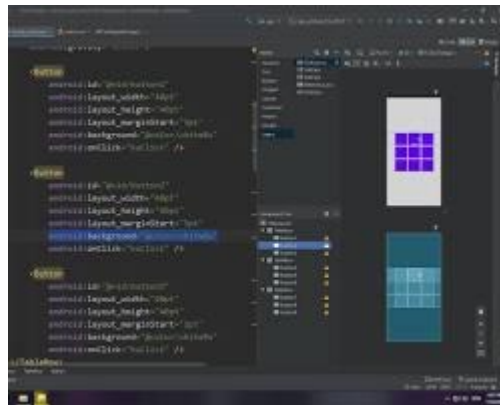
Step 3: Add a Button

- In the Palette (under the "Common" or "Widgets" section), locate the Button element.
- Drag and Drop: Click and hold the Button and drag it directly onto the layout canvas (the visual representation of your screen).
- Positioning: Because the default layout is usually a ConstraintLayout, you need to define constraints (rules for where the element should be positioned relative to other elements).
 - Drag the small circles (constraint anchors) on the sides of the Button and connect them to the sides of the screen or to the existing "Hello World!" text view. For a simple placement, connect the top, bottom, left, and right sides to the parent layout borders to center it, or connect the top of the button to the bottom of the "Hello World" text.



Step 4: Modify the Button Text (Attributes)

- With the newly added Button selected in the Component Tree or on the design canvas.
- Look at the Attributes window on the right side.
- Scroll down to the text attribute.
- Change the value from the default "Button" to something like "Click Me". Press Enter.



Step 5: Run the Project with the New UI

- Save: (The file is usually saved automatically, but you can press Ctrl+S or Cmd+S).
- Select Device: Ensure your Virtual Device (Emulator) is selected in the toolbar dropdown.
- Run: Click the green "Run" button (▶) on the toolbar.
- The Gradle build will run, and the updated app (now featuring the "Hello World!" text AND your new "Click Me" button) will be installed and launched on the emulator.
- You have successfully created and run a simple User Interface.

Exercise

Short Questions:

1. What is the role of the MainActivity.kt file in an Android project?
2. Define the purpose of activity_main.xml.
3. Why is it important to select a minimum SDK when creating a project?
4. Describe the steps to create a new AVD in Android Studio.
5. How can you run an app on a physical Android device instead of an emulator?
6. What happens during the Gradle build process?
7. What is the function of the Palette in the Layout Editor?
8. Explain how to add a Button in the Layout Editor and change its text.
9. What is the Component Tree, and why is it useful during UI design?
10. Describe what you typically see on the screen after successfully running a simple app created.

Multiple Choice Questions:

1. In Android Studio, which option do you select to start a new project?
 - a) File → Open Project
 - b) Welcome Screen → New Project
 - c) File → Import Project
 - d) Tools → Create New

2. Which project template provides a minimal starting structure with one screen?
 - a) Basic Activity
 - b) Empty Activity
 - c) Navigation Drawer Activity
 - d) Login Activity
3. The default programming languages for Android development are:
 - a) Java and Python
 - b) Kotlin and Java
 - c) C++ and Kotlin
 - d) HTML and CSS
4. What is Gradle used for in Android Studio?
 - a) Creating UI layouts
 - b) Managing project builds and dependencies
 - c) Debugging Android devices
 - d) Running background services
5. The term AVD stands for:
 - a) Android Virtual Device
 - b) Advanced Virtual Design
 - c) Android Version Data
 - d) Automated Virtual Deployment
6. To create a new virtual device in Android Studio, you go to:
 - a) File → Project Settings
 - b) Tools → Device Manager
 - c) Build → Emulator
 - d) Edit → Virtual Device
7. The default layout file in a simple Android project is:
 - a) MainActivity.kt
 - b) activity_main.xml
 - c) AndroidManifest.xml
 - d) strings.xml
8. What does the “Hello World!” text represent in a new Android project?
 - a) A Button element
 - b) A TextView element
 - c) An ImageView element
 - d) A Layout Manager
9. In the Layout Editor, the area that displays the visual design of your screen is called the:
 - a) Component Tree
 - b) Palette
 - c) Canvas
 - d) Logcat

10. To run your app on an emulator, you click the:

- a) Blue Debug Button
- b) Green Run Button
- c) Sync Project Button
- d) Refresh Gradle Button

Fill in the Blanks

1. The _____ file defines the user interface layout of an activity.
2. The two main files created by default in a simple project are _____ and _____.
3. The process of setting up dependencies and building the project is managed by _____.
4. To check if an emulator is configured, we look at the _____ menu on the toolbar.
5. The default layout type used in Android Studio is _____.
6. The window showing all layout components in a hierarchy is called the _____.
7. The programming language preferred for modern Android development is _____.
8. The unique identifier for every Android app is called the _____ name.
9. A virtual Android device created for testing is known as an _____.
10. To modify the text displayed on a Button, we change the _____ attribute in the Attributes panel.

Notes



Lined area for taking notes, consisting of multiple horizontal lines.



3. Configuring Value Added Services (VAS) in Android Applications for Telecom Devices



Unit 3.1 - Managing Data within Android Applications

Unit 3.2 - Messaging and Networking Service Integration on Android

Unit 3.3 - Fundamentals of Google Maps Integration and Location-Based Services

Unit 3.4 - Background Android Services



Key Learning Outcomes

By the end of this module, the participants will be able to:

1. Explain the purpose of data storage options and assist in saving or retrieving data using SQLite or SharedPreferences.
2. Support integration of messaging services like SMS and Firebase Cloud Messaging (FCM).
3. Assist in establishing basic networking operations using HTTP or Retrofit.
4. Configure and test location-based services such as GPS and Google Maps API.
5. Identify and assist in implementing different Android service types — foreground, background, and bound.
6. Support execution of background tasks using WorkManager or JobScheduler.
7. Follow standard procedures for testing, debugging, and documenting value-added service configurations.

UNIT 3.1: Managing Data within Android Applications

Unit Objectives

By the end of this unit, the participants will be able to:

1. Demonstrate the ability to assist in storing, retrieving, and updating user data using various Android storage methods.
2. Apply basic SQL operations to manage and maintain structured data in predefined databases.
3. Support secure and efficient data handling across internal, external, and cloud storage environments.
4. Assist in enabling controlled data sharing between Android applications using content providers.

3.1.1 Android Application Lifecycle

The Android application lifecycle governs how an app's components (Activities, Fragments, Services, etc.) are created, started, resumed, paused, stopped, and destroyed by the operating system. Understanding this is crucial for managing resources and ensuring a smooth user experience.

1. Activity Lifecycle

An Activity is a single, focused thing that the user can do. It's the most common application component and its lifecycle methods are vital for managing the user interface (UI) and state.

Method	Description	When It's Called
onCreate()	The Activity is created. This is where you perform basic application setup, such as defining the layout (setContentView()) and initializing views.	Once, when the activity is first created.
onStart()	The Activity becomes visible to the user (but may not be in the foreground).	After onCreate(), or when returning from onStop().
onResume()	The Activity is in the foreground and the user can interact with it.	After onStart(), or when returning from onPause(). This is where the app should begin user-intensive operations (e.g., animations, accessing a camera).
onPause()	The system calls this as the first indication that the user is leaving the activity (e.g., a dialogue appears). The activity is still partially visible, but usually should pause heavy resource consumption.	When the activity is partially obscured. Use this to save uncommitted data.
onStop()	The Activity is no longer visible to the user.	When the activity is completely hidden, or another activity covers it.
onDestroy()	The Activity is about to be destroyed and completely removed from memory.	Before the activity is explicitly finished (finish()) or the system is destroying it to reclaim resources. Use this to clean up resources like thread pools.
onRestart()	The Activity is stopped and is about to be restarted.	When an activity returns to the foreground after being stopped.

Key Lifecycle States

- Entire Lifetime (between `onCreate()` and `onDestroy()`): Perform all necessary setup and final resource release.
- Visible Lifetime (between `onStart()` and `onStop()`): The activity is visible to the user.
- Foreground Lifetime (between `onResume()` and `onPause()`): The activity is in front of all others and the user can fully interact with it.

2. Fragment Lifecycle

A Fragment represents a reusable portion of UI or behavior within an Activity. Its lifecycle is tightly coupled with its host Activity, but it also has independent methods to manage its view and existence.

Method	Description
<code>onAttach()</code>	Called when the Fragment is associated with its host Activity.
<code>onCreate()</code>	Called to do initial creation of the Fragment (non-UI setup).
<code>onCreateView()</code>	Called to create the view hierarchy associated with the fragment. This is where you inflate the layout.
<code>onViewCreated()</code>	Called immediately after <code>onCreateView()</code> . Setup of the views should occur here (e.g., setting up RecyclerView).
<code>onStart()</code>	The Fragment is visible. Corresponds to the Activity's <code>onStart()</code> .
<code>onResume()</code>	The Fragment is in the foreground and ready for interaction. Corresponds to the Activity's <code>onResume()</code> .
<code>onPause()</code>	The Fragment is partially obscured or about to stop interacting. Corresponds to the Activity's <code>onPause()</code> .
<code>onStop()</code>	The Fragment is no longer visible. Corresponds to the Activity's <code>onStop()</code> .
<code>onDestroyView()</code>	Called when the view hierarchy associated with the fragment is being removed. Clean up view-related resources here.
<code>onDestroy()</code>	Called to do final cleanup of the Fragment's state.
<code>onDetach()</code>	Called when the Fragment is disassociated from its host Activity.

View vs. Fragment Destruction

A key difference is the separation of view and fragment cleanup:

- `onDestroyView()`: Cleans up the views created in `onCreateView()`. The Fragment instance itself still exists (e.g., it might be in the back stack).
- `onDestroy()`: Cleans up the Fragment object and its associated resources.

4. Service Lifecycle

A Service is an application component that can perform long-running operations in the background, typically without a UI. Services are designed for operations that shouldn't be interrupted by the Activity lifecycle (e.g., playing music, downloading files).

There are two main types of Services:

A. Started Service (Runs an operation)

A started service is initiated by calling `startService()` and typically runs a single operation for an indefinite period.

Method	Description
<code>onCreate()</code>	Called when the Service is first created. Perform setup here.
<code>onStartCommand()</code>	Called every time a client calls <code>startService()</code> . The service performs the task and then should stop itself by calling <code>stopSelf()</code> .
<code>onDestroy()</code>	Called when the Service is no longer used and is being destroyed. Use this to clean up resources.

Service Flow: `onCreate()` \rightarrow `onStartCommand()` (multiple times) \rightarrow `onDestroy()`

B. Bound Service (Client-Server Interface)

A bound service allows application components (like Activities) to bind to it, allowing them to interact with the service and receive results.

Method	Description
<code>onCreate()</code>	Called when the Service is first created.
<code>onBind()</code>	Called when an Activity calls <code>bindService()</code> . This method must return an <code>IBinder</code> interface for clients to interact with the service.
<code>onUnbind()</code>	Called when all clients have disconnected from the service.
<code>onDestroy()</code>	Called when the Service is destroyed after all clients have unbound.

Bound Service Flow: `onCreate()` \rightarrow `onBind()` \rightarrow `onUnbind()` \rightarrow `onDestroy()`

3.1.2 Common Data Storage Options in Android

Choosing the right storage solution in Android is critical for performance, security, and a robust user experience. The main local options are suited for different data types and volumes, while cloud solutions provide scalability and synchronization.

Android offers several dedicated options for local data persistence, each with its own trade-offs.

Storage Option	Data Type	Volume/Complexity	Key Characteristics	Best Use Case
SharedPreferences	Primitive types (String, int, boolean, etc.)	Small, simple, key-value pairs	Lightweight, fast, stored in XML file, non-thread-safe	User preferences, settings, session tokens.
Room Database	Structured, Relational Data (Objects)	Large, complex datasets with relationships	Abstraction layer over SQLite, compile-time query checking, thread-safe, integrates with Jetpack components.	Structured application data, offline caching, user profiles.
SQLite Database	Structured, Relational Data (Tables)	Large, complex datasets with relationships	Raw SQL access, highly flexible, requires manual setup of tables, queries, and thread safety.	Legacy apps or cases requiring fine-grained, complex SQL control.
Internal Storage	Files (Bytes, text, JSON, images)	Large files/data that must be private	Files are stored in the app's dedicated private folder, automatically deleted on uninstall.	Sensitive app-specific data, configuration files, cache files.

1. SharedPreferences

SharedPreferences is the simplest mechanism for saving small amounts of data.

- **How it Works:** Data is stored as key-value pairs in a private XML file. It is usually cached in memory for fast access.
- **Access:** Use `getSharedPreferences("filename", Context.MODE_PRIVATE)` or `getPreferences(Context.MODE_PRIVATE)`. Write data using a `SharedPreferences.Editor` and calling `apply()` (asynchronously) or `commit()` (synchronously).
- **Caveat:** It is not designed for large datasets. Storing complex objects requires manual serialization (e.g., converting an object to a JSON string), which is cumbersome. It is generally not thread-safe without proper handling.

2. Room Database

Room is part of Android Jetpack and is the recommended way to handle structured, relational data. It simplifies working with the underlying SQLite database.

- **How it Works:** It uses annotations (`@Entity`, `@Dao`, `@Database`) to define database structure, queries, and access methods, providing an object-oriented abstraction over SQLite.
- **Key Benefits:**
 - **Compile-time SQL verification:** Catches errors before runtime.
 - **Boilerplate reduction:** Less manual SQL and cursor handling.
 - **LiveData/Flow integration:** Supports reactive programming for observing data changes.
- **When to Use:** When you have multiple pieces of related data (tables) that need to be queried, filtered, or updated in a structured way, especially for offline caching of server data.

3. SQLite Database

SQLite is a powerful, lightweight, and open-source relational database engine built directly into Android. Room is essentially a library that uses SQLite.

- How it Works: Developers interact directly with the SQLiteDatabase object, writing raw SQL queries to define tables (CREATE TABLE) and perform CRUD operations (INSERT, SELECT, UPDATE, DELETE).
- When to Use: Only use raw SQLite directly if you need features or specific performance optimizations that Room does not easily expose, or if working with a legacy codebase. For all new development, Room is preferred over raw SQLite.

3.1.3 Best Practices for Android Data Storage

A. Local Data Storage Best Practices

Choose the Right Tool:

- Settings/Prefs: Use SharedPreferences (or the newer DataStore which is asynchronous and safer).
- Structured Data: Use Room Database.
- Sensitive/Private Files: Use Internal Storage (e.g., getFilesDir()).
- Shared/Large Media: Use Shared Storage (MediaStore API, especially post-Android 10 Scoped Storage).

1. Ensure Thread Safety:

- Always perform database (Room/SQLite) and file I/O operations off the main thread to prevent Application Not Responding (ANR) errors. Room and DataStore inherently support this using Kotlin Coroutines/Flows or Executors.

2. Data Security:

- Encryption: Use EncryptedSharedPreferences for sensitive key-value data (e.g., API tokens). Use libraries like SQLCipher or Android's own Security library for encrypted databases if storing highly sensitive structured data.
- No Hardcoding: Never hardcode sensitive information (like API keys or passwords) directly in code or plain resource files.

3. Manage Data Integrity:

- Transactions: For database operations involving multiple steps (e.g., updating data in two tables), wrap them in a transaction to ensure atomicity (all or nothing). Room provides excellent transaction support.

4. Use Modern APIs:

- Prefer DataStore over SharedPreferences for new code due to its asynchronous and thread-safe nature.
- Target Scoped Storage (Android 10+): When dealing with external storage, use the MediaStore API for media files and the Storage Access Framework for non-media documents to comply with modern Android security policies.

B. Cloud-Based Storage Solutions Best Practices

Cloud solutions (like Firebase Realtime Database, Firestore, AWS DynamoDB, or Azure Cosmos DB) are used for data synchronization, backup, and handling data volumes too large for local storage.

1. Prioritize Offline Support and Caching:

- Implement a Repository Pattern that acts as a single source of truth. The repository should first check the local Room database and then query the cloud if needed. This provides a fast, reliable offline experience.
- Use cloud features (like Firestore's offline persistence) that automatically handle data syncing when the device is back online.

2. Secure Communication (API Keys & Tokens):

- HTTPS: Always use HTTPS for all communication between the app and the cloud service.
- Authentication: Use secure authentication mechanisms like OAuth 2.0 or Firebase Authentication to verify the user's identity before allowing read/write access to data.
- Proxy Secrets: Use a proxy server or Android's Secrets Gradle Plugin to securely inject API keys and sensitive configuration, preventing them from being directly exposed in the APK.

3. Implement Robust Access Control (Security Rules):

- The most critical practice: Never rely on the client-side code for security.
- Database Rules: Configure server-side security rules (e.g., Firebase Security Rules, AWS IAM policies) to strictly define who can read, write, or delete specific data nodes. For example, a user should only be able to modify their own profile data.

4. Optimize Data Structure for Cloud:

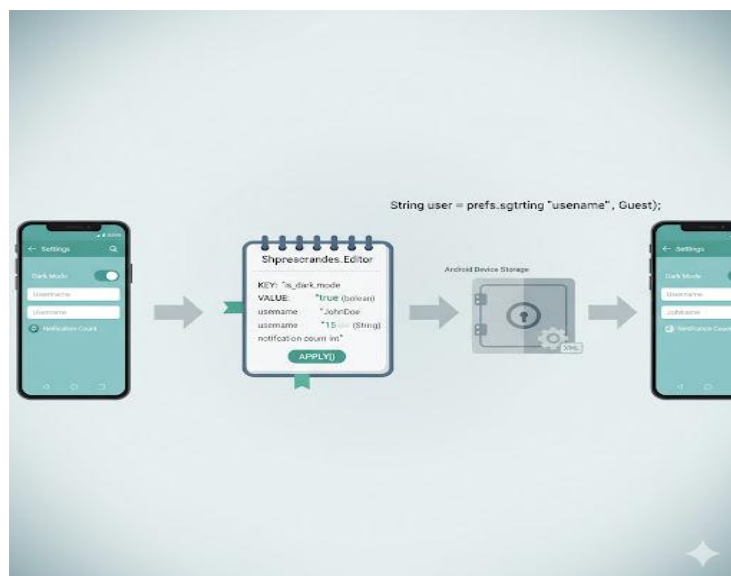
- Denormalization: Cloud NoSQL databases (like Firebase/Firestore) often benefit from denormalization (duplicating data) to reduce the number of queries needed to fetch related information, optimizing cost and latency.
- Minimize Transfer: Only download the necessary data. Use filtering, pagination, and real-time listeners for specific data paths to minimize bandwidth usage.

3. Retrieve Data: Get data using the appropriate getter method, providing a default value in case the key doesn't exist.

Java

```
String username = sharedPreferences.getString("username", "Guest");int userId = sharedPreferences.getInt("user_id", 0);boolean isLoggedIn = sharedPreferences.getBoolean("is_logged_in", false);
```

4. Image for Shared Preferences Concept:



Room Database is an abstraction layer over SQLite that makes database interaction much easier and safer. It's part of Android Jetpack and is recommended for robust local data storage.

Steps for Room Database:

1. **Add Room Dependencies:** In your build.gradle (app-level), add the necessary Room dependencies.

Gradle

```
def room_version = "2.5.0" // Use the latest stable version
implementation "androidx.room:room-runtime:$room_version"
annotationProcessor "androidx.room:room-compiler:$room_version"// For Kotlin:// kapt
"androidx.room:room-compiler:$room_version"// For Kotlin Coroutines support:// implementation
"androidx.room:room-ktx:$room_version"
```

2. **Define an Entity (Table):** Create a data class (Kotlin) or Java class annotated with @Entity to represent a table in your database.

Java

```
// User.java@Entity(tableName = "users")public class User {
    @PrimaryKey(autoGenerate = true)
    public int id;
```

```
@ColumnInfo(name = "first_name")
public String firstName;
```

```
@ColumnInfo(name = "last_name")
public String lastName;
```

// Constructor and getters/setters

```
public User(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}
```

```
}
```

3. **Define a DAO (Data Access Object):** Create an interface annotated with @Dao that contains methods for database operations (insert, query, update, delete).

Java

// UserDao.java

```
@Daopublic interface UserDao {
    @Insert
    void insert(User user);
    @Query("SELECT * FROM users")
    List<User> getAllUsers();
    @Query("SELECT * FROM users WHERE id = :userId LIMIT 1")
    User getUserById(int userId);
    @Update
    void update(User user);
    @Delete
    void delete(User user);
}
```


4. Define a Database Class: Create an abstract class that extends `RoomDatabase` and is annotated with `@Database`. This class defines the entities and version of your database.¹

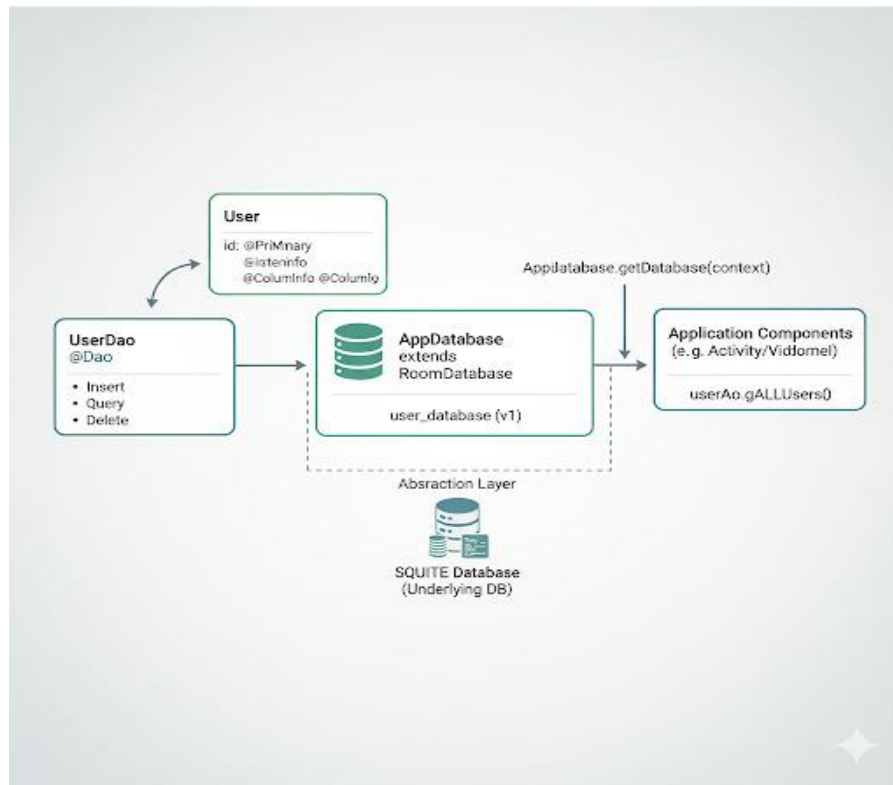
Java

```
// AppDatabase.java
@Database(entities = {User.class}, version = 1, exportSchema = false)
public abstract class AppDatabase extends RoomDatabase {
    public abstract UserDao userDao();
    private static volatile AppDatabase INSTANCE;
    public static AppDatabase getDatabase(final Context context) {
        if (INSTANCE == null) {
            synchronized (AppDatabase.class) {
                if (INSTANCE == null) {
                    INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                        AppDatabase.class, "user_database")
                        .build();
                }
            }
        }
        return INSTANCE;
    }
}
```

5. Access the Database: Get an instance of your database and use the DAO to perform operations. (Typically done in a `ViewModel` or `Repository`).

Java

```
// In an Activity or Fragment:
AppDatabase db = AppDatabase.getDatabase(getApplicationContext());
UserDao userDao = db.userDao();
// Perform operations (preferably on a background thread)
// Example: Insert a user
new Thread(() -> {
    userDao.insert(new User("Jane", "Doe"));
}).start();
// Example: Retrieve all users
new Thread(() -> {
    List<User> users = userDao.getAllUsers();
    // Update UI with users
}).start();
Image for Room Database Architecture:
```



3.1.5 Updating User Preferences and Retrieving Stored Data

It involves using `SharedPreferences` for user preferences, as it's the most common and straightforward method.

Steps for Updating User Preferences:

1. Identify Preferences UI: Locate the part of the Android application where user preferences are displayed and can be modified (e.g., a "Settings" screen). This usually involves UI elements like `Switch`, `EditText`, `RadioButtons`, or `Spinner`.

2. Attach Listeners: For each preference UI element, attach an appropriate listener to detect when the user makes a change.

- For a `Switch`: `setOnCheckedChangeListener`
- For an `EditText`: `addTextChangedListener`
- For `RadioButtons`: `setOnCheckedChangeListener` on a `RadioGroup`
- For a `Spinner`: `setOnItemSelectedListener`

3. Store Changes with SharedPreferences: Inside the listener callbacks, retrieve the new value from the UI element and store it using SharedPreferences.Editor.

Java

```
// Example for a Switch preference (e.g., dark mode toggle)
Switch darkModeSwitch = findViewById(R.id.dark_mode_switch);
SharedPreferences sharedPref = getSharedPreferences("MyUserPrefs", Context.MODE_PRIVATE);
darkModeSwitch.setOnCheckedChangeListener((buttonView, isChecked) -> {
    SharedPreferences.Editor editor = sharedPref.edit();
    editor.putBoolean("dark_mode_enabled", isChecked);
    editor.apply();
    // Optionally, apply the theme change immediately
    recreate(); // Or handle theme change without restarting activity
});
```

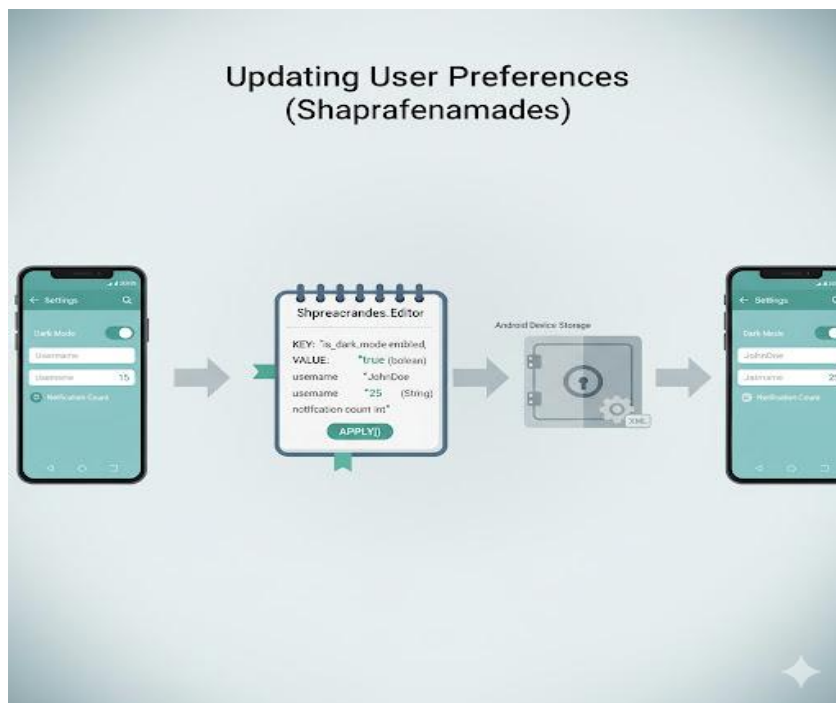


Fig. 3.1.: Updating Preferences UI to SharedPreferences

Steps for Retrieving Stored Data (User Preferences):

1. Access SharedPreferences: Obtain an instance of SharedPreferences as described previously.
2. Retrieve Values: Use the appropriate get...() method to retrieve the stored data. Always provide a default value.

Java

```
SharedPreferences sharedPref = getSharedPreferences("MyUserPrefs", Context.MODE_PRIVATE);

String username = sharedPref.getString("username", "Guest");
int notificationCount = sharedPref.getInt("notification_count", 0);
boolean darkModeEnabled = sharedPref.getBoolean("dark_mode_enabled", false);
```

3. Populate UI Elements: Use the retrieved values to set the initial state of your preference UI elements. This should typically be done in onCreate() or onResume() of your activity/fragment.

Java

```
EditText usernameEditText = findViewById(R.id.username_edittext);
Switch darkModeSwitch = findViewById(R.id.dark_mode_switch);
usernameEditText.setText(username);
darkModeSwitch.setChecked(darkModeEnabled);
```

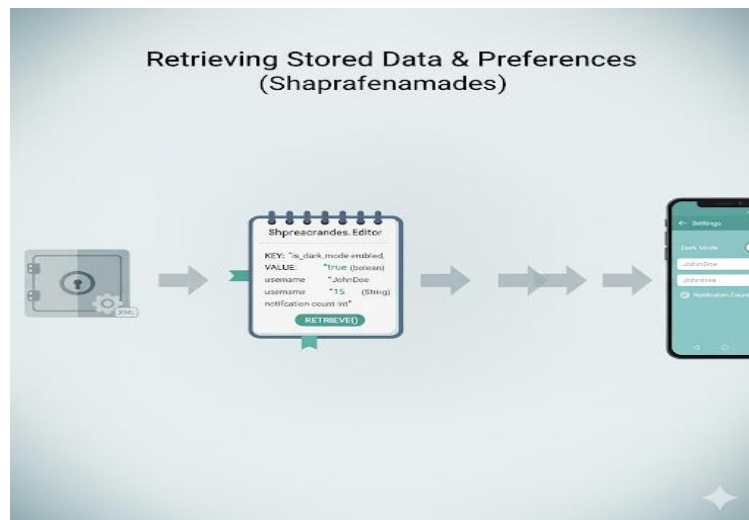


Fig 3.1. Retrieving Preferences to Populate UI

3.1.6 Storing and Accessing Data

Internal Storage

Internal storage is always available, private to your app, and automatically deleted when the app is uninstalled. It's suitable for sensitive data that should not be accessible by other apps.

Steps for Internal Storage:

1. **Open/Create File:** Use `openFileOutput()` to write to a file and `openFileInput()` to read from it.

Java

```
// Writing to internal storage
```

```
String filename = "my_private_file.txt";
```

```
String fileContents = "Hello, internal storage!";try (FileOutputStream fos =
openFileOutput(filename, Context.MODE_PRIVATE)) {
```

```
    fos.write(fileContents.getBytes());
```

```
} catch (IOException e) {
```

```
    e.printStackTrace();
```

```
}
```

```
// Reading from internal storage
try (FileInputStream fis = openFileInput(filename)) {
    InputStreamReader inputStreamReader = new InputStreamReader(fis, StandardCharsets.UTF_8);
    StringBuilder stringBuilder = new StringBuilder();
    try (BufferedReader reader = new BufferedReader(inputStreamReader)) {
        String line = reader.readLine();
        while (line != null) {
            stringBuilder.append(line).append('\n');
            line = reader.readLine();
        }
    }
    String contents = stringBuilder.toString();
    // Use contents
} catch (IOException e) {
    e.printStackTrace();
}
```

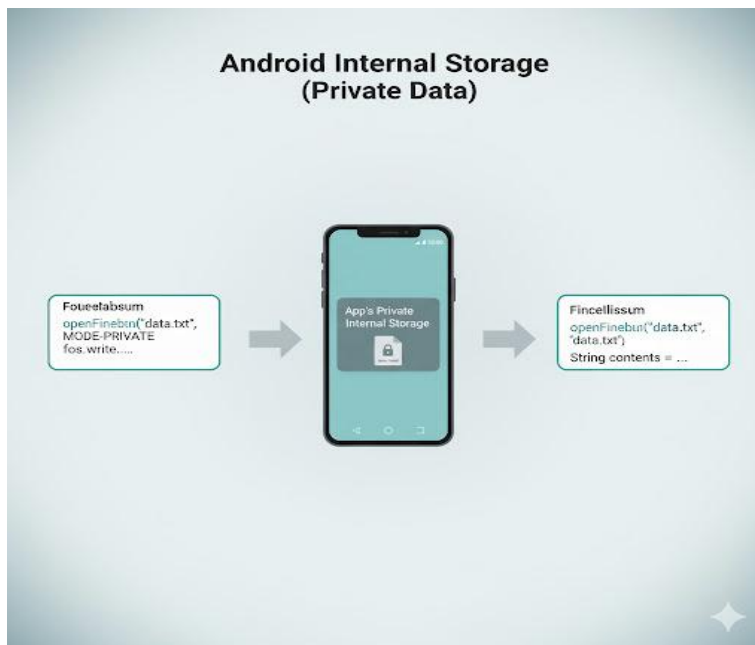


Fig 3.1. Internal Storage

External Storage (SD Card)

External storage can be public (accessible by other apps and the user) or private to your app. It's often used for larger files, media, or data that might be shared. Requires permissions.

Steps for External Storage:

1. **Add Permissions:** In `AndroidManifest.xml`. For public external storage, you'll need `WRITE_EXTERNAL_STORAGE` (and implicitly `READ_EXTERNAL_STORAGE` on older APIs, or explicitly `READ_EXTERNAL_STORAGE` on newer ones if just reading).
2. XML


```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
    android:maxSdkVersion="29" /><uses-permission
    android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Note: From Android 10 (API 29) onwards, scoped storage is enforced, making WRITE_EXTERNAL_STORAGE less effective for public storage. Use MediaStore for shared media or getExternalFilesDir() for app-private external storage.

2. Check Media Availability: Always check if external storage is mounted before attempting to read/write.

Java

```
public boolean isExternalStorageWritable() {
    return Environment.MEDIA_MOUNTED.equals(Environment.getExternalStorageState());
}
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    return Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state);
}
```

3. Get Directory: For app-private files on external storage, use getExternalFilesDir(). For public files, use Environment.getExternalStoragePublicDirectory() (deprecated in newer APIs, use MediaStore instead).

Java

```
// App-private external storage
File file = new File(getExternalFilesDir(Environment.DIRECTORY_DOCUMENTS),
    "my_external_file.txt");
// Writingtry (FileOutputStream fos = new FileOutputStream(file)) {
    fos.write("Data on external storage.".getBytes());
} catch (IOException e) {
    e.printStackTrace();
}
// Readingtry (FileInputStream fis = new FileInputStream(file)) {
    // Read file contents
} catch (IOException e) {
    e.printStackTrace();}
```



Fig 3.1. External Storage (App-private)

Cloud-Based Solutions (Firebase, AWS, or Azure)

These provide scalable and persistent storage, allowing data synchronization across devices and robust backend capabilities. This is a very broad topic, so we'll focus on a common use case like Firebase Realtime Database.

Steps for Firebase Realtime Database:

1. Set Up Firebase Project:

- Go to the Firebase Console.
- Create a new project.
- Add your Android app to the project (follow the on-screen instructions, including downloading google-services.json and adding Firebase dependencies to your build.gradle files).

2. Add Realtime Database Dependencies: In build.gradle (app-level).

Gradle

implementation 'com.google.firebase:firebase-database:20.1.0' // Use latest version

3. Define Data Model: Create a Java/Kotlin class to represent the data you want to store.

Java

```
public class Message {
    public String sender;
    public String text;
    public long timestamp;
```

```
    public Message() {
        // Default constructor required for calls to DataSnapshot.getValue(Message.class)
    }
```

```
    public Message(String sender, String text) {
        this.sender = sender;
        this.text = text;
        this.timestamp = System.currentTimeMillis();
    }
}
```

4. Get Database Reference:

Java

```
DatabaseReference database = FirebaseDatabase.getInstance().getReference();
```

5. Write Data: Use setValue(), push(), or updateChildren().

Java

```
// Write a new message
```

```
String messageId = database.child("messages").push().getKey(); // Generates unique ID
```

```
Message newMessage = new Message("Alice", "Hello from Android!");
```

```
database.child("messages").child(messageId).setValue(newMessage);
```

```
// Or set a value directly
```

```
database.child("users").child("user123").child("name").setValue("Bob");
```

6. Read Data: Use addValueEventListener for real-time updates or addListenerForSingleValueEvent for a one-time read.

Java

```
// Read all messages in real-time
```

```
database.child("messages").addValueEventListener(new ValueEventListener() {
    @Override
```

```
@Override
public void onCancelled(@NonNull DatabaseError error) {
    Log.e("Firebase", "Failed to read value.", error.toException());
}
});
```

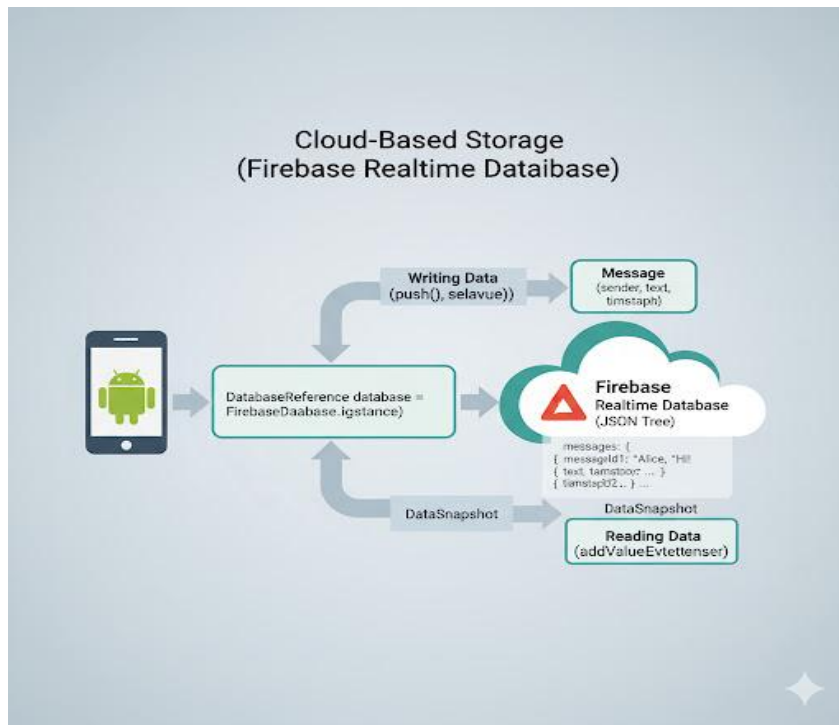


Fig 3.1. Firebase Realtime Database

Encrypted Storage Mechanisms

For highly sensitive data, simply storing it in internal storage might not be enough. Encryption adds another layer of security. Android offers EncryptedSharedPreferences and EncryptedFile (from the Security library).

Steps for Encrypted Shared Preferences:

1. Add Security Library Dependency: In your build.gradle (app-level).

Gradle

implementation "androidx.security:security-crypto:1.1.0-alpha03" // Use the latest stable alpha or beta

2. Create a Master Key: Use MasterKeys to generate a secure master key for encryption.

Java

```
try {
    String masterKeyAlias = MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC);
    // ...
} catch (GeneralSecurityException | IOException e) {
    e.printStackTrace();
}
```


3. Initialize EncryptedSharedPreferences:

```

Java
try {
    String masterKeyAlias = MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC);

    SharedPreferences encryptedSharedPreferences = EncryptedSharedPreferences.create(
        "secret_prefs", // Filename
        masterKeyAlias,
        getApplicationContext(),
        EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
        EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
    );
    // Now use encryptedSharedPreferences just like regular SharedPreferences
    encryptedSharedPreferences.edit()
        .putString("secret_token", "super_secret_value_123")
        .apply();

    String secretToken = encryptedSharedPreferences.getString("secret_token", null);
    Log.d("EncryptedStorage", "Retrieved secret: " + secretToken);
} catch (GeneralSecurityException | IOException e) {
    e.printStackTrace();
}

```

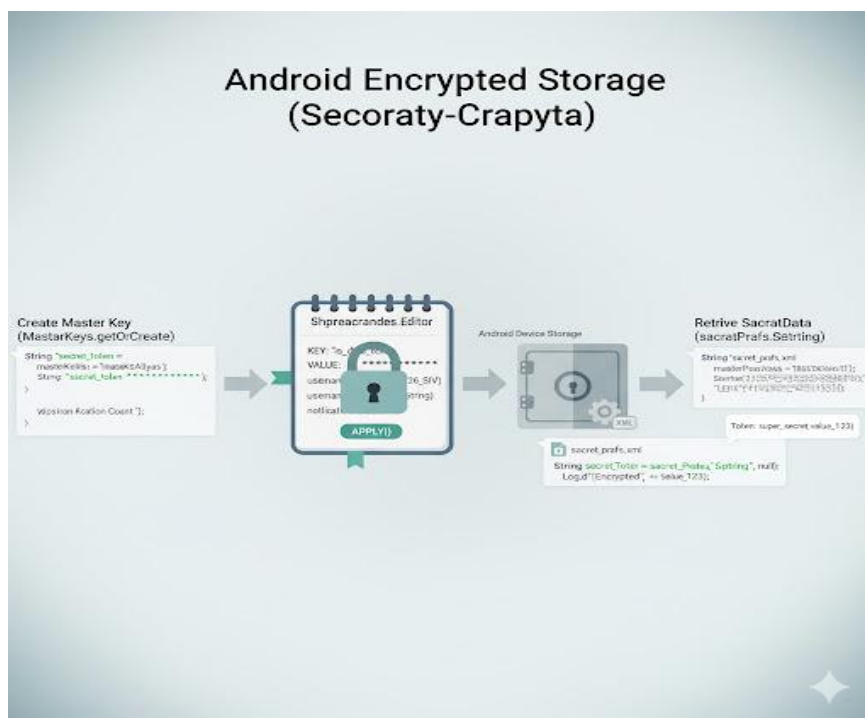


Fig 3.1. EncryptedSharedPreferences

3.1.7 Using Structured Query Language (SQL)

This competency focuses on direct SQL usage, which is typically done when working directly with SQLiteOpenHelper or when using Room with @Query annotations. We'll use a SQLiteOpenHelper example for raw SQL demonstration.

Prerequisite: A class extending SQLiteOpenHelper is needed to manage database creation and versioning.

Java

```
public class MyDatabaseHelper extends SQLiteOpenHelper {
    private static final String DATABASE_NAME = "mydatabase.db";
    private static final int DATABASE_VERSION = 1;

    public MyDatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        // This is where you create your tables
        String CREATE_TABLE_USERS = "CREATE TABLE users (" +
            "id INTEGER PRIMARY KEY AUTOINCREMENT," +
            "name TEXT," +
            "email TEXT);";
        db.execSQL(CREATE_TABLE_USERS);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS users");
        onCreate(db);
    }
}
```

Now, let's look at CRUD operations:

Create (INSERT)

SQL Command: INSERT INTO table_name (column1, column2) VALUES (value1, value2);

Android Implementation (SQLiteDatabase insert() or execSQL()):

Java

```
MyDatabaseHelper dbHelper = new MyDatabaseHelper(context);
SQLiteDatabase database = dbHelper.getWritableDatabase();
// Option 1: Using ContentValues (recommended for safety)
ContentValues values = new ContentValues();
values.put("name", "Alice");
values.put("email", "alice@example.com"); long newRowId = database.insert("users", null, values);
Log.d("SQL_CRUD", "Inserted new row with ID: " + newRowId);
// Option 2: Using raw SQL (be careful with SQL injection!)
database.execSQL("INSERT INTO users (name, email) VALUES ('Bob', 'bob@example.com');");
```

```
database.close();
```

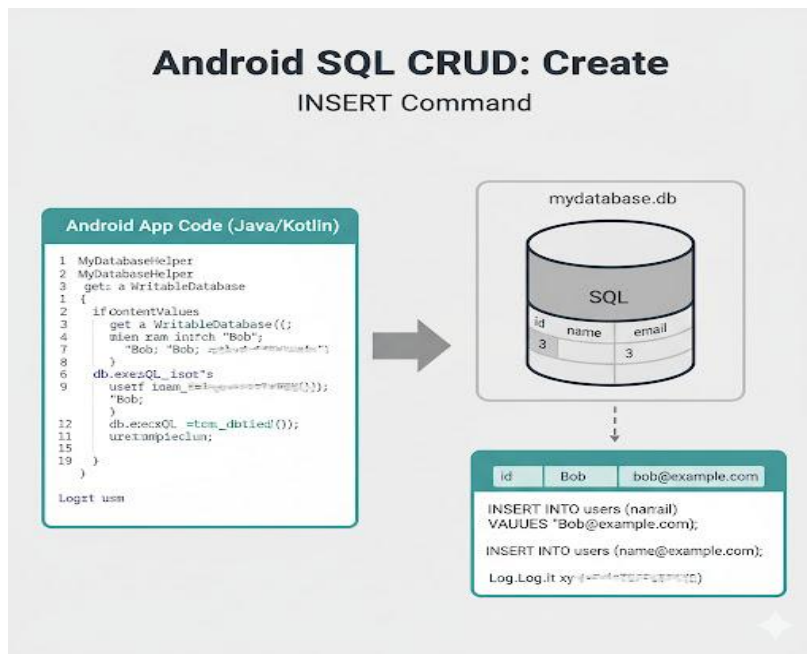


Fig 3.1. SQL INSERT

Read (SELECT)

SQL Command: `SELECT column1, column2 FROM table_name WHERE condition;`

Android Implementation (SQLiteDatabase query() or rawQuery()):

Java

```
MyDatabaseHelper dbHelper = new MyDatabaseHelper(context);
```

```
SQLiteDatabase database = dbHelper.getReadableDatabase();
```

```
// Option 1: Using query() (recommended)
```

```
String[] projection = {"id", "name", "email"};
```

```
String selection = "name = ?";
```

```
String[] selectionArgs = {"Alice"};
```

```
Cursor cursor = database.query(
```

```
    "users",      // The table to query
```

```
    projection,   // The columns to return
```

```
    selection,    // The columns for the WHERE clause
```

```
    selectionArgs, // The values for the WHERE clause
```

```
    null,         // don't group the rows
```

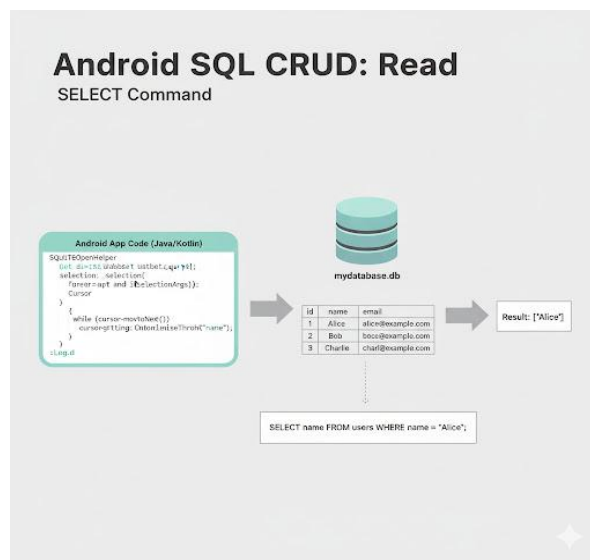
```
    null,         // don't filter by row groups
```

```
    null          // The sort order
```

```
);
```

```
List<String> usernames = new ArrayList<>();while (cursor.moveToNext()) {
    String userName = cursor.getString(cursor.getColumnIndexOrThrow("name"));
    usernames.add(userName);
}
cursor.close();
Log.d("SQL_CRUD", "Users found: " + usernames.toString());
// Option 2: Using.rawQuery()
Cursor rawCursor = database.rawQuery("SELECT * FROM users WHERE email LIKE ?", new
    String[]{"%example.com"}); // Process rawCursor as above
rawCursor.close();

database.close();
```



```
List<String> userNames = new ArrayList<>();while (cursor.moveToNext()) {
    String userName = cursor.getString(cursor.getColumnIndexOrThrow("name"));
    userNames.add(userName);
}
cursor.close();
Log.d("SQL_CRUD", "Users found: " + userNames.toString());
// Option 2: Using.rawQuery()
Cursor rawCursor = database.rawQuery("SELECT * FROM users WHERE email LIKE ?", new
    String[]{"%example.com"}); // Process rawCursor as above
rawCursor.close();
database.close();
```

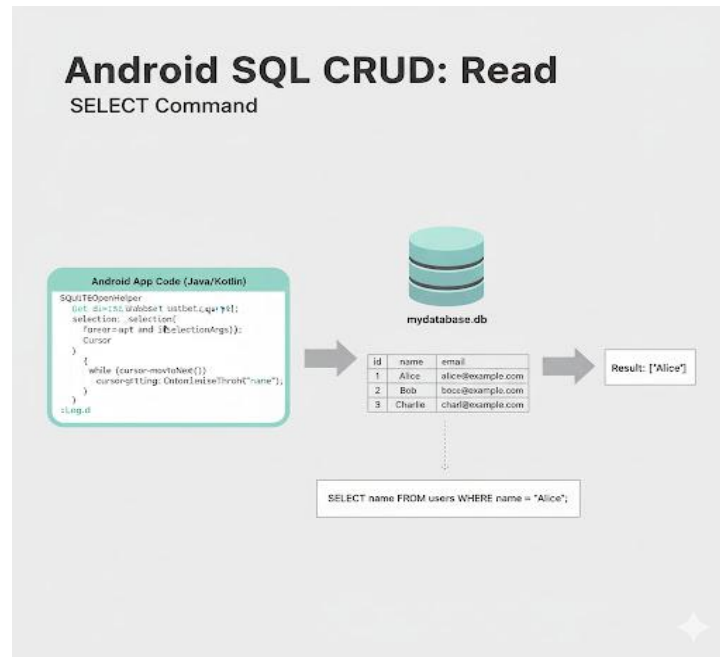


Fig 3.1.1 SQL SELECT

3.1.8 Configuring Predefined Databases

This competency focuses on the initial setup and management of a local database in an Android application, specifically using the built-in SQLite capabilities, often managed via the SQLiteOpenHelper class.

The SQLiteOpenHelper is a utility class that handles the creation and version management of the database. When the application is installed or first run, it checks the database version and calls the appropriate methods (onCreate() or onUpgrade()).

Steps for Configuring an SQLite Database

The following steps define the required boilerplate code and configuration for using a predefined SQLite database structure.

Step 1: Define Database Constants and Schema (The Guidelines)

First, define the core guidelines provided for the database: the database file name, version number, and the structure of the tables (schema).

Java

```
public final class DatabaseContract {
    // Database name and version
    public static final String DATABASE_NAME = "telecom_vas.db";
    public static final int DATABASE_VERSION = 1;
```

```
// Table definition
```

```
public static final class ServiceEntry implements BaseColumns {
    public static final String TABLE_NAME = "services";
    public static final String COLUMN_NAME = "name";
    public static final String COLUMN_CODE = "ussd_code";
    public static final String COLUMN_STATUS = "is_active";
}
}
```

Step 2: Implement the SQLiteOpenHelper Class

Create a custom class that extends SQLite Open Helper. This class ensures the database file is correctly created and that the schema is maintained when the version changes.

Define the SQL Creation Command: Use the schema guidelines to write the raw SQL.

Java

```
private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + DatabaseContract.ServiceEntry.TABLE_NAME + " (" +
    DatabaseContract.ServiceEntry._ID + " INTEGER PRIMARY KEY," + // Required by BaseColumns
    DatabaseContract.ServiceEntry.COLUMN_NAME + " TEXT," +
    DatabaseContract.ServiceEntry.COLUMN_CODE + " TEXT UNIQUE," +
    DatabaseContract.ServiceEntry.COLUMN_STATUS + " INTEGER DEFAULT 0)";
```

Override onCreate():

This method runs only once, the first time the app attempts to access the database, to create the tables.

Java

```
public class VasDbHelper extends SQLiteOpenHelper {
    public VasDbHelper(Context context) {
        super(context, DatabaseContract.DATABASE_NAME, null, DatabaseContract.DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        // Execute the schema creation command
        db.execSQL(SQL_CREATE_ENTRIES);
        Log.d("DB_CONFIG", "Database created and schema applied.");
    }
}
```

```
// Define how to handle schema changes (usually dropping and recreating for simplicity during development)
```

```
db.execSQL("DROP TABLE IF EXISTS " + DatabaseContract.ServiceEntry.TABLE_NAME);
```

```
onCreate(db);
```

```
}
```

```
}
```

Step 3: Initialize and Access the Database

In your application code (e.g., Activity or Repository), initialize the helper and get a readable or writable instance.

Java

```
// In an activity's onCreate() or a dedicated Repository class
```

```
VasDbHelper dbHelper = new VasDbHelper(context);
```

```
// Get a writable database instance (this triggers onCreate() if the database file doesn't exist)
```

```
SQLiteDatabase db = dbHelper.getWritableDatabase();
```

```
// You can now perform CRUD operations using this 'db' object.
```

3.1.9 Implementing Content Providers to Enable Data Sharing Between Apps

A Content Provider manages access to a structured set of data. It serves as a standard interface to allow different applications to query, insert, update, or delete data, even if the apps are running in different processes. This is essential for exposing your application's data to other components (including widgets or other apps) securely.

Steps for Implementing a Content Provider

Step 1: Define the URI Authority and Paths

A Content Provider is identified by its unique authority and uses a URI (Uniform Resource Identifier) to point to the specific data being accessed.

Define Authority: This must be a unique, fully qualified name, usually based on the app's package name.

Java

```
public static final String AUTHORITY = "com.example.telecom.vas.provider";public static final Uri
BASE_CONTENT_URI = Uri.parse("content://" + AUTHORITY);
```

Define Content URIs: Append the table name to the base URI to create specific paths.

Java

```
// Content URI for the entire 'services' tablepublic static final Uri CONTENT_URI =
```

```
BASE_CONTENT_URI.buildUpon().appendPath(DatabaseContract.ServiceEntry.TABLE_NAME).build();
```

```
// MIME Types (used to describe the type of data returned)public static final String
CONTENT_LIST_TYPE =
```

```
ContentResolver.CURSOR_DIR_BASE_TYPE + "/" + AUTHORITY + "/" +
DatabaseContract.ServiceEntry.TABLE_NAME;public static final String CONTENT_ITEM_TYPE =
```

```
ContentResolver.CURSOR_ITEM_BASE_TYPE + "/" + AUTHORITY + "/" +
DatabaseContract.ServiceEntry.TABLE_NAME;
```

Step 2: Implement the ContentProvider Class

Create a class that extends ContentProvider and implement the required abstract methods (query, insert, update, delete, getType, onCreate). This uses the SQLiteOpenHelper from PC5 internally.

URI Matching: Use UriMatcher to efficiently map incoming URIs to specific actions (e.g., querying a whole table vs. a single row).

Java

```
public class VasContentProvider extends ContentProvider {
    private VasDbHelper mOpenHelper;
    private static final UriMatcher sUriMatcher = buildUriMatcher();

    // Constants for the URI Matcher
    private static final int SERVICES = 100;
    private static final int SERVICE_ID = 101;

    public static UriMatcher buildUriMatcher() {
        final UriMatcher matcher = new UriMatcher(UriMatcher.NO_MATCH);
        final String authority = DatabaseContract.AUTHORITY;
        // URI for the entire table: content://com.example.telecom.vas.provider/services
        matcher.addURI(authority, DatabaseContract.ServiceEntry.TABLE_NAME, SERVICES);
        // URI for a single row: content://com.example.telecom.vas.provider/services/12
        matcher.addURI(authority, DatabaseContract.ServiceEntry.TABLE_NAME + "/" + "#", SERVICE_ID);
        return matcher;
    }

    @Override
    public boolean onCreate() {
        mOpenHelper = new VasDbHelper(getContext()); // Initialize the helper
        return true;
    }

    @Override
    public Cursor query(@NonNull Uri uri, String[] projection, String selection, String[] selectionArgs,
String sortOrder) {
        SQLiteDatabase db = mOpenHelper.getReadableDatabase();
        int match = sUriMatcher.match(uri);
        Cursor cursor;

        switch (match) {
            case SERVICES:
                // Query the whole table
                cursor = db.query(DatabaseContract.ServiceEntry.TABLE_NAME, projection, selection,
selectionArgs, null, null, sortOrder);
```



```

break;

    case SERVICE_ID:
        // Query a single item by ID
        String id = uri.getPathSegments().get(1);
        selection = DatabaseContract.ServiceEntry._ID + "=?";
        selectionArgs = new String[]{id};

        cursor = db.query(DatabaseContract.ServiceEntry.TABLE_NAME, projection, selection,
selectionArgs, null, null, sortOrder);

        break;

    default:
        throw new UnsupportedOperationException("Unknown URI: " + uri);
}

// Notify content resolver of changes
cursor.setNotificationUri(getContext().getContentResolver(), uri);
return cursor;
}

// Must implement insert, update, delete, and getType as well.
// ... (Implementation for other CRUD methods) ...
}

```

Step 3: Register the Content Provider in AndroidManifest.xml

For Android to recognize the provider and grant access permissions, it must be declared in the application manifest.

Declare the <provider> Tag:

XML

```

<application
    ... >
    <provider
        android:name=".data.VasContentProvider"
        android:authorities="com.example.telecom.vas.provider"
        android:exported="true"
        android:readPermission="com.example.telecom.vas.provider.READ_PERMISSION"
        android:writePermission="com.example.telecom.vas.provider.WRITE_PERMISSION"
    />
</application>

```

Define Custom Permissions (Optional but Recommended for Security): To restrict access to only trusted apps, define custom permissions that external apps must request.

XML

```
<permission
    android:name="com.example.telecom.vas.provider.READ_PERMISSION"
    android:protectionLevel="normal"
    android:description="@string/permission_read_vas_desc" />
<permission
    android:name="com.example.telecom.vas.provider.WRITE_PERMISSION"
    android:protectionLevel="normal"
    android:description="@string/permission_write_vas_desc" />
```

Step 4: Access Data from an External App

An external application uses the ContentResolver and the content URI to access the data without knowing the underlying database implementation.

Java

```
// In a separate, external application:
Uri servicesUri = Uri.parse("content://com.example.telecom.vas.provider/services");
ContentResolver resolver = getContentResolver();

Cursor cursor = resolver.query(
    servicesUri,
    null, // Projection
    null, // Selection
    null, // SelectionArgs
    null // SortOrder
);
if (cursor != null) {
    while (cursor.moveToNext()) {
        String serviceName = cursor.getString(cursor.getColumnIndexOrThrow("name"));
        // Process data
    }
    cursor.close();
}
```

2. MMS (Multimedia Messaging Service)

MMS, or Multimedia Messaging Service, is an extension of SMS that allows users to send and receive multimedia content. This includes images, audio clips, video clips, and longer text messages, which SMS alone cannot handle.

How it Works:

- **Content Packaging:** When you send an MMS, the content (image, video, text) is encoded and packaged.
- **MMS Center (MMSC):** This package is sent to a Multimedia Messaging Service Center (MMSC), which is similar in function to an SMSC but handles multimedia.
- **Content Hosting:** The MMSC typically stores the multimedia content on a server and sends a notification message (an extended SMS) to the recipient's phone.
- **Retrieval:** The recipient's phone receives this notification and then connects to the MMSC (often via mobile data) to download the actual multimedia content. This is why you often need an active data connection to receive MMS.

Key Characteristics:

- **Multimedia Support:** Supports images, audio, video, and longer text.
- **Data Connection Dependent:** Requires a mobile data connection for content retrieval on the recipient's side.
- **Variable File Sizes:** Can handle larger file sizes than SMS, but still has operator-imposed limits.
- **Legacy Technology:** While still used, it's largely superseded by internet-based rich messaging (e.g., WhatsApp, iMessage) for most consumer use cases.
- **Cost:** Often more expensive than SMS or consumes data plan allowance.

Use Cases: MMS is primarily used today for:

- Sending photos and short videos to users who might not have internet-based messaging apps.
- Cross-platform multimedia communication when other rich messaging options aren't available.
- Some marketing campaigns that require rich media.

3. Push Notifications

Push notifications are real-time messages sent from an application server to a user's mobile device, even when the app is not actively running in the foreground. They are a powerful tool for re-engaging users, delivering timely information, and enhancing user experience.

How it Works (General Model):

- **Client Registration:** When an app is installed and opened, it registers with a platform-specific Push Notification Service (PNS) like Firebase Cloud Messaging (FCM) for Android or Apple Push Notification Service (APNs) for iOS. The PNS returns a unique "device token" or "registration ID" for that app instance on that device.
- **Token to App Server:** The app sends this device token to its own backend server (the "app server"). The app server stores this token, associating it with the specific user.

- **Sending a Push:** When the app server wants to send a notification to a user, it creates a message and sends it to the respective PNS (FCM or APNs), including the user's device token.
- **PNS Delivery:** The PNS takes the message, processes it, and "pushes" it to the target device via a persistent connection or other efficient mechanisms.
- **Device Reception:** The device receives the push notification. The operating system handles displaying it (e.g., in the notification shade, as a badge icon), and can also wake up the app (or a part of it) to process data associated with the notification.

Key Characteristics:

- **Real-time:** Designed for immediate delivery, making them ideal for time-sensitive alerts.
- **Internet Dependent:** Requires an active internet connection on the device to receive.
- **Rich Content:** Can include text, images, sound, action buttons, and can carry a data payload for the app to process internally.
- **Cost-Effective:** Typically free to send (e.g., FCM offers a generous free tier).
- **Cross-Platform Services:** Services like FCM abstract away the differences between Android and iOS push mechanisms, allowing a single backend to send to both.

Use Cases:

- **Social Media:** New messages, friend requests, likes, comments.
- **News Apps:** Breaking news alerts.
- **E-commerce:** Order updates, shipping notifications, promotional offers.
- **Productivity Apps:** Reminders, task assignments.
- **Gaming:** New game events, multiplayer invitations.
- **System Alerts:** Low battery warnings, software updates.

Understanding these core messaging technologies empowers developers to choose the right tool for the job, whether it's the ubiquitous reach of SMS, the multimedia capabilities of MMS, or the real-time, interactive power of push notifications

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 3.2: Messaging and Networking Service Integration on Android

Unit Objectives

By the end of this unit, the participants will be able to:

1. Explain how to integrate messaging services such as SMS and Firebase Cloud Messaging (FCM) into Android applications.
2. Demonstrate how to perform basic networking operations, including HTTP connections and RESTful API configuration.
3. Demonstrate the Bluetooth-based data exchange and device pairing.
4. Demonstrate techniques for managing data requests efficiently using synchronous, asynchronous, and caching methods.

3.2.1 Messaging Protocols and Technologies

Communication is at the heart of nearly every application, whether it's a simple text message or a complex real-time notification. Understanding the underlying technologies like SMS, MMS, and Push Notifications is crucial for any developer.

1. SMS (Short Message Service)

SMS, or Short Message Service, is the original text messaging standard for mobile phones. It allows users to send short text messages (typically up to 160 characters for GSM alphabets, or 70 characters for UCS-2 encoding which supports non-Latin characters) between mobile devices.

How it Works:

- **Store-and-Forward:** SMS operates on a "store-and-forward" mechanism. When you send an SMS, it goes from your phone to a Short Message Service Center (SMSC) operated by your mobile network provider.
- **Delivery Attempt:** The SMSC then attempts to deliver the message to the recipient's phone. If the recipient's phone is unreachable (e.g., off or out of coverage), the SMSC stores the message and retries delivery for a certain period.
- **Signaling Channels:** SMS messages often use the same signaling channels as voice calls, meaning they don't consume dedicated data bandwidth in the same way modern internet-based messaging does.

Key Characteristics:

- **Simplicity:** Very simple and widely supported across all mobile phones.
- **Reliability (within limits):** Generally reliable for delivery, but not guaranteed real-time.
- **Limited Length:** Restricted to short text messages.
- **No Internet Required (for basic sending/receiving):** Relies on cellular network signaling, not an active internet connection on the device.
- **Cost:** Usually incurs a small per-message cost, or is part of a bundled plan.

Use Cases: SMS is still heavily used for:

- Two-factor authentication (2FA) and OTPs (One-Time Passwords).
- Alerts and notifications from banks, airlines, etc.
- Basic peer-to-peer communication, especially in areas with limited internet access.
- Marketing and promotional messages (A2P - Application to Person)

2. MMS (Multimedia Messaging Service)

MMS, or Multimedia Messaging Service, is an extension of SMS that allows users to send and receive multimedia content. This includes images, audio clips, video clips, and longer text messages, which SMS alone cannot handle.

How it Works:

- **Content Packaging:** When you send an MMS, the content (image, video, text) is encoded and packaged.
- **MMS Center (MMSC):** This package is sent to a Multimedia Messaging Service Center (MMSC), which is similar in function to an SMSC but handles multimedia.
- **Content Hosting:** The MMSC typically stores the multimedia content on a server and sends a notification message (an extended SMS) to the recipient's phone.
- **Retrieval:** The recipient's phone receives this notification and then connects to the MMSC (often via mobile data) to download the actual multimedia content. This is why you often need an active data connection to receive MMS.

Key Characteristics:

- **Multimedia Support:** Supports images, audio, video, and longer text.
- **Data Connection Dependent:** Requires a mobile data connection for content retrieval on the recipient's side.
- **Variable File Sizes:** Can handle larger file sizes than SMS, but still has operator-imposed limits.
- **Legacy Technology:** While still used, it's largely superseded by internet-based rich messaging (e.g., WhatsApp, iMessage) for most consumer use cases.
- **Cost:** Often more expensive than SMS or consumes data plan allowance.

Use Cases: MMS is primarily used today for:

- Sending photos and short videos to users who might not have internet-based messaging apps.
- Cross-platform multimedia communication when other rich messaging options aren't available.
- Some marketing campaigns that require rich media directly to the phone's native messaging app.

3. Push Notifications

Push notifications are real-time messages sent from an application server to a user's mobile device, even when the app is not actively running in the foreground. They are a powerful tool for re-engaging users, delivering timely information, and enhancing user experience.

How it Works (General Model):

- **Client Registration:** When an app is installed and opened, it registers with a platform-specific Push Notification Service (PNS) like Firebase Cloud Messaging (FCM) for Android or Apple Push Notification Service (APNs) for iOS. The PNS returns a unique "device token" or "registration ID" for that app instance on that device.
- **Token to App Server:** The app sends this device token to its own backend server (the "app server"). The app server stores this token, associating it with the specific user.
- **Sending a Push:** When the app server wants to send a notification to a user, it creates a message and sends it to the respective PNS (FCM or APNs), including the user's device token.
- **PNS Delivery:** The PNS takes the message, processes it, and "pushes" it to the target device via a persistent connection or other efficient mechanisms.
- **Device Reception:** The device receives the push notification. The operating system handles displaying it (e.g., in the notification shade, as a badge icon), and can also wake up the app (or a part of it) to process data associated with the notification.

Key Characteristics:

- **Real-time:** Designed for immediate delivery, making them ideal for time-sensitive alerts.
- **Internet Dependent:** Requires an active internet connection on the device to receive.
- **Rich Content:** Can include text, images, sound, action buttons, and can carry a data payload for the app to process internally.
- **Cost-Effective:** Typically free to send (e.g., FCM offers a generous free tier).
- **Cross-Platform Services:** Services like FCM abstract away the differences between Android and iOS push mechanisms, allowing a single backend to send to both.

Use Cases:

- **Social Media:** New messages, friend requests, likes, comments.
- **News Apps:** Breaking news alerts.
- **E-commerce:** Order updates, shipping notifications, promotional offers.
- **Productivity Apps:** Reminders, task assignments.
- **Gaming:** New game events, multiplayer invitations.
- **System Alerts:** Low battery warnings, software updates.

Understanding these core messaging technologies empowers developers to choose the right tool for the job, whether it's the ubiquitous reach of SMS, the multimedia capabilities of MMS, or the real-time, interactive power of push notifications.

3.1.9 Implementing Content Providers to Enable Data Sharing Between Apps

Integrating robust messaging and networking capabilities is crucial for modern Android applications. Let's cover the key aspects, from push notifications to API consumption.

1. Configuring Messaging Functionalities with Firebase Cloud Messaging (FCM)

Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that lets you reliably send messages at no cost. You can use FCM to notify a client app that new email is available to sync, or to send promotional messages.

To get started, you'll need to set up Firebase in your Android project. This involves adding the Google services plugin and the FCM SDK to your build.gradle files.

Once configured, you can send messages from your server (or the Firebase console) to your Android app. Your app will need a service that extends `FirebaseMessagingService` to handle incoming messages.

Here's a basic idea of how the `FirebaseMessagingService` might look:

Java

```
import com.google.firebase.messaging.FirebaseMessagingService;
import com.google.firebase.messaging.RemoteMessage;
import android.util.Log;

public class MyFirebaseMessagingService extends FirebaseMessagingService {
```



```
private static final String TAG = "MyFirebaseMsgService";

@Override
public void onMessageReceived(RemoteMessage remoteMessage) {
    Log.d(TAG, "From: " + remoteMessage.getFrom());

    if (remoteMessage.getData().size() > 0) {
        Log.d(TAG, "Message data payload: " + remoteMessage.getData());
        // Handle data payload messages
    }

    if (remoteMessage.getNotification() != null) {
        Log.d(TAG, "Message Notification Body: " + remoteMessage.getNotification().getBody());
        // Handle notification payload messages
    }
}

@Override
public void onNewToken(String token) {
    Log.d(TAG, "Refreshed token: " + token);
    // Send this token to your app server.
}
}
```

2. Implementing SMS-Based Tasks Using BroadcastReceivers and Intents

Android provides mechanisms to send and receive SMS messages directly from your app, though with increasing restrictions for privacy and security. For basic SMS tasks, you'll primarily use `SmsManager` for sending and `BroadcastReceiver` for receiving.

Sending SMS: To send an SMS, you'll need the `SEND_SMS` permission in your `AndroidManifest.xml`. Then, you can use `SmsManager` like this:

```
Java
import android.telephony.SmsManager; import android.widget.Toast;
public void sendSms(String phoneNumber, String message) {
try {
    SmsManager smsManager = SmsManager.getDefault();
    smsManager.sendTextMessage(phoneNumber, null, message, null, null);
    Toast.makeText(getApplicationContext(), "SMS sent.", Toast.LENGTH_LONG).show();
} catch (Exception e) {
    Toast.makeText(getApplicationContext(), "SMS failed, please try again.",
    Toast.LENGTH_LONG).show();
    e.printStackTrace();
}
}
```

Receiving SMS: Receiving SMS requires the `RECEIVE_SMS` permission and a `BroadcastReceiver`. When an SMS arrives, the system broadcasts an `SMS_RECEIVED_ACTION` intent, which your receiver can catch.

```

@Override
public void onReceive(Context context, Intent intent) {
    if (intent.getAction().equals("android.provider.Telephony.SMS_RECEIVED")) {
        Bundle bundle = intent.getExtras();
        if (bundle != null) {
            Object[] pdus = (Object[]) bundle.get("pdus");
            if (pdus != null) {
                for (Object pdu : pdus) {
                    SmsMessage smsMessage = SmsMessage.createFromPdu((byte[]) pdu);
                    String sender = smsMessage.getDisplayOriginatingAddress();
                    String messageBody = smsMessage.getMessageBody();
                    Log.d(TAG, "SMS From: " + sender + ", Body: " + messageBody);
                    Toast.makeText(context, "SMS from " + sender + ": " + messageBody,
                        Toast.LENGTH_LONG).show();
                }
            }
        }
    }
}

```

Remember to declare your BroadcastReceiver in the AndroidManifest.xml.

3. Configuring Sample Email and HTTP Connections

For making HTTP requests, particularly to RESTful APIs, libraries like Retrofit are incredibly popular due to their type-safety and ease of use. Retrofit turns your HTTP API into a Java interface.

Setting up Retrofit: First, add the Retrofit and Gson converter dependencies to your build.gradle file.

Gradle

```
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
```

```
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
```

Define an interface that describes your API endpoints:

Java

```
import retrofit2.Call;import retrofit2.http.GET;import retrofit2.http.Path;
```

```
public interface GitHubService {
    @GET("users/{user}/repos")
    Call<List<Repo>> listRepos(@Path("user") String user);
}

```

Then, create a Retrofit instance and use it to create an implementation of your API interface:

Java

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

```
public class ApiClient {
    private static Retrofit retrofit = null;
    private static final String BASE_URL = "https://api.github.com/";

    public static Retrofit getClient() {
        if (retrofit == null) {
            retrofit = new Retrofit.Builder()
                .baseUrl(BASE_URL)
                .addConverterFactory(GsonConverterFactory.create())
                .build();
        }
    }
}

```

```

    }
    return retrofit;
  }
}

```

Making a request:

Java

```

import retrofit2.Call;import retrofit2.Callback;import retrofit2.Response;import java.util.List;
public void fetchUserRepositories(String username) {
    GitHubService service = ApiClient.getClient().create(GitHubService.class);
    Call<List<Repo>> call = service.listRepos(username);

    call.enqueue(new Callback<List<Repo>>() {
        @Override
        public void onResponse(Call<List<Repo>> call, Response<List<Repo>> response) {
            if (response.isSuccessful() && response.body() != null) {
                List<Repo> repos = response.body();
                // Process the list of repositories
            } else {
                // Handle error
            }
        }

        @Override
        public void onFailure(Call<List<Repo>> call, Throwable t) {
            // Handle network errors
        }
    });
}

```

For email connections, you would typically integrate with third-party email APIs (e.g., SendGrid, Mailgun) via HTTP requests, similar to the Retrofit example, or use JavaMail API for direct SMTP/IMAP connections (which is more complex for Android).`

4. Integrating Application Functionalities with Bluetooth

Bluetooth integration allows your Android app to communicate with other Bluetooth-enabled devices. This involves handling permissions, enabling Bluetooth, discovering devices, pairing, and managing connections for data transfer.

Permissions: You'll need `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions. For scanning and connecting to nearby devices, especially on Android 10+, you'll also need `ACCESS_FINE_LOCATION` or `BLUETOOTH_SCAN`, `BLUETOOTH_ADVERTISE`, and `BLUETOOTH_CONNECT` for Android 12+.

Enabling Bluetooth: You can prompt the user to enable Bluetooth if it's off:

Java

```

import android.bluetooth.BluetoothAdapter;import android.content.Intent;
private static final int REQUEST_ENABLE_BT = 1;
public void enableBluetooth() {
    BluetoothAdapter bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
    if (bluetoothAdapter == null) {
        // Device doesn't support Bluetooth
        return;
    }
}

```

```

}
if (!bluetoothAdapter.isEnabled()) {
    Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
}
}

```

Device Discovery and Pairing: You can start scanning for other Bluetooth devices or query already paired devices. Pairing typically involves displaying a UI for the user to confirm.

Data Sharing (Bluetooth Sockets): For data transfer, you'll use `BluetoothServerSocket` (for the server side) and `BluetoothSocket` (for the client side) to establish a connection using RFCOMM channels. `

5. Configuring RESTful APIs Using Standard Methods and Sample Code

Configuring RESTful APIs primarily involves understanding HTTP methods (GET, POST, PUT, DELETE), endpoints, request/response formats (usually JSON), and authentication. As shown in the Retrofit example (PC9), you define an interface where each method corresponds to an API endpoint and an HTTP action.

Example of an API Interface with various methods:

Java

```

import retrofit2.Call;import retrofit2.http.Body;import retrofit2.http.DELETE;import
retrofit2.http.GET;import retrofit2.http.POST;import retrofit2.http.PUT;import retrofit2.http.Path;
public interface SampleApiService {

    // GET request to fetch a list of items
    @GET("items")
    Call<List<Item>> getItems();

    // GET request to fetch a specific item by ID
    @GET("items/{id}")
    Call<Item> getItem(@Path("id") String itemId);

    // POST request to create a new item
    @POST("items")
    Call<Item> createItem(@Body Item newItem);
    // PUT request to update an existing item
    @PUT("items/{id}")
    Call<Item> updateItem(@Path("id") String itemId, @Body Item updatedItem);

    // DELETE request to remove an item
    @DELETE("items/{id}")
    Call<Void> deleteItem(@Path("id") String itemId);
}

```

This interface clearly defines how your application interacts with a RESTful service, adhering to standard HTTP methods for common operations (Create, Read, Update, Delete - CRUD). The `@Body` annotation is used to send data in the request body, typically as JSON. `

6. Implementing Efficient Data Requests

Let us now cover the implementation of efficient data requests using Synchronous and asynchronous calls, and application of basic caching techniques to support optimal resource usage.

Asynchronous vs. Synchronous Calls: On Android, you must perform network operations on a background thread to avoid blocking the UI thread (main thread) and causing Application Not Responding (ANR) errors. This means using asynchronous calls.

- **Asynchronous:** The call doesn't block the current thread. It returns immediately, and the result is delivered via a callback mechanism once the operation completes (e.g., Retrofit's enqueue method, AsyncTask, Kotlin Coroutines, RxJava). This is the standard for Android networking.
- Java
- // Retrofit's enqueue is asynchronous
- `call.enqueue(new Callback<List<Repo>>() { /* ... */ });`
- **Synchronous (blocking):** The call waits for the operation to complete before returning. This is generally avoided on the main thread but can be used on a dedicated background thread.
- Java
- `// Don't run on main thread!// Response<List<Repo>> response = call.execute(); // Synchronous call`
- **Basic Caching Techniques:** Caching data can significantly improve performance, reduce network usage, and provide a better user experience (e.g., showing stale data instead of a blank screen when offline).
- **HTTP Caching (with OkHttp):** Retrofit uses OkHttp underneath, which supports HTTP caching. You can configure an OkHttpClient with a Cache.
- Java
- `import okhttp3.Cache;import okhttp3.OkHttpClient;import java.io.File;import java.util.concurrent.TimeUnit;`
- `// ... in your ApiClient setupint cacheSize = 10 * 1024 * 1024; // 10 MiB`
- `File httpCacheDirectory = new File(context.getCacheDir(), "http-cache");`
- `Cache myCache = new Cache(httpCacheDirectory, cacheSize);`

`OkHttpClient okHttpClient = new OkHttpClient.Builder()`

- `.cache(myCache)`
- `.readTimeout(30, TimeUnit.SECONDS)`
- `.connectTimeout(30, TimeUnit.SECONDS)`
- `.build();`

`retrofit = new Retrofit.Builder()`

- `.baseUrl(BASE_URL)`
- `.addConverterFactory(GsonConverterFactory.create())`
- `.client(okHttpClient) // Set the custom OkHttpClient`
- `.build();`
- **In-Memory Caching:** Store frequently accessed data in memory (e.g., using LruCache or a simple HashMap). This is fast but volatile.
- **Disk Caching (Persistence):** Store data on disk for longer-term persistence. This can be done using:
 - **Room Database:** For structured data, Room is Google's recommended persistence library.
 - **Shared Preferences:** For small amounts of key-value data.
 - **Files:** For larger, unstructured data (e.g., images, large JSON blobs).

By combining asynchronous requests with appropriate caching strategies, you can build responsive and efficient Android applications.

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 3.3: Fundamentals of Google Maps Integration and Location-Based Services

Unit Objectives

By the end of this unit, the participants will be able to:

1. Explain the process of configuring Google Maps API for location-based functionalities.
2. Describe the procedure for setting and managing device or application location settings.
3. Demonstrate geocoding and reverse-geocoding using standard tools or utilities.
4. Demonstrate the configuration of GPS and Fused Location Provider for accurate positioning.
5. Elaborate on methods to test and report the performance of location-based services

3.3.1 Google Maps Integration

Integrating Google Maps into an Android application allows you to display interactive maps, show specific locations, draw routes, and add custom markers or overlays.

- **API Key:** At its core, Google Maps requires an API Key. This key authenticates your app with Google's services and tracks usage. It must be securely stored (e.g., in AndroidManifest.xml or local.properties) and restricted to prevent unauthorized use.
- **Support Map Fragment / Map View:** These are the primary UI components for displaying a map.
- **Support Map Fragment:** A Fragment that hosts a map. Recommended for most cases as it integrates well with the activity lifecycle.
- **Map View:** A View that displays a map. More flexible but requires manual lifecycle management (passing Activity lifecycle events to MapView).
- **Google Map Object:** Once the map UI component is ready, you obtain a GoogleMap object. This is the main class you interact with to control map properties (e.g., camera position, zoom, map type) and add elements (markers, polylines, polygons).
- **On Map Ready Call back:** An interface implemented by your activity or fragment. The onMapReady(GoogleMap googleMap) method is called when the map is ready to be used, providing you with the GoogleMap instance.
- **Markers:** Represent specific points of interest on the map. You can customize their icons and add info windows.
- **Camera Control:** The map's visible area is controlled by the camera. You can programmatically move, zoom, tilt, and rotate the camera using CameraUpdateFactory and animateCamera() or moveCamera().
- **Location-Based Services (LBS):** LBS refers to services that use location data to provide information or functionality tailored to a user's geographical position.
- **Permissions:** Accessing a user's location is a sensitive operation and requires explicit runtime permissions from the user.
 - ACCESS_FINE_LOCATION: Provides precise location (GPS, Wi-Fi, cellular).
 - ACCESS_COARSE_LOCATION: Provides approximate location (Wi-Fi, cellular).

Location Providers: Different technologies can determine location:

- **GPS (Global Positioning System):** Provides highly accurate location outdoors, but consumes more battery and doesn't work well indoors.
- **Network Location (Wi-Fi and Cellular):** Less accurate but faster and works better indoors. Consumes less battery.

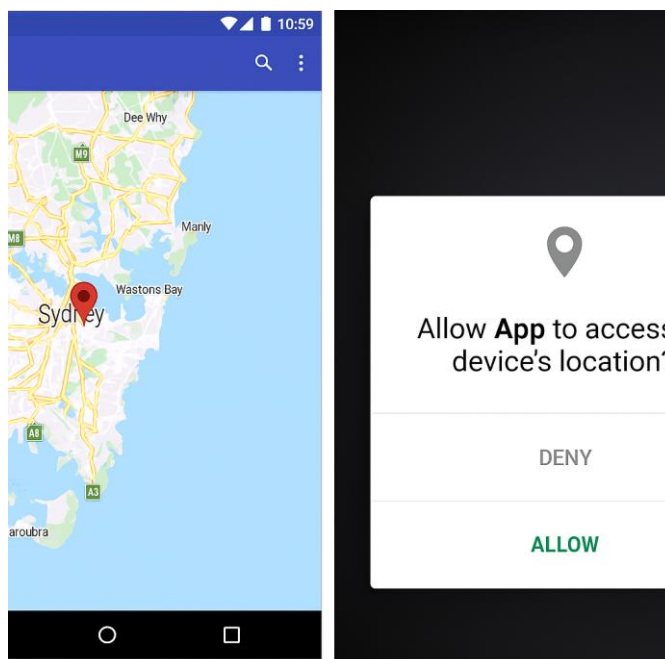
Location-Based Services (LBS): LBS refers to services that use location data to provide information or functionality tailored to a user's geographical position.

- **Permissions:** Accessing a user's location is a sensitive operation and requires explicit runtime permissions from the user.
- **ACCESS_FINE_LOCATION:** Provides precise location (GPS, Wi-Fi, cellular).
- **ACCESS_COARSE_LOCATION:** Provides approximate location (Wi-Fi, cellular).
- **Location Providers:** Different technologies can determine location:
- **GPS (Global Positioning System):** Provides highly accurate location outdoors, but consumes more battery and doesn't work well indoors.
- **Network Location (Wi-Fi and Cellular):** Less accurate but faster and works better indoors. Consumes less battery.
- **Location Settings Dialog:** Android devices often have location services turned off or in a low-accuracy mode. The Fused Location Provider can help prompt the user to enable/adjust these settings for optimal app functionality.

3.3.2 Simple RESTful API Request Structure and Network Protocols

REST (Representational State Transfer): REST is an architectural style for networked applications. RESTful APIs are web services that adhere to the REST principles, making them scalable, flexible, and widely used for communication between client (e.g., Android app) and server.

- **Resources:** Everything in a REST API is considered a "resource" (e.g., a user, a product, an order). Resources are identified by URLs (Uniform Resource Locators).
- **Example:** /users, /products/123, /orders
- **Statelessness:** Each request from the client to the server must contain all the information needed to understand the request. The server should not store any client context between requests.
- **HTTP Methods (Verbs):** Standard HTTP methods are used to perform actions on resources:



GET: Retrieve data from a resource. (Idempotent and safe)

Example: GET /products (get all products), GET /products/123 (get product with ID 123)

POST: Create a new resource or submit data. (Not idempotent)

Example: POST /products (create a new product, data in request body)

PUT: Update an existing resource (replace the entire resource). (Idempotent)

Example: PUT /products/123 (update product 123 with new data)

DELETE: Remove a resource. (Idempotent)

Example: DELETE /products/123 (delete product with ID 123)

PATCH: Partially update an existing resource. (Not idempotent)

Request/Response Structure:

- **Request:** Consists of:
 - **Method:** (GET, POST, PUT, DELETE, etc.)
 - **URL/Endpoint:** Identifies the resource.
 - **Headers:** Metadata (e.g., Content-Type, Authorization for API keys/tokens).
 - **Body** (for POST/PUT/PATCH): The data being sent, typically in JSON format.

Response:

- **Status Code:** Indicates the result of the request (e.g., 200 OK, 201 Created, 404 Not Found, 500 Internal Server Error).
- **Headers:** Metadata about the response.
- **Body:** The data returned by the server, typically in JSON format.

Network Protocols:

- **HTTP (Hypertext Transfer Protocol):** The foundation of data communication for the World Wide Web. RESTful APIs primarily use HTTP.
- **HTTPS (Hypertext Transfer Protocol Secure):** The secure version of HTTP. It encrypts communication between the client and server using SSL/TLS, protecting data from eavesdropping and tampering. Always use HTTPS for production applications, especially when dealing with sensitive data.
- **JSON (JavaScript Object Notation):** A lightweight data-interchange format. It's human-readable and easy for machines to parse. It's the de-facto standard for data exchange in RESTful APIs due to its simplicity and efficiency compared to XML.

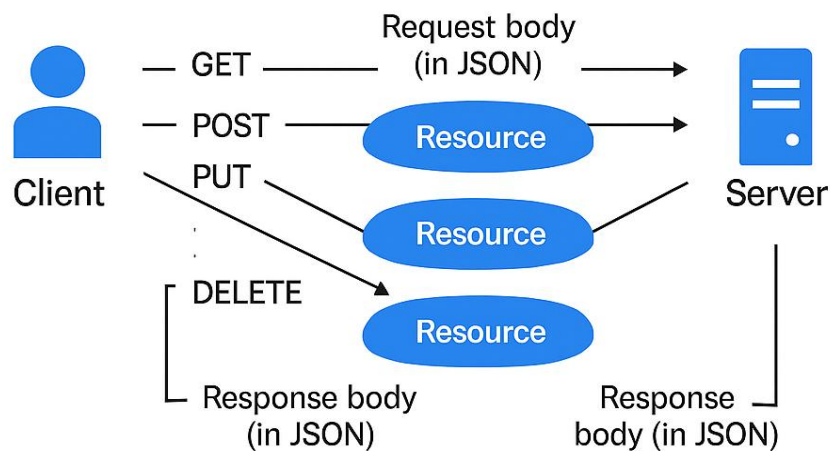


Fig 3.3. Client-server interaction with HTTP methods

3.3.3 Principles of Geocoding and Reverse Geocoding

Geocoding: Geocoding is the process of converting a human-readable address or place name into geographical coordinates (latitude and longitude).

- **Input:** A string representing an address (e.g., "Eiffel Tower, Paris", "1600 Amphitheatre Parkway, Mountain View, CA").
- **Output:** A LatLng object or similar structure containing latitude and longitude values.
- **How it works:** A geocoding service (like Google's Geocoding API or Android's built-in Geocoder) parses the input address, compares it against a database of geographical locations, and returns the most probable coordinates.
- **Accuracy and Ambiguity:** Geocoding can sometimes be ambiguous (e.g., "Springfield" exists in many states). Services often return multiple possible results or a single "best guess" with a confidence score.
- **Use Cases:**
 - Placing a marker on a map for a user-entered address.
 - Storing coordinates for user-defined locations.
 - Calculating distances or routes between named places.

Reverse Geocoding: Reverse geocoding is the opposite process: converting geographical coordinates (latitude and longitude) into a human-readable address or place description.

- **Input:** latitude and longitude values.
- **Output:** A structured address object (e.g., Address object in Android) containing components like street number, street name, city, state, country, postal code, etc

How it works: A reverse geocoding service takes coordinates, looks up geographical features at or near those coordinates in its database, and returns the corresponding address components.

Granularity: The level of detail in the returned address can vary (e.g., sometimes it's just a city, sometimes a full street address).

Use Cases:

- Displaying the current street address for a user's location on a map.
- Showing the address of a point the user tapped on a map.
- Tagging photos or posts with their physical location address.

3.3.4 Configuring Location-Based Services on Android

Location-based services are fundamental for many modern Android applications, from navigation to social networking. This guide will walk you through setting up Google Maps, handling location, geocoding, and utilizing the Fused Location Provider.

Setting Up Google Maps API and Configuring Location Settings

Setting up Google Maps API involves several steps, including obtaining an API key and configuring your Android project.

and configuring your Android project.

1. Obtain a Google Maps API Key:

- Go to the Google Cloud Console.
- Create a new project or select an existing one.
- Enable the "Maps SDK for Android" API.
- Create API credentials (an API key) and restrict it to your Android app (using package name and SHA-1 certificate fingerprint) for security.

2. Add Dependencies to build.gradle: In your app-level build.gradle file, add the Maps SDK dependency:

```
Gradle
dependencies {
```

3. Add API Key to AndroidManifest.xml: Inside the <application> tag, add a <meta-data> tag with your API key:

XML

```
<application ...>
  <meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="YOUR_API_KEY"/>
</application>
```

Replace YOUR_API_KEY with the key you obtained.

4. Request Location Permissions: To access the user's location, you need to declare appropriate permissions in AndroidManifest.xml.

XML

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

ACCESS_FINE_LOCATION provides more precise location data (GPS), while ACCESS_COARSE_LOCATION provides less precise data (Wi-Fi, cellular). You'll also need to request these permissions at runtime in your activity.

5. Add a Map Fragment to Your Layout: In your XML layout file (e.g., activity_main.xml), add a SupportMapFragment:

XML

```
<fragment
  android:id="@+id/map"
  android:name="com.google.android.gms.maps.SupportMapFragment"
  android:layout_width="match_parent"
  android:layout_height="match_parent" />
```

6. Initialize and Use the Map in Your Activity: In your Activity or Fragment, you'll get a reference to the map fragment and then initialize the map.

Java

```
Import androidx.appcompat.app.AppCompatActivity;import android.os.Bundle;import
com.google.android.gms.maps.CameraUpdateFactory;import
com.google.android.gms.maps.GoogleMap;import
com.google.android.gms.maps.OnMapReadyCallback;import
com.google.android.gms.maps.SupportMapFragment;import
com.google.android.gms.maps.model.LatLng;import
com.google.android.gms.maps.model.MarkerOptions;
public class MapsActivity extends AppCompatActivity implements OnMapReadyCallback {
  private GoogleMap mMap;
```

```
@Override
```

```
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_maps);
```

```
    SupportMapFragment mapFragment = (SupportMapFragment)
    getSupportFragmentManager()
        .findFragmentById(R.id.map);
    mapFragment.getMapAsync(this);
}
```

GET: Retrieve data from a resource. (Idempotent and safe)

Example: GET /products (get all products), GET /products/123 (get product with ID 123)

POST: Create a new resource or submit data. (Not idempotent)

Example: POST /products (create a new product, data in request body)

PUT: Update an existing resource (replace the entire resource). (Idempotent)

Example: PUT /products/123 (update product 123 with new data)

DELETE: Remove a resource. (Idempotent)

Example: DELETE /products/123 (delete product with ID 123)

PATCH: Partially update an existing resource. (Not idempotent)

Request/Response Structure:

- **Request:** Consists of:
- **Method:** (GET, POST, PUT, DELETE, etc.)
- **URL/Endpoint:** Identifies the resource.
- **Headers:** Metadata (e.g., Content-Type, Authorization for API keys/tokens).
- **Body** (for POST/PUT/PATCH): The data being sent, typically in JSON format.

Response: Consists of:

- **Status Code:** Indicates the result of the request (e.g., 200 OK, 201 Created, 404 Not Found, 500 Internal Server Error).
- **Headers:** Metadata about the response.
- **Body:** The data returned by the server, typically in JSON format.

Network Protocols:

- **HTTP (Hypertext Transfer Protocol):** The foundation of data communication for the World Wide Web. RESTful APIs primarily use HTTP.
- **HTTPS (Hypertext Transfer Protocol Secure):** The secure version of HTTP. It encrypts communication between the client and server using SSL/TLS, protecting data from eavesdropping and tampering. Always use HTTPS for production applications, especially when dealing with sensitive data.
- **JSON (JavaScript Object Notation):** A lightweight data-interchange format. It's human-readable and easy for machines to parse. It's the de-facto standard for data exchange in RESTful APIs due to its simplicity and efficiency compared to XML.

3.3.5 Geocoding Using Pre-built Tools or Utilities

Geocoding is the process of converting a human-readable address (like "1600 Amphitheatre Parkway, Mountain View, CA") into geographical coordinates (latitude and longitude), and reverse geocoding is the opposite – converting coordinates into an address. Android provides a built-in Geocoder class for this.

1. Using the Geocoder Class:

Java

```
import android.content.Context; import android.location.Address; import
android.location.Geocoder; import android.util.Log; import java.io.IOException; import
java.util.List; import java.util.Locale;
public class GeocodingHelper {
    private static final String TAG = "GeocodingHelper";
    // Geocoding: Address to LatLng
    public void getLatLngFromAddress(Context context, String addressString) {
        Geocoder geocoder = new Geocoder(context, Locale.getDefault());
        try {
```

```

List<Address> addresses = geocoder.getFromLocationName(addressString, 1);
if (addresses != null && !addresses.isEmpty()) {
    Address address = addresses.get(0);
    double latitude = address.getLatitude();
    double longitude = address.getLongitude();
    Log.d(TAG, "Address: " + addressString + " -> Lat: " + latitude + ", Lng: " + longitude);
} else {
    Log.d(TAG, "No results found for address: " + addressString);
}
} catch (IOException e) {
    Log.e(TAG, "Error getting LatLng from address: " + e.getMessage());
}
}

// Reverse Geocoding: LatLng to Address
public void getAddressFromLatLng(Context context, double latitude, double longitude) {
    Geocoder geocoder = new Geocoder(context, Locale.getDefault());
    try {
        List<Address> addresses = geocoder.getFromLocation(latitude, longitude, 1);
        if (addresses != null && !addresses.isEmpty()) {
            Address address = addresses.get(0);
            StringBuilder addressDetails = new StringBuilder();
            for (int i = 0; i <= address.getMaxAddressLineIndex(); i++) {
                addressDetails.append(address.getAddressLine(i)).append("\n");
            }
            Log.d(TAG, "LatLng: " + latitude + ", " + longitude + " -> Address: " +
addressDetails.toString());
        } else {
            Log.d(TAG, "No address found for LatLng: " + latitude + ", " + longitude);
        }
    } catch (IOException e) {
        Log.e(TAG, "Error getting address from LatLng: " + e.getMessage());
    }
}
}

```

Note: Geocoder requires an internet connection and might not always return results, or might be rate-limited by the device. For production apps or high-volume geocoding, consider using Google Maps Geocoding API directly via HTTP requests.

3.3.6 Configuring GPS Settings and Fused Location Provider Integration

4. Request Regular Location Updates: For continuous updates, you need to create a `LocationRequest` and a `LocationCallback`.

Java

```
import com.google.android.gms.location.LocationCallback;import
com.google.android.gms.location.LocationRequest;import
com.google.android.gms.location.LocationResult;import com.google.android.gms.location.Priority; //
For newer Play Services versions
private LocationRequest locationRequest;private LocationCallback locationCallback;
private void createLocationRequest() {
    locationRequest = new LocationRequest.Builder(Priority.PRIORITY_HIGH_ACCURACY, 10000) // 10
seconds interval
        .setMinUpdateIntervalMillis(5000) // Smallest interval for updates
        .setMaxUpdateDelayMillis(15000) // Max wait time for updates
        .build();
}
private void createLocationCallback() {
    locationCallback = new LocationCallback() {
        @Override
        public void onLocationResult(LocationResult locationResult) {
            if (locationResult == null) {
                return;
            }
            for (Location location : locationResult.getLocations()) {
                // Update UI with location data
                Log.d(TAG, "Location update: " + location.getLatitude() + ", " + location.getLongitude());
            }
        }
    };
}
public void startLocationUpdates() {
    if (ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) !=
PackageManager.PERMISSION_GRANTED &&
        ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_COARSE_LOCATION) !=
PackageManager.PERMISSION_GRANTED) {
        // Request permissions
        return;
    }
    createLocationRequest();
    createLocationCallback();
    fusedLocationClient.requestLocationUpdates(locationRequest, locationCallback, getMainLooper());
}
public void stopLocationUpdates() {
    fusedLocationClient.removeLocationUpdates(locationCallback);
}
@Overrideprotected void onResume() {
    super.onResume();
    startLocationUpdates(); // Start updates when activity resumes
}
@Overrideprotected void onPause() {
    super.onPause();
    stopLocationUpdates(); // Stop updates when activity pauses to save battery
```

3.3.7 Testing Location-Based Service Functionality

Thorough testing is crucial for location-based services due to the variability of real-world conditions. Testing Scenarios:

1. Permissions:

- Test with permissions granted, denied, and "Ask every time."
- Verify app behavior when permissions are revoked mid-use.

2. Location Settings (GPS/Wi-Fi/Network):

- High Accuracy: GPS, Wi-Fi, and mobile networks are all active.
- Battery Saving: Wi-Fi and mobile networks are used, but not GPS.
- Device Only/GPS Only: Only GPS is used.
- Location Off: Verify the app prompts the user to enable location services.

3. Network Conditions:

- Online (Wi-Fi/Mobile Data): Verify map loading, geocoding, and location updates.
- Offline/No Network: Check how the app handles map

4. Movement and Environment:

- Stationary: Test accuracy when the device is still.
- Moving (walking, driving): Test update frequency and accuracy.
- Indoors/Outdoors: GPS signals are weaker indoors. Test how the Fused Location Provider adapts.
- Urban Canyon/Open Area: Test signal strength and accuracy in different environments.

5. Background Operation:

- Test location updates when the app is in the background (mind Android's background location limits for battery saving).

6. Device State:

- Low Battery: Observe how the Fused Location Provider changes its behavior to save power.
- Airplane Mode: Verify location services are disabled.

7. Reporting Functionality:

For each test scenario, document:

- Test Case: Clear description of what is being tested.
- Preconditions: Required settings (e.g., GPS enabled, network on).
- Steps: Actions taken to perform the test.
- Expected Result: How the app should behave.
- Actual Result: How the app actually behaved.
- Status: Pass/Fail.
- Notes/Observations: Any deviations, errors, or performance issues.
- Screenshots/Logs: Include relevant visual proof or logcat output.

Tools like the Android Emulator's extended controls (for simulating location) and real-world testing are essential to ensure a robust location-based experience

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 3.4: Background Android Services

Unit Objectives

By the end of this unit, the participants will be able to:

1. Explain the types of Android services (foreground, background, bound) and their appropriate use cases.
2. Demonstrate the setup of background tasks using WorkManager or JobScheduler as per given instructions.
3. Describe standard procedures for handling recurring background tasks and monitoring their performance impact.
4. Demonstrate data sharing techniques between activities and background services.
5. Explain memory management and UI update practices using tools such as ViewModel and LiveData..

3.4.1 Fundamentals of Background Processing: Multi-threading, WorkManager, and JobScheduler

Background processing is essential for performing long-running tasks or operations that shouldn't block the main UI thread, ensuring a smooth and responsive user experience.

1. Multi-threading (The Core Concept): At its heart, background processing on Android involves multi-threading. The UI thread (also known as the main thread) is responsible for handling user interface updates and input events. Any operation that takes too long on this thread will cause the app to freeze and potentially result in an Application Not Responding (ANR) error.

- **Worker Threads:** To avoid blocking the UI thread, long-running tasks (like network requests, heavy computations, database operations, or file I/O) should be executed on separate "worker threads" or "background threads."
- **Thread Pools:** Managing individual threads can be complex. Thread pools (e.g., `ExecutorService`) are often used to efficiently manage a group of worker threads, reusing them for multiple tasks.
- **Communication:** Once a background task is complete, its result often needs to be communicated back to the UI thread to update the UI. Handlers, `post()`, or various concurrency utilities facilitate this.

2. Work Manager (Recommended for Persistent Background Work): WorkManager is a part of Android Jetpack and is the recommended solution for deferrable, guaranteed background work. It's suitable for tasks that need to run reliably even if the app exits or the device restarts.

- **Guaranteed Execution:** WorkManager ensures your tasks run, even if system constraints are not met immediately. It handles rescheduling work after device restarts.
- **Constraints:** You can define constraints for when the work should run, such as:
 - Network availability (e.g., `NetworkType.CONNECTED`, `NetworkType.UNMETERED`)
 - Device charging state (`Charging.REQUIRED`)
 - Device idle state (`Idle.REQUIRED`)
 - Storage low (`StorageNotLow.REQUIRED`)
- **Types of Work:**
 - One Time Work Request: For tasks that run once.
 - Periodic Work Request: For tasks that repeat at regular intervals.

Underlying APIs: Work Manager intelligently chooses the appropriate underlying API based on device API level and app state (e.g., `JobScheduler`, `AlarmManager` + `BroadcastReceiver`). This abstraction simplifies development.

- Use Cases:
- Syncing data with a server.
- Uploading logs or analytics.
- Applying filters to images and saving them.
- Sending periodic notifications.

3. JobScheduler (System Service for Flexible, Opportunistic Work - Pre-WorkManager): Introduced in API 21 (Lollipop), JobScheduler is a system service that allows you to schedule various types of jobs to be executed in batches, under specified conditions, for optimal battery performance. While WorkManager is now the preferred choice, understanding JobScheduler's principles is still valuable as WorkManager often uses it internally.

- System-Managed Batching: JobScheduler batches similar jobs from multiple apps, optimizing system resource usage and battery life.
- Flexible Conditions: Like WorkManager, it allows you to define conditions for when a job should run:
 - Network type (e.g., Wi-Fi, cellular)
 - Device charging status
 - Device idle state
 - Minimum latency or deadline
- No Guarantee for Immediate Execution: Jobs are opportunistic; the system decides the best time to run them based on the specified conditions and overall device state.
- Use Cases (where JobScheduler excels, though WorkManager provides a higher-level API):
 - Downloading app updates.
 - Backing up data.
 - Processing image queues when on charge and Wi-Fi.

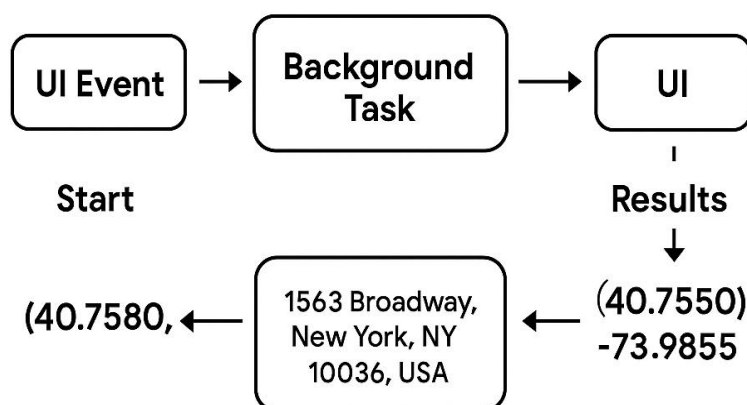


Fig 3.4. Fundamentals of Background Processing

3.4.2 Efficient Resource Management for Network, Storage, and Memory Optimization

Efficient resource management is crucial for creating high-performing, stable, and battery-friendly Android applications.

1. Network Optimization:

Reduce Data Transfer:

- **Compress Data:** Use GZIP or other compression algorithms for network requests and responses.
- **Cache Data:** Implement HTTP caching (e.g., with OkHttp) or store frequently accessed data locally (see Storage Optimization).
- **Partial Updates:** Request only the necessary data instead of entire objects (PATCH requests instead of PUT for partial updates).
- **Lazy Loading:** Load images and other media only when they are visible on the screen.
- **Optimize Request Frequency:**
- **Batch Requests:** Combine multiple small requests into one larger request.
- **Debounce/Throttle:** Limit the rate of continuous requests (e.g., for search suggestions).
- **Pre-fetching:** Download data proactively when you anticipate it will be needed soon (e.g., next page in a list).
- **Handle Network Changes:** Listen for network connectivity changes and adjust network operations accordingly (e.g., defer large downloads to Wi-Fi).
- **Use HTTPS:** Not just for security, but also for performance benefits from HTTP/2 multiplexing.

2. Storage Optimization:

Choose the Right Storage:

- **SharedPreferences:** For small amounts of primitive data (settings, preferences).
- **Internal/External Storage:** For files (images, large data blobs).
- **Room Database:** For structured, relational data. Efficient querying and schema management.

Minimize Storage Footprint:

- **Compress Data:** Compress large files (images, JSON) before saving.
- **Delete Unused Data:** Periodically clean up temporary files or old caches.
- **Avoid Duplication:** Don't store the same data multiple times.
- **Disk I/O:** Perform disk I/O operations (reading/writing files, database transactions) on background threads to avoid blocking the UI.
- **Caching:** Use a robust caching strategy (e.g., LruCache for memory, DiskLruCache for disk) to avoid re-downloading or re-processing data.

3. Memory Optimization:

Avoid Memory Leaks:

- **Static Contexts:** Be cautious with static references to Context (especially Activity contexts), as they can prevent activities from being garbage collected. Use applicationContext or WeakReference where appropriate.
- **Inner Classes:** Non-static inner classes implicitly hold a reference to their outer class. If they outlive the outer class, they can cause leaks

- **Register/Unregister Listeners:** Always unregister listeners (e.g., BroadcastReceivers, sensor listeners) when they are no longer needed, typically in onPause() or onDestroy().

Reduce Memory Usage:

- **Optimize Bitmaps:** Decode bitmaps efficiently (e.g., scale them down to the target ImageView size using inSampleSize). Release bitmap memory when no longer needed (bitmap.recycle()).
- **Object Pools:** Reuse objects instead of creating new ones constantly, reducing garbage collection overhead.
- **Sparse Arrays/ArrayMaps:** Use SparseArray, SparseBooleanArray, LongSparseArray, or ArrayMap instead of HashMap for mapping primitives to objects or objects to objects, respectively, as they are more memory-efficient for smaller collections.
- **Limit Background Processes:** Be mindful of services and background threads that consume memory even when the app is not in the foreground.
- **Profiling:** Use Android Studio's Profiler (Memory Profiler) to identify memory leaks, excessive allocations, and overall memory usage.

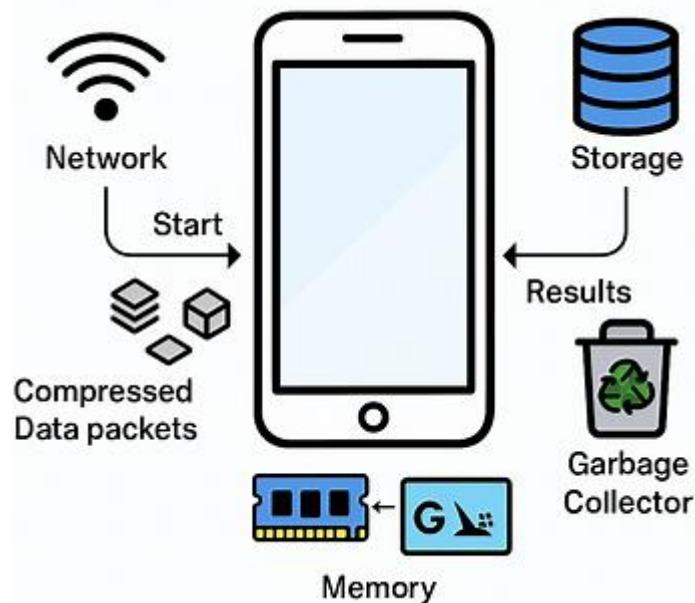


Fig 3.4. Data Flow and Optimization points

3.4.3 Security Considerations in Android Applications

Security is paramount in Android development to protect user data and maintain trust.

1. Data Encryption: Protecting sensitive data at rest and in transit is critical.

- **Encryption at Rest (Local Storage):**
- **KeyStore:** Android's KeyStore system is used to generate and store cryptographic keys in a secure, hardware-backed location (if available). Keys stored here are resistant to extraction.
- **Encrypted Shared Preferences:** Part of AndroidX Security library, this provides a secure way to store Shared Preferences by encrypting keys and values.

- **SQLCipher / Room with Encryption:** For encrypting databases. Room itself doesn't offer built-in encryption, but you can integrate with libraries like SQLCipher for encrypted Room databases.
- **File Encryption:** For larger files, you can encrypt/decrypt them using standard Java cryptographic APIs (e.g., AES) and keys from KeyStore.

Encryption in Transit (Network Communication):

- **HTTPS:** Always use HTTPS for all network communication. This encrypts data between your app and the server, preventing man-in-the-middle attacks. Ensure your app properly validates SSL certificates.
- **Certificate Pinning:** For extremely sensitive communication, you can implement certificate pinning, which means your app will only trust a specific set of server certificates, even if the device's trust store contains others. This defends against compromised Certificate Authorities.
- **Authentication:** Verifying a user's identity is a fundamental security requirement.
- **Strong Passwords & Hashing:** If your app handles user accounts, ensure password storage uses strong, industry-standard hashing algorithms (e.g., bcrypt, scrypt) with salts. Never store plaintext passwords.
- **Token-Based Authentication (OAuth2/JWT):**
 - After initial login, the server issues a short-lived access token (e.g., JWT).
 - The app stores this token securely (e.g., EncryptedSharedPreferences).
 - Subsequent API requests include this token in the Authorization header.
 - Refresh tokens can be used to obtain new access tokens without re-authenticating the user.
- **Biometric Authentication (BiometricPrompt):** Allow users to authenticate using fingerprint or face unlock. This provides a convenient and secure way to access protected parts of your app.
- **Google Sign-In / Firebase Authentication:** Leverage existing robust authentication solutions that handle much of the complexity securely.
- **Session Management:** Implement secure session management on your backend, invalidate sessions on logout, and set appropriate token expiration times.
- **Secure APIs:** Designing and interacting with APIs securely is vital.
- **API Key Protection:** Never embed sensitive API keys directly in your code that are not meant for public access. Store them securely (e.g., in local.properties and inject them via Gradle, or retrieve them from a secure backend at runtime).
- **Input Validation:** Always validate all data received from APIs (and from users) to prevent injection attacks (SQL injection, XSS) and unexpected behavior.
- **Least Privilege:** Your app should only request the minimum necessary permissions and API scopes required for its functionality.
- **API Rate Limiting:** Implement rate limiting on your server-side APIs to prevent abuse (e.g., brute-force attacks, denial-of-service).
- **Tampering Detection:** Implement measures to detect if your app has been tampered with (e.g., root detection, code integrity checks), although these can be bypassed by sophisticated attackers.
- **ProGuard/R8:** Use code obfuscation and shrinking tools (enabled by default in release builds) to make reverse engineering harder, though it's not a primary security measure.

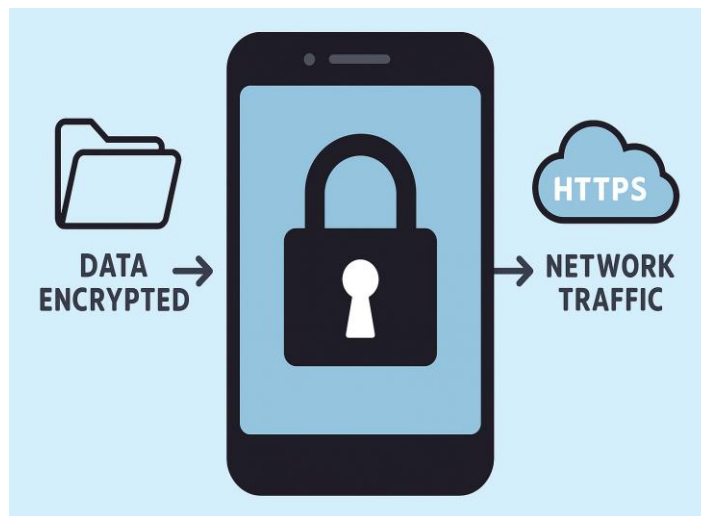


Fig 3.4. Data Encryption

3.4.4 Basic Service Types and When to Use Them

In Android, Services are components that perform long-running operations in the background without providing a user interface. There are three main types of services: Foreground, Background, and Bound. Each serves specific use cases based on the task's requirements.

Service Types and Use Cases

1. Foreground Services:

- Definition: A service that runs with a visible notification to the user, indicating ongoing work (e.g., music playback, location tracking).

Use Case: Tasks requiring user awareness, such as playing music or tracking a workout.

- Characteristics:
- Must display a notification.
- High priority, less likely to be killed by the system.
- Requires explicit user permission on modern Android versions (e.g., Android 9+).
- Example: A music app playing songs in the background.

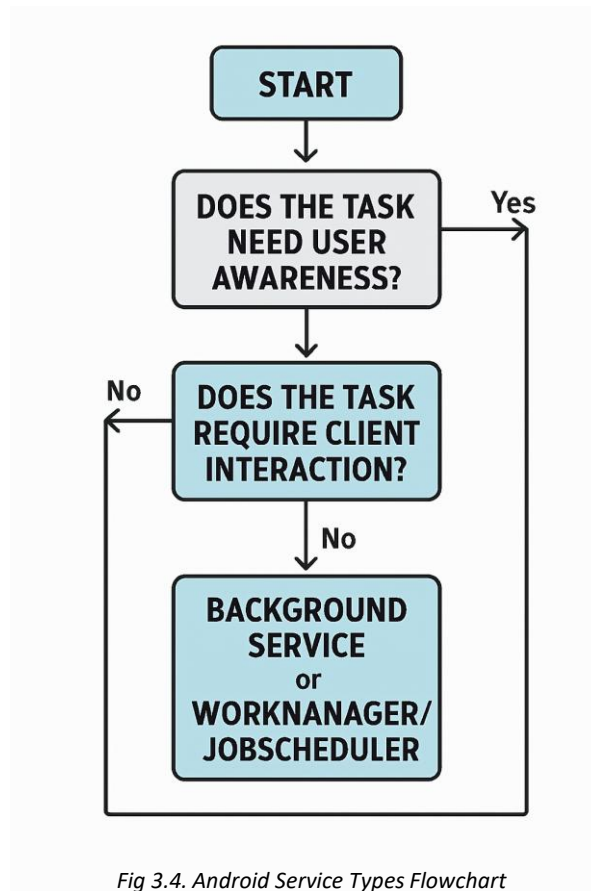
Background Services:

- Definition: Services that run without direct user interaction or visibility, typically for short-lived tasks.
- Use Case: Tasks like syncing data or processing small computations that don't require user interaction.
- Characteristics:
- No notification required (pre-Android Oreo).
- Restricted in modern Android (post-Oreo) due to battery optimization; use WorkManager or Job Scheduler instead for most cases.
- Example: Uploading a file to a server in the background.

When to Use Each

- Foreground: Use for tasks that must continue even when the app is not in focus and require user awareness (e.g., navigation apps).
- Background: Use for non-user-facing tasks that are short-lived or deferrable (modern Android restricts these; prefer Work Manager).

Bound: Use when components need to interact with a service, such as retrieving data or controlling service behavior



3.4.5 Implementing Background Tasks Using Work Manager or Job Scheduler

Modern Android restricts traditional background services due to battery optimizations. WorkManager and JobScheduler are recommended for scheduling and executing background tasks, as they handle system constraints (e.g., Doze mode, network availability).

WorkManager

WorkManager is a flexible library for scheduling deferrable, asynchronous tasks that are expected to run even if the app exits or the device restarts.

Steps to Implement WorkManager

1. **Add Dependency:** In your app's `build.gradle`:

```

```gradle
implementation "androidx.work:work-runtime-ktx:2.9.1"
```

```

2. **Create a Worker:** Create a class extending `Worker` to define the task.

```

```kotlin
import androidx.work.Worker

```



```
import android.content.Context
import android.util.Log
class UploadWorker(appContext: Context, params: WorkerParameters) : Worker(appContext,
params) {
 override fun doWork(): Result {
 // Simulate background task (e.g., upload data)
 Log.d("UploadWorker", "Uploading data...")
 Thread.sleep(2000) // Simulate work
 return Result.success()
 }
}
...

```

**3. Schedule the Work:** Use `WorkManager` to enqueue the task with constraints (e.g., network required).

```
```kotlin
import androidx.work.Constraints
import androidx.work.NetworkType
import androidx.work.OneTimeWorkRequestBuilder
import androidx.work.WorkManager
fun scheduleUpload(context: Context) {
    val constraints = Constraints.Builder()
.setRequiredNetworkType(NetworkType.CONNECTED)
    .build()

    val uploadRequest = OneTimeWorkRequestBuilder<UploadWorker>()
    .setConstraints(constraints)
    .build()

    WorkManager.getInstance(context).enqueue(uploadRequest)
}
...

```

4. Monitor Work: Observe the work's status.

```
```kotlin
WorkManager.getInstance(context).getWorkInfoByIdLiveData(uploadRequest.id)
 .observe(lifecycleOwner) { workInfo ->
 if (workInfo != null && workInfo.state.isFinished) {
 Log.d("UploadWorker", "Work finished: ${workInfo.state}")
 }
 }
}
...

```

### When to Use WorkManager

- Tasks that can be deferred (e.g., syncing data, uploading logs).
- Tasks requiring constraints like network or battery status.
- Guaranteed execution, even after app termination.

### JobScheduler

JobScheduler is an API for scheduling tasks that respect system constraints, available on API 21+.



### Steps to Implement JobScheduler

#### 1. Create a JobService:

```
```kotlin
import android.app.job.JobParameters
import android.app.job.JobService
import android.util.Log

class MyJobService : JobService() {
    override fun onStartJob(params: JobParameters?): Boolean {
        Log.d("MyJobService", "Job started")
        // Perform task (e.g., sync data)
        Thread {
            Thread.sleep(2000) // Simulate work
            Log.d("MyJobService", "Job completed")
            jobFinished(params, false)
        }.start()
        return true // Return true if work is ongoing
    }

    override fun onStopJob(params: JobParameters?): Boolean {
        Log.d("MyJobService", "Job stopped")
        return false // Return true to reschedule
    }
}
```
```

#### 2. Schedule the Job:

```
```kotlin
import android.app.job.JobInfo
import android.app.job.JobScheduler
import android.content.ComponentName
import android.content.Context

fun scheduleJob(context: Context) {
    val jobScheduler = context.getSystemService(Context.JOB_SCHEDULER_SERVICE) as JobScheduler
    val jobInfo = JobInfo.Builder(123, ComponentName(context, MyJobService::class.java))
        .setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)
        .setPersisted(true) // Survive reboots
        .build()
    jobScheduler.schedule(jobInfo)
}
```
```

#### 3. Add Permission: In `AndroidManifest.xml`:

```
```xml
<service
    android:name=".MyJobService"
    android:permission="android.permission.BIND_JOB_SERVICE" />
```
```

**When to Use Job Scheduler**

- Tasks requiring precise scheduling with system constraints.
- Older apps or when Work Manager is not suitable.

| FEATURE     | WORKMANAGER                                       | JOBSCHEDULER                  |
|-------------|---------------------------------------------------|-------------------------------|
| API LEVEL   | 14+                                               | 21+                           |
| CONSTRAINTS | COMPREHENSIVE                                     | LIMITED                       |
| PERSISTENCE | BUILT-IN                                          | REQUIRES<br>ADDITIONAL CODING |
| EASE OF USE | HIGH                                              | MODERATE                      |
|             | BACKGROUND SERVICE or<br>WORKMANAGER/JOBSCHEDULER |                               |

Fig 3.4. Work Manager vs. Job Scheduler

### 3.4.6 Procedures for Recurring Tasks and Performance Impact Monitoring

Recurring tasks are scheduled to run periodically (e.g., syncing data every hour). Use WorkManager for recurring tasks due to its simplicity and compatibility with modern Android restrictions.

Implementing Recurring Tasks

#### 1. Schedule Periodic Work:

```

kotlin
import androidx.work.PeriodicWorkRequestBuilder
import androidx.work.WorkManager
import java.util.concurrent.TimeUnit
fun scheduleRecurringUpload(context: Context) {
 val constraints = Constraints.Builder()
 .setRequiredNetworkType(NetworkType.CONNECTED)
 .build()

 val periodicWorkRequest = PeriodicWorkRequestBuilder<UploadWorker>(15, TimeUnit.MINUTES)
 .setConstraints(constraints)
 .build()
 WorkManager.getInstance(context)
 .enqueueUniquePeriodicWork("uploadWork", ExistingPeriodicWorkPolicy.KEEP,
periodicWorkRequest)
}

```

- Note: The minimum interval is 15 minutes for periodic work.  
data = newData

**2. Monitor Performance:**

- CPU and Battery Usage: Use Android Studio's Profiler to monitor CPU, memory, and battery usage.
- Logging: Add logs in the worker to track execution time and errors.
- WorkManager Status: Observe work status to detect failures or delays.

```

```kotlin
WorkManager.getInstance(context).getWorkInfosForUniqueWorkLiveData("uploadWork")
    .observe(lifecycleOwner) { workInfos ->
        workInfos.forEach { Log.d("RecurringWork", "State: ${it.state}") }
    }
...
```kotlin
WorkManager.getInstance(context).getWorkInfosForUniqueWorkLiveData("uploadWork")
 .observe(lifecycleOwner) { workInfos ->
 workInfos.forEach { Log.d("RecurringWork", "State: ${it.state}") }
 }
}

```

**3. Best Practices:**

- Avoid frequent executions to minimize battery drain.
- Use constraints to run tasks only when necessary (e.g., network available, device charging).
- Test on devices with Doze mode to ensure tasks execute as expected.

### 3.4.7 Simple Data Sharing Between Activities and Services

Data sharing between activities and services is common in Android apps. Bound Services or BroadcastReceivers are typical approaches for communication.

Using a Bound Service

**1. Create a Bound Service:**

```

```kotlin
import android.app.Service
import android.content.Intent
import android.os.Binder
import android.os.IBinder

class DataService : Service() {
    private val binder = LocalBinder()
    private var data: String = "Initial Data"
    inner class LocalBinder : Binder() {
        fun getService(): DataService = this@DataService
    }
    override fun onBind(intent: Intent?): IBinder = binder

    fun getData(): String = data
    fun updateData (newData: String) {
fun updateData(newData: String) {
    data = newData
}
}
}

```

2. Bind to the Service from an Activity:

```

```kotlin
import android.content.ComponentName
import android.content.Context
import android.content.Intent
import android.content.ServiceConnection
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import android.util.Log

class MainActivity : AppCompatActivity() {
 private var dataService: DataService? = null
 private var isBound = false

 private val connection = object : ServiceConnection {
 override fun onServiceConnected(name: ComponentName, service: IBinder) {
 val binder = service as DataService.LocalBinder
 dataService = binder.getService()
isBound = true
 Log.d("MainActivity", "Service Bound: ${dataService?.getData()}")
 }

 override fun onServiceDisconnected(name: ComponentName) {
 isBound = false
 }
 }

 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 setContentView(R.layout.activity_main)

 // Bind to service
 val intent = Intent(this, DataService::class.java)
 bindService(intent, connection, Context.BIND_AUTO_CREATE)
 }

 override fun onDestroy() {
 super.onDestroy()
 if (isBound) {
 unbindService(connection)
 isBound = false
 }
 }

 fun updateServiceData(newData: String) {
 if (isBound) {
 dataService?.updateData(newData)
 }
 }
}
```

```

3. Register the Service:

In `AndroidManifest.xml`:

```
```xml
<service android:name=".DataService" />
```
```

Using BroadcastReceiver

For one-way communication, use a `BroadcastReceiver`:

1. Send Broadcast from Service:

```
```kotlin
fun sendDataFromService(context: Context, data: String) {
 val intent = Intent("com.example.DATA_UPDATE")
 intent.putExtra("data", data)
 context.sendBroadcast(intent)
}
```
```

2. Receive Broadcast in Activity:

```
```kotlin
import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent
import android.content.IntentFilter
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.util.Log

class MainActivity : AppCompatActivity() {
 private val receiver = object : BroadcastReceiver() {
 override fun onReceive(context: Context?, intent: Intent?) {
 val data = intent?.getStringExtra("data")
 Log.d("MainActivity", "Received data: $data")
 }
 }

 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 setContentView(R.layout.activity_main)

 val filter = IntentFilter("com.example.DATA_UPDATE")
 registerReceiver(receiver, filter)
 }

 override fun onDestroy() {
 super.onDestroy()
 unregisterReceiver(receiver)
 }
}
```
```

3.4.8 Applying Memory Management Practices to Reduce Leaks or Slowdowns

Poor memory management in services can lead to memory leaks or app slowdowns. Key practices include minimizing resource usage, cleaning up resources, and monitoring memory.

Memory Management Practices

1. Stop Services When Done:

- For started services, call `stopSelf()` or `stopService()` when the task completes.

```
```kotlin
class MyService : Service() {
 override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
 // Perform task
 stopSelf() // Stop service when done
 return START_NOT_STICKY
 }
 override fun onBind(intent: Intent?): IBinder? = null
}
```
```

2. Unbind Bound Services:

- Activities must unbind from services in `onDestroy()` to prevent leaks.

```
```kotlin
override fun onDestroy() {
 super.onDestroy()
 if (isBound) {
 unbindService(connection)
 isBound = false
 }
}
```
```

3. Avoid Holding Static References:

- Static references to activities or contexts in services can cause leaks. Use `WeakReference` if needed.

```
```kotlin
import java.lang.ref.WeakReference

class MyService : Service() {
 private var activityRef: WeakReference<Context>? = null

 fun setActivity(context: Context) {
 activityRef = WeakReference(context)
 }
 override fun onBind(intent: Intent?): IBinder? = null
}
```
```

4. Use Lightweight Data Structures:

- Avoid large objects in memory-intensive tasks. Serialize data to disk if possible.

5. Monitor Memory Usage:

- Use Android Studio's Memory Profiler to detect leaks.
- Libraries like LeakCanary can help identify memory leaks:

```
```gradle
debugImplementation "com.squareup.leakcanary:leakcanary-android:2.12"
```

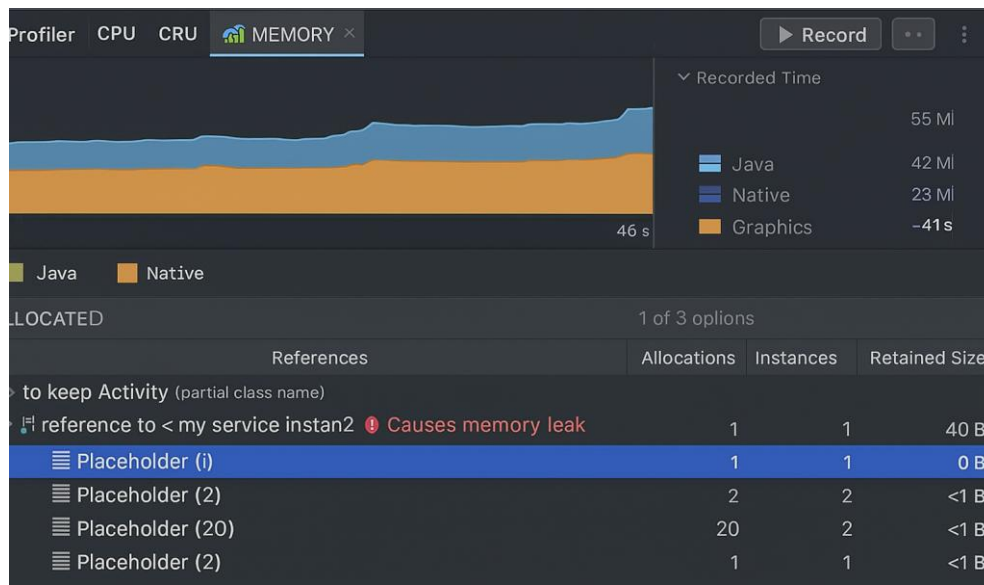


Fig 3.4. Memory Leak Analysis in Android Studio's Memory Profiler

### 3.4.8 Updating UI from Background Services Using View Model or Live Data

Updating the UI from a background service requires thread-safe communication. ViewModel and LiveData are standard tools for this purpose, ensuring data survives configuration changes and is observed by the UI.

#### Using ViewModel and LiveData

##### 1. Create a ViewModel:

```
```kotlin
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel

class DataViewModel : ViewModel() {
    private val _data = MutableLiveData<String>()
    val data: LiveData<String> get() = _data
}
```

```
fun updateData(newData: String) {
    _data.postValue(newData) // Thread-safe update
}
...

```

2. Update ViewModel from Service:

```
```kotlin
class DataService : Service() {
 private val viewModel by lazy { DataViewModel() }

 override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
 Thread {
 // Simulate background work
 Thread.sleep(2000)
 viewModel.updateData("Data from Service")
 }.start()
 return START_STICKY
 }

 override fun onBind(intent: Intent?): IBinder? = null
}
...

```

## 3. Observe LiveData in Activity:

```
```kotlin
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import androidx.activity.viewModels
import android.widget.TextView
import androidx.lifecycle.Observer
class MainActivity : AppCompatActivity() {
    private val viewModel: DataViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val textView = findViewById<TextView>(R.id.textView)
        viewModel.data.observe(this, Observer { data ->
            textView.text = data
        })

        // Start service
        startService(Intent(this, DataService::class.java))
    }
}
...

```


4. Layout File (`res/layout/activity_main.xml`):

```

<?xml
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Waiting for data..." />

```

Best Practices

Use `postValue()` for thread-safe updates to `LiveData`.

- Ensure `ViewModel` is scoped to the activity or fragment lifecycle.
- Avoid direct UI updates from services; always use intermediaries like `LiveData`.

Exercise

Short Questions:

1. Define the purpose of the onCreate() method in an Android Activity lifecycle.
2. Differentiate between Internal Storage and External Storage in Android.
3. What are the advantages of using Room Database over direct SQLite operations?
4. Describe how the Fused Location Provider improves location accuracy and battery efficiency.
5. What are the basic steps required to configure Google Maps API in an Android application?

Multiple Choice Questions:

1. Which Android component manages long-running background operations without a user interface?
 - a) Activity
 - b) Fragment
 - c) Service
 - d) BroadcastReceiver
2. Which of the following is best suited for storing small key-value pairs like user preferences?
 - a) SQLite Database
 - b) SharedPreferences
 - c) Internal Storage
 - d) Firebase
3. The Room database provides compile-time verification of:
 - a) App UI Layouts
 - b) SQL Queries
 - c) API Endpoints
 - d) Permissions

4. Firebase Cloud Messaging (FCM) is used for:

- a) Local database operations
- b) Sending push notifications
- c) Handling Bluetooth connections
- d) Encrypting stored data

5. The method used to obtain a precise device location through GPS in Android is:

- a) `getMapAsync()`
- b) `FusedLocationProviderClient`
- c) `startService()`
- d) `onBind()`

Fill in the Blanks

1. _____ is an abstraction layer over SQLite that simplifies database interactions using annotations.
2. In Android, _____ is used to store simple data such as Boolean, String, and Integer values.
3. Firebase _____ Messaging allows sending notifications to Android devices.
4. The _____ method in an Activity is called when it becomes visible but not yet interactive.
5. The process of converting coordinates into a readable address is called _____

Notes



A large rectangular area with horizontal lines for writing notes.



4. Testing and Publishing Android Applications for Telecom Devices



Unit 4.1 - Testing and Validating Android Applications

Unit 4.2 - Publishing Android Applications



Key Learning Outcomes

By the end of this module, the participants will be able to:

1. Explain the purpose and types of testing used in Android applications, including functional, UI, compatibility, and network-related testing.
2. Demonstrate the execution of predefined test cases and checklists to validate application functionality, usability, and performance.
3. Describe the process of identifying, documenting, and reporting software defects and security issues during testing.
4. Explain the procedures involved in preparing an Android application for release, including versioning, signing, and packaging.
5. Demonstrate the process of uploading application builds and completing publishing requirements for app stores or internal platforms.
6. Describe methods for monitoring app performance, crash reports, and user feedback to support continuous improvement.

UNIT 4.1: Testing and Validating Android Applications

Unit Objectives

By the end of this unit, the participants will be able to:

1. Describe various Android application testing types and their relevance to different features.
2. Demonstrate the execution of test cases and scripts to validate functional and usability aspects of applications.
3. Perform compatibility and network-related testing across devices, Android versions, and screen sizes using guided tools.
4. Apply standard UI/UX and accessibility guidelines while validating app interfaces.
5. Document and communicate software defects, performance issues, and test observations to the development team.
6. Demonstrate basic security and performance validation using standard testing tools and procedures.
7. Collect and organize test data and logs for analysis and review by the quality assurance team.

4.1.1 Different Types of Mobile Application Testing

Mobile application testing is a comprehensive process that ensures an app meets quality standards and provides a seamless user experience. Here are the key types:

1. Functional Testing: This type of testing verifies that each feature and function of the app works as expected according to the requirements. It includes testing all the functionalities, user interfaces, database connections, APIs, and overall system integration.

Example:

- Login/Logout: Can users successfully log in and out?
- Data Entry: Is data saved correctly after submission?
- Navigation: Do all buttons and links lead to the correct screens?

2. Compatibility Testing: Compatibility testing ensures that the application performs correctly across various devices, operating systems, network conditions, and screen sizes. This is crucial for mobile apps due to the vast fragmentation of the Android ecosystem.

Example:

- Testing on different Android versions (e.g., Android 11, 12, 13).
- Testing on various device manufacturers (Samsung, Google Pixel, OnePlus).
- Checking performance on different screen resolutions and aspect ratios.
- Verifying functionality with 3G, 4G, and Wi-Fi networks.

3. Usability Testing: Usability testing focuses on how easy and user-friendly the application is. It assesses the app's intuitiveness, efficiency, and satisfaction for the end-user. The goal is to identify any areas where users might struggle or find the app frustrating.

Example:

- Is the navigation straightforward?
- Are error messages clear and helpful?
- Can users complete tasks efficiently?
- Is the design aesthetically pleasing and easy to understand?

4. Performance Testing: Performance testing evaluates the app's speed, responsiveness, stability, and resource usage under various loads. It's vital to ensure the app remains fast and stable even with many users or complex operations.

Example:

- **Load Testing:** How does the app perform with many concurrent users?
- **Stress Testing:** How does the app behave under extreme load conditions beyond its capacity?
- **Stability Testing:** Does the app remain stable over a long period of continuous use?
- **Resource Usage:** How much battery, CPU, and memory does the app consume?

5. Security Testing: Security testing aims to identify vulnerabilities and weaknesses in the application that could be exploited by malicious users. It ensures the app and user data are protected from unauthorized access, data breaches, and other security threats.

Example:

- **Authentication & Authorization:** Are login mechanisms secure? Can unauthorized users access restricted features?
- **Data Encryption:** Is sensitive user data encrypted in transit and at rest?
- **Session Management:** Are user sessions managed securely?
- **Input Validation:** Is the app protected against common attacks like SQL injection or cross-site scripting?

4.1.2 Basic Use of Android Testing Tools

Android provides several powerful tools to aid developers and QAs in testing their applications.

1. Android Emulator: The Android Emulator allows you to run virtual Android devices on your computer. It simulates various hardware configurations, screen sizes, and Android versions, making it an indispensable tool for compatibility and functional testing without needing physical devices.

How to Use:

- Open Android Studio and go to Tools > Device Manager.
- Click Create device to set up a new virtual device with your desired specifications (e.g., Pixel 4, Android 13).
- Once created, you can launch the emulator from the Device Manager.
- You can then install and run your app on this virtual device just like a physical one.

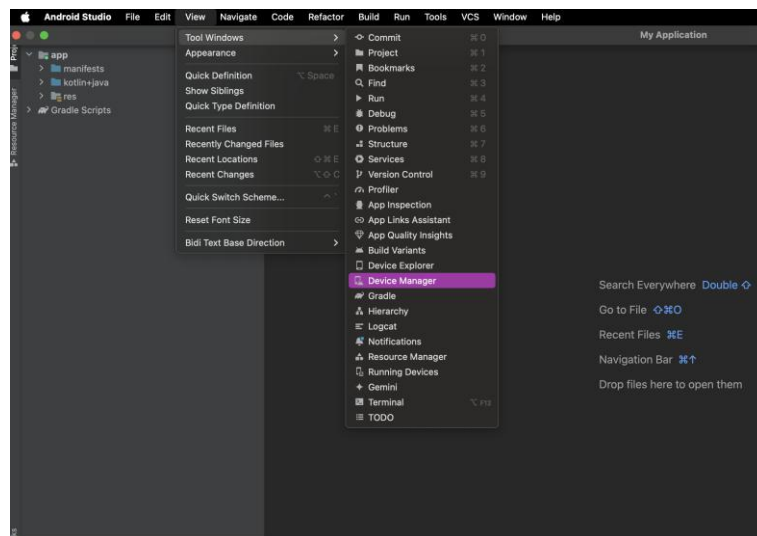


Fig 4.1.1 Android Emulator Running an App

2. Logcat: Logcat is a command-line tool and a window in Android Studio that displays system messages, app crashes, and messages explicitly logged by your application. It's essential for debugging and understanding what's happening internally in your app.

```

2021-12-13 21:28:16.343 17327-17327 Choreographer com.example.jetnews I Skipped 75 frames! The applicat
2021-12-13 21:28:16.454 17327-17369 OpenGLRenderer com.example.jetnews I Davey! duration=1351ms; Flags=0,
HandleInputStart=18005681432046, AnimationStart=18005681551046, PerformTraversalsStart=18005735226046, DrawStart=18005740566046, Fr
SyncStart=18005753115046, IssueDrawCommandsStart=18005753823046, SwapBuffers=18005763925046, FrameCompleted=18005770743046, Dequeue
DisplayPresentTime=0,
2021-12-13 21:28:16.679 17327-17327 Com...ityChangeReporter com.example.jetnews D Compat change id reported: 17122
----- PROCESS ENDED (17327) for package com.example.jetnews -----
----- PROCESS STARTED (17742) for package com.example.jetnews -----
2021-12-13 21:47:34.653 17742-17742 GraphicsEnvironment com.example.jetnews V ANGLE Developer option for 'com.
2021-12-13 21:47:34.654 17742-17742 GraphicsEnvironment com.example.jetnews V Neither updatable production dri
2021-12-13 21:47:34.656 17742-17742 NetworkSecurityConfig com.example.jetnews D No Network Security Config speci
2021-12-13 21:47:34.656 17742-17742 NetworkSecurityConfig com.example.jetnews D No Network Security Config speci
2021-12-13 21:47:34.766 17742-17795 LibEGL com.example.jetnews D loaded /vendor/lib64/egl/libEGL_
2021-12-13 21:47:34.768 17742-17795 LibEGL com.example.jetnews D loaded /vendor/lib64/egl/libGLES
2021-12-13 21:47:34.771 17742-17795 LibEGL com.example.jetnews D loaded /vendor/lib64/egl/libGLES
2021-12-13 21:47:35.017 17742-17742 example.jetnew com.example.jetnews W Accessing hidden method Landroid
2021-12-13 21:47:35.017 17742-17742 example.jetnew com.example.jetnews W Accessing hidden method Landroid
2021-12-13 21:47:36.105 17742-17742 example.jetnew com.example.jetnews W Class androidx.compose.runtime.s

```

Fig 4.1.2 Logcat - Android Studio

How to Use:

- In Android Studio, the Logcat window is usually at the bottom.
- You can filter logs by process ID (PID), application name, log level (Verbose, Debug, Info, Warn, Error, Assert), and specific keywords.
- To add your own logs in your app's code: `Log.d("MyTag", "This is a debug message");`

3. ADB (Android Debug Bridge): ADB is a versatile command-line tool that allows you to communicate with an Android-powered device or emulator. It provides commands for installing and uninstalling apps, copying files, accessing the device's shell, viewing logcat, and much more.

How to Use (Command Line):

- `adb devices:` Lists all connected devices/emulators.
- `adb install path/to/your/app.apk:` Installs an application package.
- `adb uninstall com.your.package:` Uninstalls an application.
- `adb pull /sdcard/file.txt .:` Copies a file from the device to your computer.
- `adb push localfile.txt /sdcard/:` Copies a file from your computer to the device.
- `adb shell:` Gives you a shell prompt on the device, allowing you to run Linux commands.

4.1.3 Importance of UI Consistency and Responsiveness Across Devices

UI consistency and responsiveness are paramount for a positive user experience on mobile.

UI Consistency: Consistency means that design elements, interactions, and behaviors within an app (and ideally across related apps) remain uniform. This includes:

- **Visual Elements:** Consistent use of colors, fonts, icons, spacing, and branding.
- **Navigation Patterns:** Similar menu structures, back button behavior, and tab bar layouts.
- **Interaction Design:** Consistent gestures, animations, and feedback mechanisms.
- **Terminology:** Using the same language and labels for actions and features.

Why it's important:

- **Reduces Cognitive Load:** Users don't have to learn new patterns for different parts of the app.
- **Improves Usability:** Makes the app intuitive and easy to navigate.
- **Builds Trust:** A consistent interface feels polished and professional.
- **Enhances Learnability:** Once a user learns one part of the app, they can easily understand others.

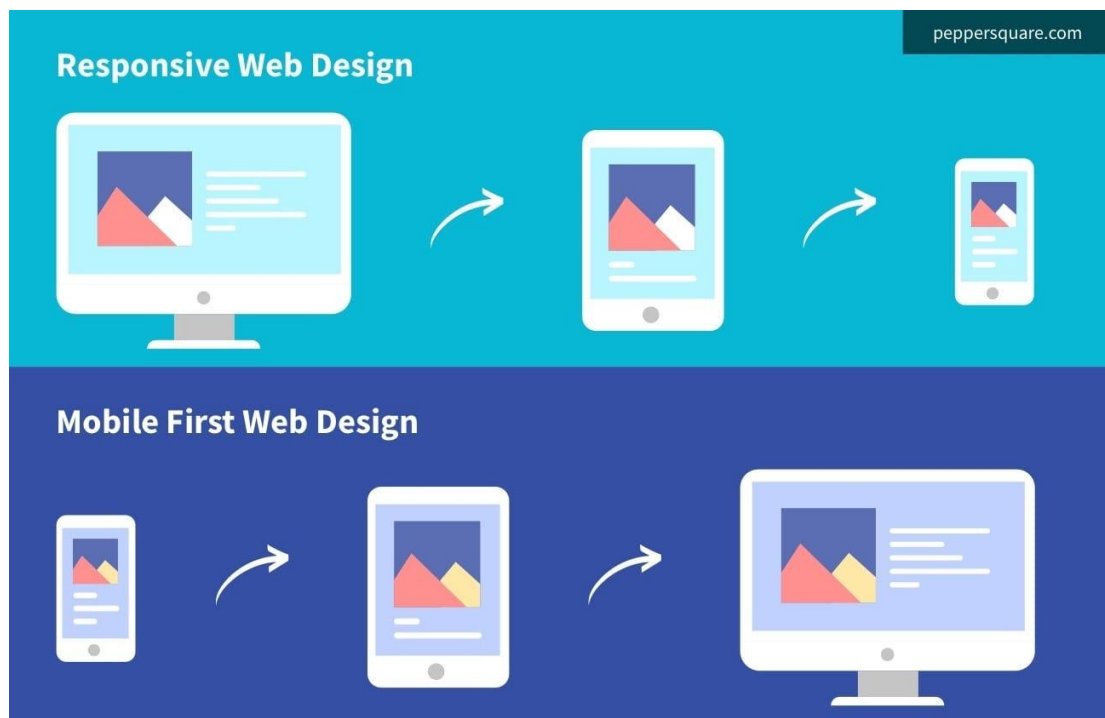


Fig 4.1.3 UI Consistency and Responsiveness

Responsiveness Across Devices: Responsiveness refers to an app's ability to adapt its layout and functionality seamlessly to different screen sizes, orientations (portrait/landscape), and device characteristics.

Why it's important:

- **Wider Audience Reach:** Ensures the app is usable and visually appealing on a vast array of Android devices, from small phones to large tablets.
- **Optimized User Experience:** Prevents cut-off content, awkward layouts, or tiny tap targets.
- **Accessibility:** Ensures users with different visual needs or input methods can effectively interact with the app.
- **Professional Appearance:** A responsive app looks well-designed and engineered, regardless of the device.

Techniques for achieving responsiveness:

- Using ConstraintLayout or Compose's flexible layouts to define relationships between UI elements rather than fixed positions.
- Providing alternative layouts and resources for different screen sizes and orientations (e.g., layout-land, values-sw600dp).

4.1.4 Performance Metrics and Benchmarks for Application Optimization

Optimizing app performance is crucial for user retention and satisfaction. Here are key metrics and benchmarks:

Key Performance Metrics:

- **App Launch Time:** How quickly does the app start and become interactive? (Benchmark: Under 2 seconds for a cold start).
- **Frame Rate (FPS - Frames Per Second):** How smoothly does the UI animate and scroll? Low FPS indicates jank or lag. (Benchmark: Consistent 60 FPS for smooth interactions).
- **Memory Usage:** How much RAM does the app consume? Excessive memory can lead to out-of-memory errors and slow performance. (Benchmark: Varies by app, but minimize unnecessary allocations).
- **CPU Usage:** How much processing power does the app use? High CPU usage can drain battery and slow down the device. (Benchmark: Keep background CPU usage low).
- **Battery Consumption:** How much battery does the app drain over time? (Benchmark: Minimize background activity, optimize network calls).
- **Network Usage:** How much data does the app transfer? Efficient networking is vital for user data plans. (Benchmark: Minimize data transfer, use compression, cache data).
- **App Size (APK/AAB size):** The size of the installed application. Smaller apps download faster and consume less storage. (Benchmark: Keep it as small as possible).
- **Jank/Stutter:** Visual hitches or pauses in the UI. Often measured by frames dropped.
- **ANRs (Application Not Responding):** Occur when the UI thread is blocked for too long, leading to a "App not responding" dialog. (Benchmark: Zero ANRs).

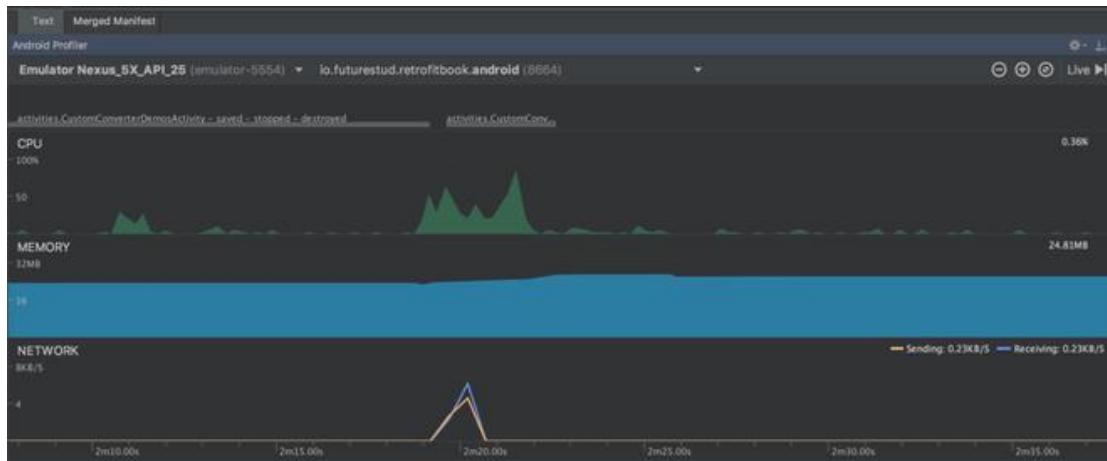


Fig 4.1.4 Performance Metrics in Android Studio

Benchmarking and Tools:

- **Android Profiler (in Android Studio):** A powerful tool to monitor CPU, memory, network, and energy usage in real-time. It helps pinpoint performance bottlenecks.
- **Firebase Performance Monitoring:** Allows you to gather performance data from real users in production, including app start-up times, network request latency, and custom traces.
- **Firebase Performance Monitoring:** Allows you to gather performance data from real users in production, including app start-up times, network request latency, and custom traces.
- **Systrace/Perfetto:** Advanced command-line tools for in-depth analysis of system performance, showing how processes interact with the CPU and other system resources.
- **Strict Mode:** A developer tool that detects accidental disk or network access on the application's main thread, and other violations.

Image Suggestion: A screenshot of the Android Profiler in Android Studio, showing graphs for CPU, Memory, and Network usage.

4.1.5 Android Security Policies, OWASP Mobile Security Best Practices, and GDPR Compliance

Security and privacy are critical in mobile app development.

1. Android Security Policies: Google enforces various security policies and features within the Android OS:

- **Permissions System:** Apps must explicitly request user permission to access sensitive data (e.g., location, contacts, camera) or system resources.
- **App Sandboxing:** Each app runs in its own isolated process, preventing unauthorized access to other apps' data or system resources.
- **SELinux (Security-Enhanced Linux):** Provides mandatory access control (MAC) for all processes, restricting what applications can do even if they gain root access.
- **Hardware-backed Keystore:** Allows storing cryptographic keys in hardware, making them more resistant to extraction.
- **Biometric Authentication:** Integrates with fingerprint and face unlock for secure user authentication.
- **Regular Security Updates:** Google regularly releases security patches for Android to address vulnerabilities.

2. OWASP Mobile Security Best Practices: The Open Web Application Security Project (OWASP) provides a comprehensive guide for secure mobile app development. The OWASP Mobile Top 10 lists the most critical mobile security risks:

- **M1: Improper Platform Usage:** Misusing platform security controls.
- **M2: Insecure Data Storage:** Storing sensitive data insecurely on the device.
- **M3: Insecure Communication:** Transmitting data over insecure network channels.
- **M4: Insecure Authentication/Authorization:** Weak or broken user authentication and authorization.
- **M5: Insufficient Cryptography:** Improper or weak use of encryption.
- **M6: Insecure Code Handling:** Vulnerabilities due to insecure coding practices.
- **M7: Poor Code Quality:** Bugs and errors that could lead to security flaws.
- **M8: Tampering:** Code or data tampering.
- **M9: Reverse Engineering:** Vulnerabilities to reverse engineering.
- **M10: Extraneous Functionality:** Leaving development/debugging functionality in production.

3. Best Practices include:

- **Secure Data Storage:** Encrypt sensitive data, avoid storing credentials, use Android Keystore.
- **Secure Communication:** Always use HTTPS/TLS for network communication, validate certificates.
- **Strong Authentication:** Implement multi-factor authentication, secure password policies.
- **Input Validation:** Sanitize all user inputs to prevent injection attacks.
- **Code Obfuscation:** Make it harder for attackers to reverse engineer your app.
- **Regular Security Audits:** Conduct penetration testing and code reviews.
- **Image Suggestion:** A stylized image representing mobile security, perhaps with a shield icon over a smartphone, or a padlock.



Fig 4.1.5 Mobile Security

4. GDPR Compliance (General Data Protection Regulation): GDPR is a European Union law that mandates data protection and privacy for all individuals within the EU. Even if your app isn't based in the EU, if it processes data of EU citizens, you must comply.

Key GDPR Principles for Mobile Apps:

- **Lawfulness, Fairness, and Transparency:** Clearly inform users about data collection and processing.
- **Purpose Limitation:** Collect data only for specified, explicit, and legitimate purposes.
- **Data Minimization:** Collect only the data absolutely necessary.
- **Accuracy:** Ensure personal data is accurate and up-to-date.
- **Storage Limitation:** Store data only for as long as necessary.
- **Integrity and Confidentiality:** Protect personal data from unauthorized processing, accidental loss, destruction, or damage.
- **Accountability:** Be able to demonstrate compliance with GDPR.

Practical Steps for Mobile Apps:

- **Explicit Consent:** Obtain clear and unambiguous consent from users before collecting personal data, especially for sensitive data.
- **Privacy Policy:** Provide a clear, accessible, and understandable privacy policy.
- **Data Encryption:** Encrypt personal data both in transit and at rest.
- **Data Access and Deletion:** Provide mechanisms for users to access, correct, or delete their personal data ("Right to be forgotten").
- **Data Breach Notification:** Have a plan to notify authorities and affected users in case of a data breach.



Fig 4.1.5 Mobile Privacy

4.1.6 Debugging Techniques, Crash Analytics, and Log Analysis for Troubleshooting

Troubleshooting is an essential skill for any developer or tester.

1. Debugging Techniques: Debugging is the process of finding and resolving defects or bugs in computer programs.

- **Breakpoints:** Set breakpoints in your code (e.g., in Android Studio by clicking in the left gutter) to pause execution at a specific line.
- **Stepping:** Once paused, you can step through the code line by line:
- **Step Over (F8):** Executes the current line and moves to the next line.
- **Step Into (F7):** Jumps into a method call on the current line.
- **Step Out (Shift+F8):** Completes the current method and returns to the caller.
- **Variable Inspection:** While paused, inspect the current values of variables, objects, and the call stack to understand the program's state.
- **Conditional Breakpoints:** Breakpoints that only pause execution when a certain condition is met (e.g., `i == 5`).
- **Watch Expressions:** Monitor specific expressions or variables as the program execute
- **Image Suggestion:** A screenshot of Android Studio's debugger in action, showing a breakpoint hit, variables window, and call stack.

2. Crash Analytics: Crash analytics tools automatically collect and report crashes that occur in your app on users' devices. This provides invaluable insight into what went wrong in production.

- **Why it's important:** Identifies common crash points, helps prioritize fixes, and provides context (device, OS, user actions) for reproducing bugs.
- **Popular Tools:**
- **Firebase Crashlytics:** A free, powerful crash reporting solution that provides real-time crash reports, detailed stack traces, and contextual information.
- **Sentry:** Another robust error tracking platform with rich features for debugging and performance monitoring.
- **AppCenter (Microsoft):** Offers crash reporting and analytics for various platforms, including Android.
- **Key Information Provided:**
- **Stack Trace:** The exact sequence of method calls that led to the crash.
- **Device Info:** Device model, OS version, orientation.
- **User Info (optional):** User ID, custom logs leading up to the crash.
- **Crash Frequency:** How many times the crash occurred and how many users were affected.

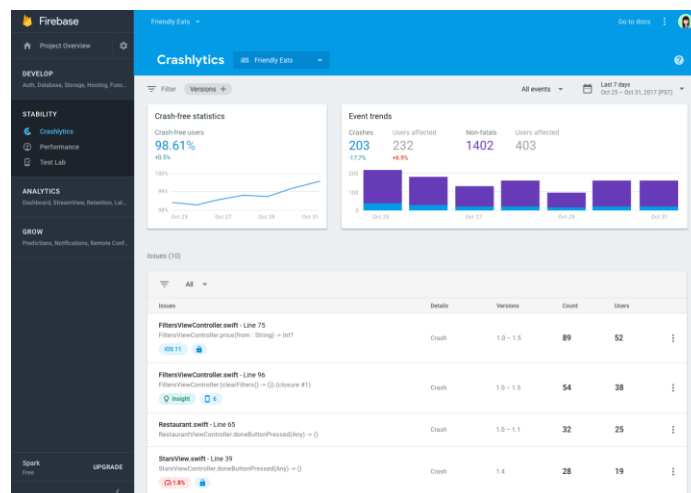


Fig 4.1.6 Crash Analytics

3. Log Analysis for Troubleshooting: As discussed, Logcat is your window into the app's runtime behavior. Analyzing logs is crucial for diagnosing issues, even those that don't result in a full crash.

How to Use Logs Effectively:

Add Strategic Logs: Place `Log.d()`, `Log.e()`, `Log.i()` statements at key points in your code to track program flow, variable values, and error conditions.

Filter Logs: Use Logcat's filtering capabilities to narrow down the noise and focus on relevant messages (by tag, PID, message content, or log level).

Understand Log Levels:

- **Verbose (V):** Everything, often too much noise.
- **Debug (D):** Debug-only messages, useful during development.
- **Info (I):** General information, milestones.
- **Warn (W):** Potential issues that are not errors.
- **Error (E):** Errors that have occurred.
- **Assert (A):** Assertions that should never happen.
- **Look for Keywords:** Search for "ERROR," "Exception," "Failed," or your app's package name to quickly find relevant issues.
- **Examine Surrounding Logs:** Often, the messages immediately before an error or crash provide crucial context.

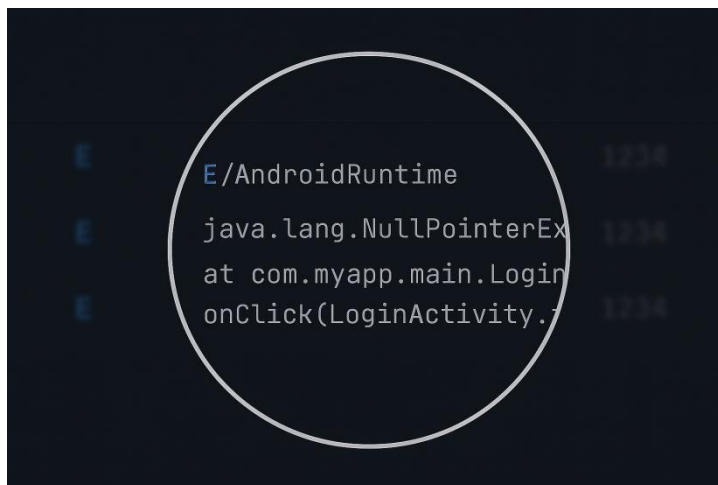


Fig 4.1.7 Logcat Error Message

By mastering testing, security, and troubleshooting techniques, you can significantly improve the quality, stability, and user experience of your Android applications.

4.1.7 Selecting Appropriate Mobile Application Testing Types

Selecting the right testing type is the first step in a successful validation process. The choice depends on the specific feature being tested and the stage of development.

Feature/Goal	Testing Type	Purpose	When to Use
New Feature Implementation	Functional Testing	Verifies that a specific feature (e.g., login, payment, data saving) works exactly as required.	Every time a new feature is added or modified.
Visual Design & Interaction	UI (User Interface) Testing	Ensures all visual elements (buttons, text, images) are correctly displayed, aligned, and styled according to design specifications.	During the design and front-end development phase.
New Android OS Version	Compatibility Testing	Checks if the application works correctly across various devices, OS versions, and screen resolutions.	Before major releases and after new Android OS/device launches.
App Usability & Flow	Usability Testing	Assesses how easy and intuitive the app is to use for the end-user.	When major UI/UX changes are made, or before a beta release.
Feature Maintenance/Fixes	Regression Testing	Ensures that recent code changes, fixes, or additions haven't broken existing, previously working features.	After every bug fix or significant code update.

Table 4.1.1 Selection Criteria for Mobile Application Testing

Executing Predefined Test Cases and Test Scripts

The core of testing involves executing detailed, predefined instructions called test cases to ensure expected outcomes.

1. Functional Testing Execution

Follow step-by-step instructions to verify a feature's intended behavior.

Example Test Case: User Login

Steps	Expected Result	Actual Result	Status (Pass/Fail)
1. Enter valid email and valid password.	User is redirected to the dashboard screen.	[Observation]	[Result]
2. Enter valid email and incorrect password.	An error message "Invalid credentials" is displayed.	[Observation]	[Result]
3. Click "Forgot Password."	User is taken to the password recovery screen.	[Observation]	[Result]

2. Usability Testing Execution

Focus on the user experience by following a user journey or scenario.

- **Scenario:** A first-time user attempts to add an item to their cart and complete checkout.
- **Checkpoints:** Observe if the user can easily find the product, if the cart icon is visible, and if the checkout process is clear. Note down any hesitation or confusion.

3. Regression Testing Execution

A regression test suite is a collection of critical, high-priority test cases that must be re-run after any change to confirm the app's core functions are intact.

- **Process:** Execute the predefined test cases from the regression suite (often automated, but done manually by testers for critical flows) and compare the results with the previous successful run.

4. Compatibility Testing

- **Compatibility testing** ensures your app provides a consistent experience across the diverse Android ecosystem. It is vital to avoid fragmenting your user base.

Key Compatibility Factors

- **Android Versions (OS):** Test on the latest version (e.g., Android 14), the most popular version (according to market share), and one or two older, still-supported versions. Use the Android Emulator to easily switch between OS versions.
- **Screen Sizes and Resolutions:** Test on a range of devices: small phones, large phones, and tablets. Use the Android Emulator or Device Manager to create Android Virtual Devices (AVDs) with different screen sizes (e.g., 4-inch phone, 10-inch tablet).
- **Device Manufacturers:** While time-consuming, it's best to test on physical devices from major manufacturers (e.g., Samsung, Xiaomi, Google Pixel) as they may have slight variations in their OS layers (skins).

5. Basic Network-Related Testing

Network testing verifies that the app handles connectivity issues gracefully and uses data efficiently.

Tools: You can simulate different network speeds and conditions using the Android Emulator settings (e.g., set to GSM, 3G, or 4G).

Checklist Items:

- **Offline Mode:** Does the app display cached data or a friendly "No Internet Connection" message when connectivity is lost?
- **Slow Speed:** Does the app time out gracefully and display an error, or does it hang indefinitely when the network is very slow (simulated as EDGE/3G)?
- **Network Switch:** Does the app resume correctly when switching from Wi-Fi to mobile data or vice-versa?
- **Data Consumption:** Observe data usage during a specific task (e.g., loading a feed) using the Android Profiler to ensure efficient data transfer.

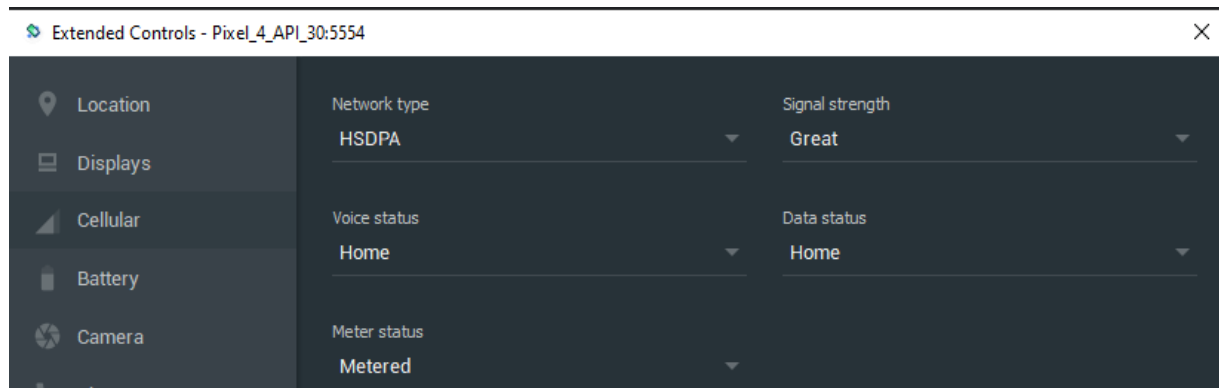


Fig 4.1.8 Android Emulator Controls

6. UI/UX and Security Verification

UI/UX Testing using Guidelines

This testing ensures the app adheres to established design standards (like Google's Material Design) and accessibility requirements.

Guideline Category	Verification Checklist
Layout & Spacing	Are all elements aligned correctly? Is there sufficient spacing between tappable elements? (Tappable targets should be at least 48dp).
Readability	Is the text size legible? Is there enough contrast between the text color and the background color?
Navigation	Is the back button behavior consistent? Does the app use standard navigation components (e.g., bottom navigation bar, hamburger menu)?
Accessibility	Does the app work well with TalkBack (Android's screen reader)? Are all icons/images provided with descriptive content descriptions?

Table 4.1.3 UI/UX Testing

Following Predefined Security Checklists

Basic security verification involves using checklists based on standards like OWASP Mobile Top 10 to catch obvious security flaws.

4.1.8 Defect Documentation and Coordination

Accurate and timely documentation of defects (bugs) is essential for the development team to fix them efficiently.

Key Elements of a Defect Report

A good defect report should allow the developer to reproduce the issue reliably.

- **Defect Title:** A clear, concise summary of the issue (e.g., "App crashes when submitting form with empty name field").
- **Environment:** The device, OS version, and network type where the bug was found (e.g., Pixel 7, Android 13, Wi-Fi).
- **Steps to Reproduce (STR):** A numbered, step-by-step list that consistently leads to the defect.
- 2. Dynamic Application Security Testing (DAST)

DAST tests an application while it is running to detect vulnerabilities in its runtime environment.

Basic Tool: ADB Commands

- Use adb shell to check app permissions and data access rights.
- Use the Android Debug Bridge (ADB) to inspect the app's local storage and logs for sensitive data exposure.
- Tool: Network Interception Proxies
- Tools like Burp Suite or OWASP ZAP can be set up to intercept and inspect network traffic between the app and the server.
- Action: Verify that no sensitive data (passwords, PII) is being sent in plain text, and check for proper SSL/TLS certificate pinning.
- Sharing Findings: Document the security findings with a detailed description of the vulnerability, the evidence (e.g., intercepted plain-text data), and the potential risk.

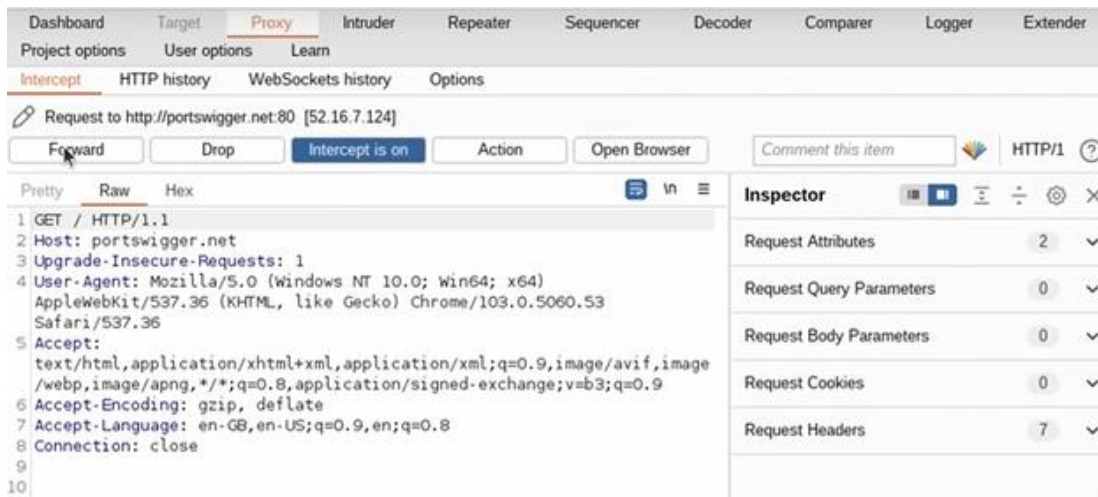


Fig 4.1.9 An Intercepted HTTPS Request

Notes



A large rectangular area with horizontal lines for writing notes.

UNIT 4.2: Publishing Android Applications

Unit Objectives

By the end of this unit, the participants will be able to:

1. Explain the essential publishing requirements and compliance criteria for releasing Android applications on app stores.
2. Demonstrate the process of versioning, signing, and packaging Android applications for deployment.
3. Complete metadata and release documentation accurately for Play Store or internal app store submission.
4. Demonstrate the uploading of application builds to designated distribution platforms under supervision.
5. Monitor and interpret crash reports, analytics, and user feedback to support post-release improvements.

4.2.1 Google Play Store Licensing Policies and Application Approval Guidelines

The Google Play Store has a comprehensive set of policies and guidelines that all apps must adhere to for listing and distribution.

Google Play Store Policies



Fig 4.2.1 Google Play Console

These policies are designed to protect users and ensure a safe, high-quality, and trustworthy ecosystem. Key policy categories include:

- **Restricted Content:** Prohibits apps containing sexual content, hate speech, violence, illegal activities, and content that promotes gambling (with restrictions).
- **Impersonation and Intellectual Property:** Apps cannot impersonate another brand, developer, or app. They must respect copyrights, trademarks, and patents.

Privacy and Security:

- **Privacy Policy:** Apps that handle personal or sensitive user data must post a comprehensive privacy policy in the Play Console and within the app itself.
- **Permissions:** Apps must request permissions that are reasonably required for their functionality and clearly inform users about the data collected.
- **Deceptive Behavior:** Apps must not engage in deceptive or malicious activities, such as phishing or injecting malware.

- **Monetization and Ads:** Policies govern in-app billing, subscriptions, and ad practices, ensuring they are clear, accurate, and fair.
- **Store Listing and Promotion:** App listings must be accurate, avoid spam, and not use misleading information or keywords.

4.2.2 Application Approval Guidelines (App Review Process)

When you submit an app to the Play Console, it undergoes a review process. Approval is granted if the app meets all policies and technical requirements.

- **Content Rating:** You must complete a content rating questionnaire to assign an appropriate age rating to your app.
- **Target Audience and Content:** You must clearly define the target age group of your app and confirm its content is suitable for that audience, particularly for children (COPPA/GDPR-K compliance).
- **Functionality:** The app must be fully functional, load correctly, and not crash or exhibit "Application Not Responding" (ANR) errors upon launch.
- **Metadata Compliance:** Your app title, description, and screenshots must accurately reflect the app's functionality and content.
- **Testing:** Google encourages thorough testing before submission. Apps with low stability or poor performance are often rejected.

4.2.3 Fundamentals of Android Services, Background Processing, and API Integrations

Effective mobile development requires handling tasks efficiently without freezing the UI.

Android Services: A Service is an application component that can perform long-running operations in the background, often without a user interface. Services are primarily used for:

- **Background Work:** Performing operations that need to continue even if the user switches to another app (e.g., playing music, downloading files).
- **Inter-Process Communication (IPC):** Allowing different applications to interact with each other.

Service Type	Description	When to Use
Foreground Service	Performs a noticeable operation and must display a persistent notification to the user.	Playing music, tracking location for a fitness app.
Background Service	Performs operations not directly noticeable to the user. Note: Heavily restricted in modern Android versions to save battery.	Deprecated for most uses; replaced by WorkManager.
Bound Service	Provides a client-server interface that allows components (like an Activity) to bind to the Service, send requests, and receive responses.	Allowing an Activity to control a background music player.

Background Processing

Modern Android APIs manage background tasks more intelligently to conserve battery and system resources.

- **WorkManager:** The recommended solution for deferrable, guaranteed background work. It handles compatibility across OS versions and uses system scheduling APIs (like JobScheduler) internally. Use it for tasks like syncing data, sending logs, or cleaning up databases.
- **Alarms (AlarmManager):** Used to schedule code to run at a specific time or recurring interval, even when the device is asleep.

API Integrations

Apps rarely exist in isolation; they integrate with external APIs to fetch or send data.

- **REST/JSON APIs:** The most common integration type. The app sends an HTTP request (GET, POST, PUT, DELETE) to a server endpoint and receives data back, usually in JSON format.
- **Libraries:** Libraries like Retrofit and Volley simplify API integration by handling network requests, threading, and JSON parsing.

4.2.4 Network Protocols, Data Handling, and Optimization for Mobile Applications

Efficient networking is key to performance and user satisfaction on mobile.

Network Protocols

- **HTTP/HTTPS:** The foundation of web communication. HTTPS (HTTP Secure) is mandatory for all sensitive data transfers as it encrypts communication using TLS/SSL, preventing eavesdropping.
- **TCP (Transmission Control Protocol):** Provides reliable, ordered, and error-checked delivery of a stream of bytes between applications. Used by HTTP/HTTPS.
- **UDP (User Datagram Protocol):** A simpler, connectionless protocol that is faster but unreliable. Used for tasks where lost data is acceptable (e.g., real-time voice/video streaming).

Data Handling and Optimization

Mobile networks are often slow, unreliable, and costly, making optimization crucial:

1. Data Minimization:

- **Compression:** Use efficient data formats like JSON or Protocol Buffers and ensure the server is compressing payloads (e.g., GZIP).
- **Partial Updates:** Use PATCH requests instead of PUT to send only the data that has changed.

2. Caching: Store frequently accessed data locally on the device.

- **HTTP Caching:** Use standard HTTP cache headers (e.g., Cache-Control, ETag) to avoid re-downloading unchanged content.
- **Database Caching:** Store data in a local database (like SQLite or Room) for offline access and instant loading.

3. Connection Management:

- **Batching:** Group multiple small network requests into a single, larger request to reduce connection overhead.
- **Pre-fetching:** Download data the user is likely to need next (e.g., the next page in a feed) while the current data is being viewed.

4. Error Handling: Implement robust code to handle network dropouts and server errors gracefully, providing clear feedback to the user.

4.2.5 Continuous Integration (CI) and Continuous Deployment (CD) Practices

CI/CD are practices that automate the process of building, testing, and deploying software, ensuring a fast and reliable release cycle.

Continuous Integration (CI)

CI is a development practice where developers frequently merge their code changes into a central repository (e.g., Git).

- **Automated Build:** Every time a change is pushed, the CI server automatically compiles the Android project (creating an APK/AAB file).
- **Automated Testing:** The server runs a suite of automated tests (unit tests, integration tests) to immediately catch integration errors.
- **Feedback:** Developers receive immediate feedback on whether their changes broke the build or introduced a new defect.

Continuous Deployment (CD)

- CD is the process of automating the deployment of the validated build to various environments.
- **Deployment Stages:** Automated deployment to internal testers (Alpha track), a small group of users (Beta track), and finally, to the Production track (Google Play Store).
- **Automation Tools:** Tools like Jenkins, GitLab CI, GitHub Actions, or Bitrise manage the entire workflow: fetching code, building, running tests, signing the app, and uploading the AAB/APK to the Play Console.

Benefits of CI/CD:

- Faster release cycles and time-to-market.
- Higher code quality due to frequent testing.
- Reduced risk by deploying smaller, incremental changes.

4.2.6 Best Practices for UX Design and Accessibility Compliance

A great app is not just functional; it's a pleasure to use and accessible to everyone.

User Experience (UX) Design Best Practices

UX focuses on optimizing the user's interaction with the app.

- **Clarity and Simplicity:** Design the interface with minimal clutter, using clear labels and intuitive icons. Follow the Principle of Least Astonishment, where the app behaves in a way users expect.
- **Consistency:** Adhere to established design guidelines (like Material Design) and maintain visual and interaction consistency throughout the app.
- **Feedback:** Provide immediate visual feedback for all user actions (e.g., button press effect, loading indicators, success messages).
- **Error Prevention and Recovery:** Design forms and processes to prevent errors, and if an error occurs, provide clear, helpful, and non-technical guidance for recovery.
- **Performance:** Ensure the UI remains responsive (60 FPS) to avoid jank and frustration.

Accessibility Compliance

Accessibility ensures that people with disabilities (visual, auditory, cognitive, or motor) can effectively use your app.

1. **Content Descriptions:** Provide `contentDescription` for all images and interactive UI elements so that TalkBack (Android's screen reader) can announce their purpose to visually impaired users.
2. **Sufficient Contrast:** Ensure a minimum color contrast ratio (e.g., 4.5:1 for standard text) for readability.
3. **Text Scaling:** Allow users to adjust text size without breaking the layout. Use Scale-independent Pixels (sp) for text dimensions.
4. **Non-Visual Cues:** Don't rely solely on color for meaning (e.g., use text labels in addition to color to indicate status).
5. **Target Size:** Ensure all interactive elements (buttons, links) are large enough to be easily tapped (minimum 48dp).

4.2.6 Use of Dashboards for Post-Release Monitoring and Reporting

Post-release monitoring is crucial for identifying performance, stability, and adoption issues in the wild. Key Monitoring Dashboards

1. Crash Analytics Dashboard (e.g., Firebase Crashlytics):

Focus: Stability and reliability.

- **Metrics:** Crash-Free Users/Sessions (the percentage of users/sessions without a crash), most frequent crashes, stack traces, and device breakdowns.
- **Reporting:** Prioritize fixing crashes that affect the largest number of users or are classified as Blockers/Critical.

2. Performance Monitoring Dashboard (e.g., Firebase Performance):

- **Focus:** Speed and resource usage.
- **Metrics:** App Start-up Time, network request latency, custom code execution traces, and device health reports (CPU/Memory).
- **Reporting:** Track performance regressions and identify slow API endpoints or inefficient code sections.

3. Google Play Console (Android Vitals):

- **Focus:** Core user-impacting issues as tracked by Google.
- **Metrics:** ANR (Application Not Responding) rate, excessive wakeups, battery drain, and excessive CPU usage.
- **Reporting:** Monitor ANR rates against Google's thresholds; failure to meet these thresholds can impact app visibility.

4. Analytics Dashboard (e.g., Google Analytics, Firebase Analytics):

- **Focus:** User behavior and adoption.
- **Metrics:** Daily/Monthly Active Users (DAU/MAU), session duration, key conversion funnels, and retention rates.
- **Reporting:** Inform product decisions by understanding which features are most used and where users are dropping off.

4.2.9 Version Control and CI/CD Tools in Team-Based Deployment Workflows

In a professional development environment, a structured workflow is essential for team collaboration and stable releases.

Role of Version Control (Git)

- Version Control Systems (VCS) like Git track and manage changes to code over time, allowing multiple developers to work concurrently.
- Collaboration: Provides a shared central repository (e.g., GitHub, GitLab, Bitbucket) where developers can merge their work without overwriting each other.
- Branching: The Git Flow strategy uses branches (e.g., main for production, develop for integration, and feature branches) to isolate work, ensuring the production code remains stable while new features are developed.
- Code Review: Pull Requests (PRs) require peers to review code before it's merged, ensuring quality and adherence to standards.
- Rollback: Allows the team to revert to any previous stable version of the code instantly in case of a major bug.

Role of CI/CD Tools in Deployment Workflows

CI/CD tools integrate with version control to automate and enforce the team's release policy.

- **Build Trigger:** CI/CD automatically triggers a build whenever code is merged into a specific branch (e.g., develop or main).
- **Automated Quality Gates:** The pipeline acts as a quality gate, automatically failing the build if unit tests fail, security scanning detects a critical vulnerability, or code coverage drops below a set threshold.
- **Environment Promotion:** Successful builds are automatically signed and deployed to the next testing environment (e.g., from the α track for internal testers to the β track for external users).
- **Release Orchestration:** The tools manage the entire release process, from creating release notes to notifying stakeholders, ensuring consistency and reducing human error.

4.2.10 Verifying Google Play Store Release Criteria (PC10)

The publishing phase is the final and critical step in the Android application lifecycle, ensuring the app is properly prepared, secured, and distributed to users. Before an app can be published on the Google Play Store, several basic criteria must be met to ensure compliance, stability, and a minimum quality standard.

Compliance Checklist

- **Policy Compliance:** Ensure the app adheres to all Google Play Developer Policies, particularly those concerning restricted content, intellectual property, and privacy.
- **Privacy Policy:** Verify a clear and accessible Privacy Policy URL is provided in the Play Console and, if applicable, within the app.
- **Content Rating:** Confirm the content rating questionnaire is completed to receive an appropriate age rating.

- **Target Audience:** Define the app's target audience and ensure its content is suitable for that age group.
- **Advertising ID Usage:** If the app uses an advertising ID, ensure its usage complies with policy and is disclosed correctly.

Technical and Quality Checklist

- **Stability:** Confirm the app is crash-free and doesn't exhibit frequent ANRs (Application Not Responding) based on internal testing or beta feedback.
- **Functionality:** Verify all core features work as intended on target devices and OS versions.
- **App Bundles (AAB):** Ensure the app is packaged as an Android App Bundle (AAB), which is the required publishing format for Google Play.
- **Testing Tracks:** Confirm the app has been thoroughly tested on the Alpha/Beta testing tracks before being promoted to production.

4.2.11 Versioning, Signing, and Packaging Applications (PC11)

These are technical steps essential for identifying and securing the application. Versioning

Android apps are versioned using two attributes in the build.gradle file:

- **Version Code:** A unique integer used internally by the Play Store and Android OS to determine if one version is more recent than another. It must always increase with every release.
- **Version Name:** A string visible to users (e.g., "2.1.0" or "Beta 3"). It's used for display purposes.
Assistance: Help verify that the versionCode has been correctly incremented and the versionName reflects the planned release number before the final build.

Signing

Every Android application must be digitally signed with a release key (keystore) before it can be installed or published. This is crucial for security and establishing trust.

- **Purpose:** The signature verifies the app's author and ensures the app hasn't been tampered with since it was signed.
- **Google Play App Signing:** Modern practice involves the developer signing the app with an upload key, and Google managing the secure App Signing Key for distribution.
- **Assistance:** Verify that the correct release build type is selected in the build process, which automatically applies the digital signature using the specified keystore configuration.

Packaging

The final output is the Android App Bundle (.aab) file.

- **Action:** In Android Studio, this involves navigating to Build > Generate Signed Bundle / APK... and selecting Android App Bundle.
- **Assistance:** Locate and verify the final, signed .aab file in the project's output directory.

4.2.12 Filling in Play Store Metadata Forms (PC12)

Metadata provides the app's details to the user and is crucial for discoverability and first impressions.

Key Metadata Fields

1. **App Name:** The name visible in the store and under the app icon. (Max 30 characters).
2. **Short Description:** A concise summary displayed prominently in search results. (Max 80 characters).
3. **Full Description:** Detailed information about features, benefits, and use cases. (Max 4,000 characters).
4. **Graphics:**
 - **App Icon:** High-resolution icon (512x512 pixels).
 - **Feature Graphic:** A prominent banner image (1024x500 pixels) used on the app page.
 - **Screenshots:** Images showing the app's key screens and functionality (must be current).
5. **Category:** The appropriate app or game category (e.g., "Finance," "Tools").

Assistance: Use the provided marketing copy and design assets to fill in these fields within the Google Play Console. Ensure that all text adheres to the character limits and that graphics meet the size and format requirements.

4.2.13 Uploading Application Builds to Distribution Platforms

Once the app is signed and the metadata is ready, the build is uploaded to the distribution platform.

1. **Platform Access:** Gain access to the Google Play Console or the internal enterprise app store (e.g., using a service account or supervisor credentials).
2. **Select a Track:** Choose the appropriate track for the upload:
 - **Internal Testing:** For small, immediate internal tests.
 - **Closed Testing (Beta):** For a defined list of testers.
 - **Open Testing (Beta):** For a wider audience.
 - **Production:** For the final public release.
3. **Upload:** Navigate to the release page for the chosen track and upload the prepared .aab file.
4. **Review Release:** After uploading, verify the Play Console correctly extracts the version code and displays the size of the release.
5. **Rollout:** Follow the supervisor's instructions on the type of rollout:
 - **Full Rollout:** Available to 100% of the audience immediately.
 - **Phased Rollout:** Gradually release to a small percentage (e.g., 5%, 10%) to monitor stability before expanding.

4.2.14 Monitoring Crash Reports and User Feedback

Post-release monitoring is a continuous process that drives iterative improvement.

Monitoring Tools

Use monitoring dashboards like Firebase Crashlytics and the Google Play Console (specifically the Android Vitals and Ratings & Reviews sections).

Exercise

Multiple Choice Questions:

1. Which testing type ensures the app performs correctly across different Android versions and devices?
 - a) Usability testing
 - b) Compatibility testing
 - c) Functional testing
 - d) Regression testing
2. The Android tool used to view system messages, crashes, and logs is:
 - a) ADB
 - b) Logcat
 - c) Emulator
 - d) Profiler
3. Which of the following ensures secure network communication between the app and the server?
 - a) HTTP
 - b) UDP
 - c) HTTPS
 - d) FTP
4. In Android Studio, WorkManager is primarily used for:
 - a) Scheduling notifications
 - b) Running long-running background tasks
 - c) Managing user permissions
 - d) Building APKs
5. Which dashboard tool helps monitor app stability and crashes in production?
 - a) Firebase Crashlytics
 - b) Google Analytics
 - c) Jenkins
 - d) GitLab CI
6. The versionCode in Android's build.gradle file is used to:
 - a) Display the app's version name to users
 - b) Indicate the app's release date
 - c) Identify the internal version for updates
 - d) Describe release notes
7. Which Play Store policy focuses on protecting user data and ensuring transparency?
 - a) Monetization policy
 - b) Privacy policy
 - c) Target audience policy
 - d) Intellectual property policy

Fill in the Blanks:

1. _____ testing verifies that each feature of the app works as intended according to requirements.
2. The Android _____ allows developers to simulate devices for testing without using real hardware.
3. _____ is used to communicate with Android devices from the command line for installing or debugging apps.
4. Apps must be digitally _____ before they can be uploaded to the Google Play Store.
5. The process of preparing and uploading builds to testing or production environments is known as _____.
6. A good defect report includes a clear title, environment details, and _____ to reproduce the issue.
7. The Android Profiler is used to monitor _____, memory, and network usage in real time.
8. Google Play requires apps handling user data to include a valid _____ policy.
9. The abbreviation GDPR stands for _____.
10. The process of gathering crash and performance data after release is known as _____ monitoring.

Short Answer Questions

1. Define functional testing and give one example of when it is used in Android development.
2. How does compatibility testing contribute to better user experience in Android applications?
3. What is the purpose of the OWASP Mobile Top 10, and how does it relate to mobile app security?
4. Describe two key functions of ADB (Android Debug Bridge) for testers.
5. Explain why UI consistency and responsiveness are important across Android devices.
6. What are the key differences between Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST)?
7. What is the role of versioning and signing before publishing an app on Google Play?
8. List three essential fields that must be completed in Play Store metadata forms.
9. How does Continuous Integration (CI) help maintain code quality in Android projects?
10. Describe how Firebase Crashlytics supports post-release quality improvement.

Notes



Lined area for taking notes, consisting of 30 horizontal lines.



5. Sustainably Practices in the Development of Mobile Applications



- Unit 5.1 - Sustainable Coding Practices
- Unit 5.2 - Application Performance Optimization
- Unit 5.3 - Network and Data Usage Reduction
- Unit 5.4 - Environmental Standards Compliance
- Unit 5.5 - Sustainable UI/UX Design
- Unit 5.6 - Sustainable Development Practices



Key Learning Outcomes



By the end of this module, the participants will be able to:

1. Explain the concept and importance of sustainability in mobile application development and its role in reducing environmental impact.
2. Demonstrate sustainable coding practices by writing optimized and efficient code that minimizes processing power, memory usage, and redundant computations.
3. Apply performance optimization techniques to manage system resources efficiently and reduce battery and storage consumption in mobile applications.
4. Implement data and network optimization methods, including caching, compression, and efficient data formats, to reduce bandwidth and energy usage.
5. Describe key environmental regulations, frameworks, and green software standards relevant to sustainable mobile application development.
6. Design user interfaces and experiences that support sustainability, including dark mode, adaptive themes, and adjustable performance settings.
7. Promote sustainable development practices by adopting cloud services powered by renewable energy, digital collaboration tools, and reusable code components.

UNIT 5.1: Sustainable Coding Practices

Unit Objectives



By the end of this unit, the participants will be able to:

1. Explain the importance of sustainable and energy-efficient coding in mobile application development.
2. Demonstrate the writing of optimized and efficient code to reduce CPU load and memory consumption.
3. Apply energy-efficient algorithms and suitable data structures to enhance processing performance.
4. Identify and eliminate redundant code segments and unnecessary computations to improve overall efficiency.
5. Describe how optimized coding contributes to lower energy usage and improved device sustainability.

5.1.1 Adopting Sustainable Coding Practices

In today's world, sustainability isn't just about saving the environment — it's also about using technology wisely and efficiently.

In software development, sustainable coding means writing programs that use less energy, run faster, and make better use of device resources like memory and battery.

This is especially important in mobile applications and telecom devices, where apps run on limited hardware and battery power.

When we adopt sustainable coding practices, we help:

- Improve app performance
- Extend device battery life
- Reduce carbon emissions caused by excess energy use
- Create software that runs smoothly on low-end devices

5.1.2 Writing Optimized and Efficient Code

Optimized code is like a well-planned journey — it gets to the destination quickly, using the least amount of fuel.

In coding, “fuel” means CPU power and memory. The more optimized your code, the less the device has to work — saving battery and reducing heat.

Key Techniques:

1. Avoid unnecessary loops or repeated operations.

Example: Instead of calculating the same value many times inside a loop, calculate it once and store it in a variable.

```
// Inefficientfor (int i = 0; i < 100; i++) {  
    System.out.println(Math.pow(2, 3)); // calculated every time
```

```

}
// Efficient double value = Math.pow(2, 3); for (int i = 0; i < 100; i++) {
    System.out.println(value);
}

```

2. Release unused memory.

In Android or Java, remove references to unused objects and let the system's garbage collector free memory.

3. Use lightweight data types.

For example, use `int` instead of `long` when you don't need very large numbers.

4. Load only what is needed.

Don't keep large images or files in memory if you can load them when required.

Example:

When a mobile app uses 20 MB of RAM instead of 200 MB because of efficient coding, it saves battery and allows other apps to run smoothly.

5.1.3 Using Energy-Efficient Algorithms and Data Structures

Algorithms are like recipes for solving problems — some are quicker and smarter than others.

Choosing the right algorithm helps reduce how much time and energy a device spends performing tasks.

Energy-Efficient Practices:

1. Use faster search or sort algorithms

For example, use binary search (which divides the data each time) instead of scanning all items one by one (linear search).

```
// Binary search is faster and uses less CPU cycles
```

```
Collections.binarySearch(myList, value);
```

2. Use proper data structures

- Use an `ArrayList` instead of `LinkedList` when you need fast access to data.
- Use a `HashMap` when you need to look up values quickly using keys.

Choosing the right structure avoids unnecessary data movement or copying.

3. Avoid busy-waiting

Don't use loops that continuously check for a condition; instead, use event-based triggers or callbacks.

5.1.4 Minimizing Redundant Code and Removing Unnecessary Computations

Redundant code is like repeating the same instructions in a recipe — it wastes time and makes the program harder to maintain.

By removing redundancy, we make the program smaller, faster, and easier to understand.

Best Practices:

- Use functions or methods to avoid repeating the same code.

// Instead of repeating this

```
System.out.println("Welcome, user!");
```

```
System.out.println("Welcome, user!");
```

```
System.out.println("Welcome, user!");
```

```
// Create a method void greetUser() {
```

```
    System.out.println("Welcome, user!");
```

```
}
```

```
greetUser();
```

- Reuse existing libraries or built-in methods instead of writing everything from scratch.

For example, use Android's built-in `Log.d()` for debugging instead of writing your own logging system.

- Avoid unnecessary calculations.

If a value doesn't change, calculate it once and reuse it.

Use constants instead of recalculating the same formula repeatedly.

- Regularly review and refactor code.

Cleaning up old or unused parts of code keeps the app efficient.

5.1.5 Benefits of Sustainable Coding

Benefit	Explanation
Faster performance	The app runs smoother and loads faster
Lower energy use	Reduces battery drain and CPU usage
Smaller app size	Saves storage and memory space
Environment-friendly	Uses less energy overall, reducing carbon footprint
Easier maintenance	Clean, well-structured code is easier to update and debug

Real-Life Example:

Imagine two versions of a mobile app that displays nearby telecom towers:

- Version A refreshes location data every second, even when the user is not moving.
- Version B updates only when the user moves a certain distance.

Version B is using sustainable coding practices — it performs fewer computations, saves power, and still provides the same functionality.

Sustainable coding means thinking smart before writing every line of code.

It focuses on performance, efficiency, and environmental responsibility.

By following simple habits — optimizing code, choosing efficient algorithms, and removing redundancy — we not only make our apps faster but also contribute to a greener, more sustainable digital world.

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 5.2: Application Performance Optimization

Unit Objectives

By the end of this unit, the participants will be able to:

1. Explain the significance of performance optimization in reducing device energy and resource usage.
2. Demonstrate efficient resource management techniques to minimize battery consumption during app operation.
3. Apply methods to optimize background processes and data retrieval for improved responsiveness and efficiency.
4. Implement strategies to reduce application size by removing redundant assets and compressing files.
5. Describe how performance optimization enhances user experience and supports sustainable mobile application design.

5.2.1 Efficient Resource Management to Reduce Battery Consumption

Optimizing application performance is crucial for user satisfaction, battery life, and even environmental sustainability. Let's delve into some key strategies, complete with illustrative images.

Efficient resource management is paramount for mobile applications, as it directly impacts battery life. This involves being mindful of how your app uses CPU, memory, network, and device sensors.

- **CPU Usage:** Minimize computationally intensive operations. If complex calculations are necessary, consider offloading them to a server or performing them in the background when the device is charging. Avoid continuous polling or busy loops.
- **Memory Management:** Optimize data structures and algorithms to reduce memory footprint. Release memory when it's no longer needed to prevent memory leaks and unnecessary garbage collection, which consumes CPU cycles.
- **Network Usage:** Reduce the frequency and size of network requests. Use techniques like data compression, caching, and batching requests. Opt for push notifications over continuous polling.
- **Sensor Usage:** Sensors like GPS, accelerometer, and camera can be significant power drains. Only activate them when absolutely necessary and deactivate them promptly when not in use. Choose less precise sensors if high accuracy isn't critical.
- **Display Usage:** The screen is a major power consumer. Avoid unnecessary screen refreshes and use darker themes or lower brightness levels where appropriate.

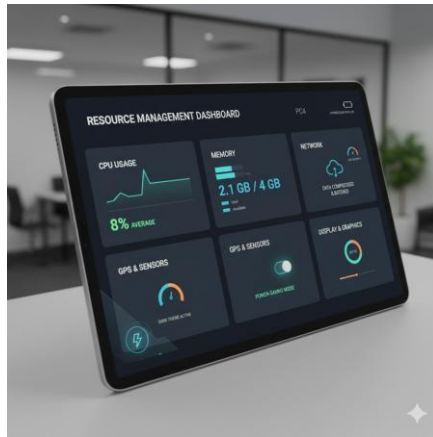


Fig 5.2.1 Resource Management Dashboard

5.2.2 Optimizing Background Processes and Data Retrieval Mechanisms

Background processes and data retrieval can silently drain battery and consume data if not managed efficiently.

- **Background Tasks:** Defer non-critical tasks to optimal times, such as when the device is charging or connected to Wi-Fi. Use platform-specific APIs (e.g., WorkManager on Android, BackgroundTasks on iOS) to schedule tasks intelligently, considering network availability and battery levels.
- **Data Synchronization:** Implement smart synchronization strategies. Instead of frequent full synchronizations, use incremental updates. Only sync data that has changed.
- **Push Notifications vs. Polling:** Prioritize push notifications for real-time updates over continuous polling, which constantly wakes up the device and consumes resources.
- **Conditional Data Fetching:** Only fetch data when it's needed by the user. For instance, lazy-load images and data as the user scrolls.

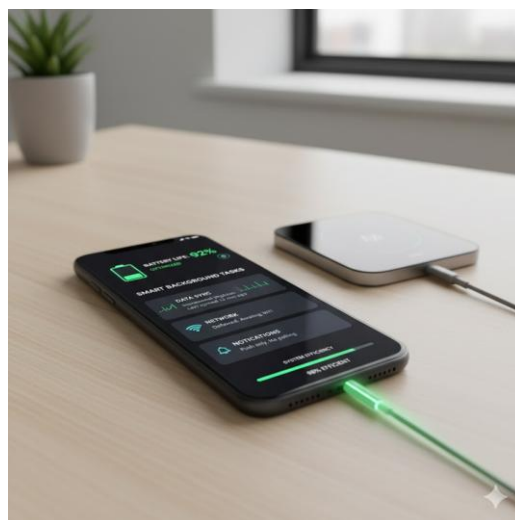


Fig 5.2.2 Optimizing Background Processes

5.2.3 Reducing Application Size by Eliminating Redundant Assets and Compressing Files

A smaller application size not only saves storage space on the user's device but also reduces download times and data consumption during updates, leading to a better user experience.

Asset Optimization:

- Images: Compress images without significant loss of quality. Use appropriate formats (e.g., WebP for Android, HEIC for iOS) and scale images to the required display density. Eliminate unused images.
- Videos and Audio: Compress media files and stream them efficiently.
- Fonts: Only include the necessary font variations and subsets.

Code Optimization:

- Remove Unused Code: Use code shrinking tools (e.g., ProGuard/R8 for Android, app thinning for iOS) to remove unused classes, methods, and fields.
- Modularization: Break down your app into smaller modules. This allows users to download only the features they need.
- Dependency Management: Be mindful of the libraries and frameworks you include. Each dependency adds to the overall app size. Only include what's truly necessary.
- Resource Exclusion: For multi-language apps, allow users to download only the language resources they need.



Fig 5.2.3 App Size Optimization

5.2.4 Environmental Benefits of Optimized and Efficient Mobile Apps

The environmental impact of digital technology is often underestimated. Optimized and efficient mobile apps contribute significantly to reducing this impact.

- **Reduced Energy Consumption:** Apps that use less battery power mean users charge their devices less frequently. This reduces the demand for electricity, leading to lower greenhouse gas emissions from power generation.
- **Extended Device Lifespan:** An app that runs smoothly and doesn't stress device resources can extend the useful life of a smartphone. This reduces electronic waste (e-waste) and the demand for new device manufacturing, which is resource-intensive.
- **Less Data Center Load:** Efficient apps require less processing power and storage on cloud servers. This translates to lower energy consumption by data centers, which are massive energy users.
- **Reduced Network Traffic:** Optimized data retrieval and compression minimize the data transferred over networks. This lessens the energy consumed by network infrastructure.



Fig 5.2.4 Global Environmental Benefits

5.2.5 Principles of Green Coding and Sustainable Software Engineering

Green coding and sustainable software engineering are about designing, developing, and deploying software in a way that minimizes environmental harm and promotes resource efficiency throughout the software's lifecycle.

- **Energy Efficiency:** Prioritize algorithms and data structures that require less computational power. Minimize I/O operations and network traffic.
- **Resource Conservation:** Optimize memory usage, reduce storage footprint, and leverage existing hardware capabilities instead of constantly demanding upgrades.
- **Longevity and Maintainability:** Write clean, modular, and well-documented code that is easy to maintain and update. This extends the software's lifespan and reduces the need for complete rewrites.

- **Scalability and Elasticity:** Design systems that can efficiently scale up and down based on demand, avoiding over-provisioning of resources.
- **Awareness and Education:** Foster a culture of sustainability among developers and stakeholders, encouraging them to consider the environmental impact of their choices.

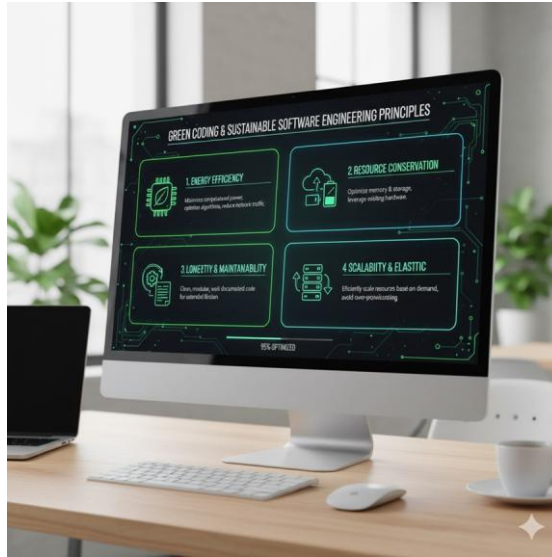


Fig 5.2.5 Core Principles of Green Coding

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 5.3: Network and Data Usage Reduction

Unit Objectives



By the end of this unit, the participants will be able to:

1. Understand strategies to optimize network performance and reduce data transfer.
2. Learn to minimize redundant network requests through the use of caching mechanisms.
3. Apply knowledge of efficient data formats to reduce processing load and improve application responsiveness.

5.3.1 Optimize API Calls and Minimize Data Transfers

Reducing network and data usage is a critical aspect of application optimization, especially for mobile users who might have limited data plans or unstable network connections. It also contributes to faster load times and reduced server costs.

The way your application interacts with APIs has a significant impact on data usage and performance.

- **Batching Requests:** Instead of making multiple small API calls, combine them into a single, larger request where logically possible. This reduces the overhead of establishing and closing network connections.
- **Selective Data Fetching:** Only request the specific data fields you need from an API. Many APIs allow you to specify which fields to include in the response, preventing the transfer of unnecessary information.
- **Pagination:** For large datasets, implement pagination to fetch data in smaller, manageable chunks rather than retrieving everything at once. This is crucial for lists, feeds, and search results.
- **Conditional Requests (ETags/Last-Modified):** Use HTTP headers like ETag and Last-Modified to allow the server to tell the client if its cached version of a resource is still valid. If it is, the server can respond with a 304 Not Modified status, avoiding re-sending the entire resource.
- **Compression:** Ensure your API responses are gzipped or compressed before being sent over the network. Most web servers and API gateways support this automatically.

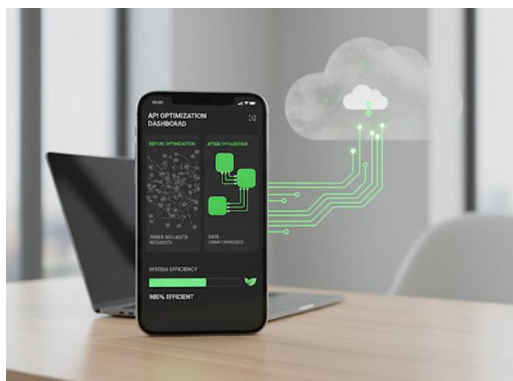


Fig 5.3.1 Optimized API calls

5.3.2 Caching Mechanisms to Reduce Redundant Network Requests

Caching is one of the most effective strategies to reduce network usage and improve application responsiveness. By storing frequently accessed data locally, your app can retrieve it much faster without needing to hit the network every time.

- **HTTP Caching:** Leverage standard HTTP caching headers (Cache-Control, Expires, ETag, Last-Modified) to instruct browsers or network layers on how to cache responses.
- **Application-Level Caching:** Implement in-memory caches or disk caches for data that is critical to your app's functionality but doesn't change frequently. This can include user profiles, configuration settings, or static content.
- **Offline Support:** For certain types of data, consider making it available offline. This allows users to access content even without an internet connection and further reduces network requests when online.
- **Stale-While-Revalidate:** A common caching strategy where the app serves stale (cached) content immediately while asynchronously revalidating it with the server. This provides a fast user experience while ensuring data freshness.
- **Content Delivery Networks (CDNs):** For static assets like images, videos, and large files, using a CDN can drastically reduce load times and bandwidth consumption by serving content from a server geographically closer to the user.

Below is an image depicting how caching works, showing data being served from a local cache instead of repeatedly fetching it from a remote server:



Fig 5.3.2 Caching

5.3.3 Use of Efficient Data Formats (e.g., JSON over XML)

The choice of data format for transmitting information between your app and servers has implications for both network bandwidth and the processing load on the device.

- **JSON (JavaScript Object Notation):** JSON is widely preferred over XML for many web and mobile applications due to its lightweight nature.
- **Less verbose:** JSON typically uses less markup than XML, resulting in smaller file sizes for the same data.
- **Easier to parse:** For JavaScript-based clients, JSON is natively parsable, requiring less code and computational effort to process. Other languages also have highly optimized JSON parsers.
- **Protocol Buffers, Thrift, Avro:** For highly performance-critical scenarios, especially in internal microservices communication, binary serialization formats like Protocol Buffers, Apache Thrift, or Apache Avro can offer even greater efficiency. They are typically much smaller than JSON/XML and faster to serialize/deserialize, but require schema definition and code generation.
- **MessagePack:** Another binary serialization format that is more compact than JSON but offers similar flexibility. It's often used in scenarios where JSON's human readability isn't a top priority and size/speed are critical.
- **Compression:** Regardless of the format chosen, always combine it with compression (like Gzip) at the HTTP level to further minimize the actual data sent over the wire.

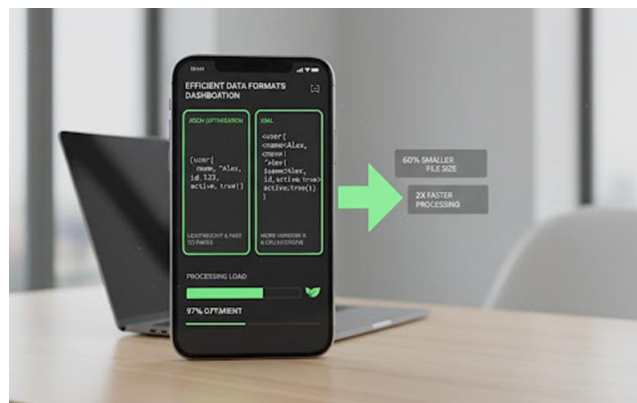


Fig 5.3.3 JSON's efficiency over XML

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 5.4: Environmental Standards Compliance

Unit Objectives



By the end of this unit, the participants will be able to:

1. Understand and adhere to environmental regulations and guidelines promoting digital sustainability.
2. Learn to apply sustainable software development frameworks and tools during application design and development.
3. Implement green software engineering best practices to minimize environmental impact in software operations.

5.4.1 Environmental Regulations and Guidelines Related to Digital Sustainability

Complying with environmental standards in software development is becoming increasingly important as the digital industry's carbon footprint grows. This involves adhering to regulations, leveraging sustainable tools, and adopting green engineering practices.

While specific environmental regulations for software are still evolving compared to physical industries, there are growing guidelines and standards that software developers and organizations should be aware of.

- **Regional and National Regulations:** Keep abreast of digital sustainability initiatives and regulations in relevant regions (e.g., EU's Digital Strategy, national circular economy policies). These might influence data center locations, energy efficiency reporting, or responsible e-waste management.
- **Industry Standards and Best Practices:** Adhere to established industry best practices for data center efficiency (e.g., PUE — Power Usage Effectiveness standards) even if your software doesn't directly manage hardware, as your software's demands contribute to these metrics.
- **Data Protection and Privacy Laws:** While not directly "environmental," regulations like GDPR (General Data Protection Regulation) encourage efficient data management and deletion, which indirectly reduces storage needs and energy consumption associated with unnecessary data.
- **Accessibility Standards:** Designing accessible software can indirectly lead to more efficient designs, as it often encourages simpler interfaces and less resource-intensive features.
- **Voluntary Commitments and Certifications:** Participate in voluntary initiatives like the Green Software Foundation's principles or aim for certifications related to sustainable IT, demonstrating a commitment to environmental responsibility.



Fig 5.4.1 Digital Sustainability Policy Framework

5.4.2 Software Development Frameworks and Tools to Support Sustainability

The tools and frameworks chosen for software development can significantly impact the environmental footprint of the resulting application. Selecting green-conscious options can reduce energy, data, and resource consumption.

- **Efficient Programming Languages:** Some languages are inherently more energy-efficient than others. For example, Rust, C++, and Go are often cited for their performance and lower energy consumption compared to more interpreted languages, though the specific use case and implementation greatly matter.
- **Optimized Frameworks and Libraries:** Choose frameworks and libraries that are known for their performance and resource efficiency. Avoid bloated or heavy frameworks when a lighter alternative can achieve the same functionality.
- **Cloud Provider Sustainability:** When using cloud services, consider the sustainability practices of the cloud provider (e.g., use of renewable energy, efficient data center designs). Many providers now offer reporting on the carbon emissions of your cloud usage.
- **Performance Monitoring Tools:** Utilize tools that measure and report on energy consumption, CPU usage, memory footprint, and network traffic of your applications. This data is crucial for identifying bottlenecks and areas for optimization.
- **CI/CD Pipelines for Efficiency:** Configure continuous integration/continuous deployment (CI/CD) pipelines to include performance and efficiency checks, ensuring that new code doesn't introduce regressions in resource usage.



Fig 5.4.2 Sustainable Development Toolchain

5.4.3 Best Practices Recommended by Green Software Engineering Standards

Green Software Engineering (GSE) is an emerging discipline focused on building sustainable software. Adhering to its best practices helps minimize the environmental impact of software throughout its lifecycle.

- **Carbon Efficiency:** Design software to use less energy. This includes optimizing algorithms, reducing computational complexity, and leveraging efficient hardware.
- **Energy Proportionality:** Strive for systems where energy consumption scales linearly with utilization. An idle system should consume minimal energy. This is particularly important for cloud-based applications.
- **Data Minimization:** Only collect, store, and process data that is absolutely necessary. Less data means less storage, less network transfer, and less processing, all of which save energy.
- **Hardware Efficiency:** Write code that makes efficient use of underlying hardware. For example, optimize for CPU cache, reduce memory allocations, and minimize I/O operations.
- **Cloud Awareness:** Deploy applications to cloud regions powered by renewable energy. Configure cloud resources to auto-scale down or even shut down when not in use.
- **Measurement and Monitoring:** Regularly measure the carbon emissions and energy consumption of your software. Use this data to identify areas for improvement and track progress.
- **Continuous Improvement:** Green software engineering is an ongoing process. Regularly review and refactor code, infrastructure, and deployment strategies to continuously reduce environmental impact.

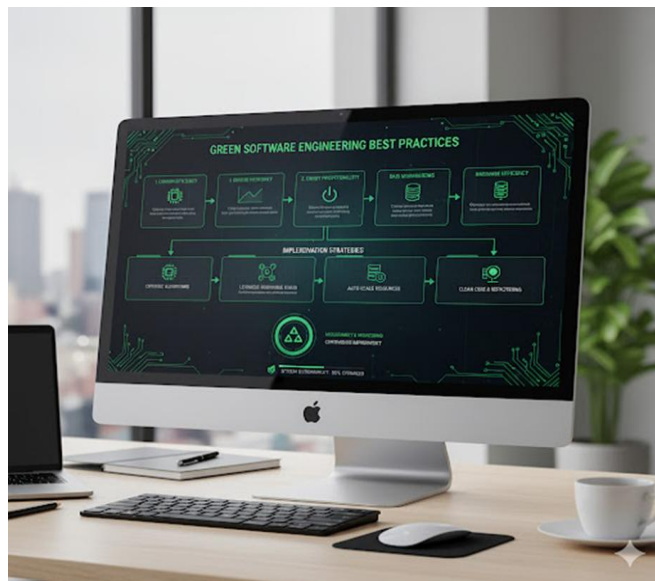


Fig 5.4.3 Best Green Software Engineering Practices

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 5.5: Sustainable UI/UX Design

Unit Objectives



By the end of this unit, the participants will be able to:

1. Learn to design energy-efficient user interfaces using dark mode and adaptive themes.
2. Understand methods to optimize animations and visual effects for reduced processing and power consumption.
3. Learn skills to enable user-controlled performance settings that enhance energy efficiency and sustainability.

5.5.1 Dark Mode and Adaptive Themes to Reduce Screen Energy Consumption

Sustainable UI/UX design goes beyond aesthetics and usability; it incorporates principles that minimize the environmental impact of digital products. By making conscious design choices, designers and developers can contribute to energy efficiency and a greener digital ecosystem.

The screen is often the most power-hungry component of a mobile device or monitor. UI/UX choices, particularly color schemes, can significantly impact its energy consumption.

- **Dark Mode (or Dark Theme):** For devices with OLED or AMOLED screens (common in modern smartphones), pixels display black by turning completely off, consuming no power. Therefore, interfaces with a predominantly dark background can lead to substantial energy savings compared to light themes.
- **Benefits:** Reduces battery drain on OLED/AMOLED screens, less eye strain in low-light conditions, and can improve perceived performance.
- **Adaptive Themes:** Allow users to choose between light and dark modes, or automatically switch based on ambient light conditions or system preferences. This provides flexibility and maximizes energy savings when applicable.
- **Minimalist Design:** Beyond just dark mode, a clean and minimalist design with fewer vibrant colors and simpler graphics can also contribute to lower energy consumption by reducing the rendering complexity.



Fig 5.5.1 Light Mode vs. Dark Mode Interface

5.5.2 Animation and Visual Effect Optimization

While animations and visual effects can enhance user experience, poorly optimized ones can consume significant CPU and GPU resources, leading to increased power consumption and potentially device overheating.

- **Purposeful Animations:** Use animations only when they serve a clear purpose (e.g., providing feedback, guiding attention, improving perceived performance). Avoid gratuitous or overly complex animations.

Performance-Optimized Techniques:

- **CSS Transforms and Opacity (Web):** Prioritize animating properties like transform (for position, scale, rotation) and opacity as they can often be handled by the GPU, leading to smoother performance and lower CPU usage. Avoid animating properties that trigger layout or paint (e.g., width, height, margin, padding).
- **Native Platform APIs:** Leverage native animation frameworks (e.g., Core Animation on iOS, Android's animation system) which are highly optimized for device hardware.
- **Reduce Frame Rate:** For non-critical animations, consider reducing the frame rate. A smooth 30fps animation can often be perceived as equally good as 60fps while consuming less power.
- **"Reduce Motion" Settings:** Respect system-wide "reduce motion" preferences (available on most operating systems) by offering a less intensive visual experience for users who prefer it.
- **Lazy Loading of Visual Assets:** Only load complex visual assets or start computationally intensive effects when they are actually in the user's viewport or explicitly triggered.

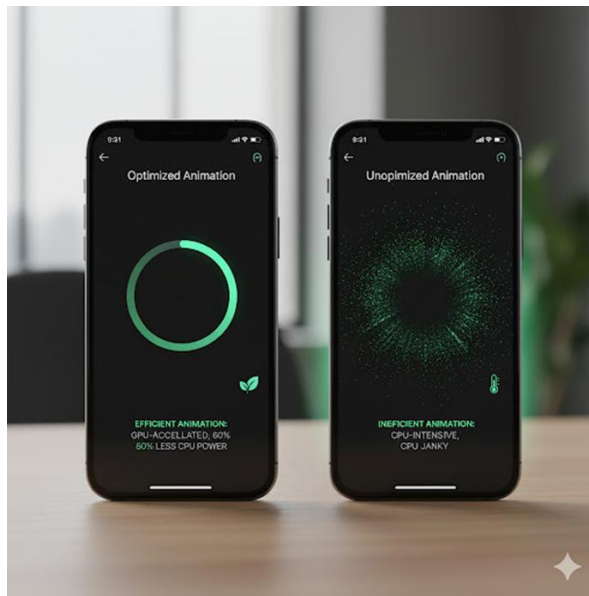


Fig 5.5.2 Animated UI Element

5.5.3 Customized App Settings for Energy Efficiency

Empowering users with control over how an app consumes resources can significantly contribute to overall energy efficiency and user satisfaction.

Performance/Battery Saving Modes: Offer distinct modes, such as "High Performance" or "Battery Saver," that adjust various app behaviors. In battery saver mode, the app might:

- Reduce animation intensity.
- Lower image quality or resolution.
- Decrease background refresh rates.
- Limit GPS or sensor usage.
- Reduce network data fetching frequency.
- Customizable Sync Settings: Allow users to define how often data is synchronized, whether to sync only on Wi-Fi, or to manually trigger syncs.
- Image/Video Quality Options: For content-heavy apps, provide settings to choose default streaming or download quality (e.g., "Standard Definition" vs. "High Definition").
- Background Activity Control: Give users explicit control over whether the app can run in the background, send notifications, or use location services. (Note: Operating systems often provide some of these controls, but app-specific settings can offer finer granularity.)
- Theme Preferences: Allowing users to select dark/light modes or adaptive themes directly impacts screen energy consumption.

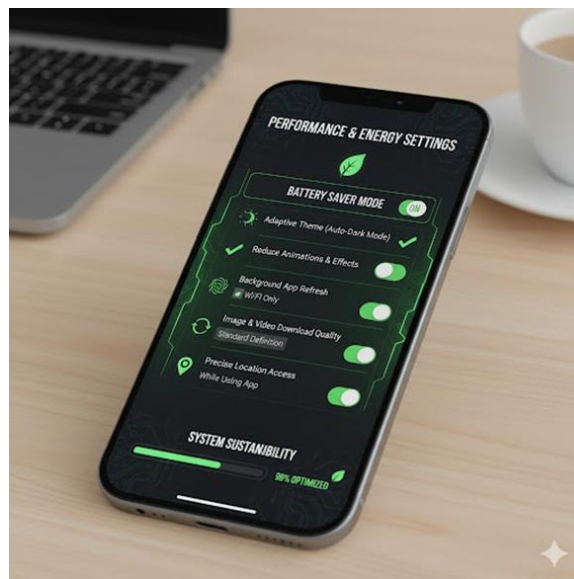


Fig 5.5.3 App Settings

Notes



Lined area for taking notes, consisting of multiple horizontal lines.

UNIT 5.6: Sustainable Development Practices

Unit Objectives

By the end of this unit, the participants will be able to:

1. Understand how to leverage cloud computing solutions powered by renewable energy to support sustainable development.
2. Learn to adopt remote work and digital collaboration tools that contribute to reducing the carbon footprint.
3. Apply practices to reuse existing components and libraries for minimizing digital waste and resource consumption.

5.6.1 Cloud Computing Solutions that Operate on Renewable Energy

Following sustainable development practices involves making conscious choices in infrastructure, work culture, and code utilization to minimize the environmental impact of software creation and operation.

The infrastructure that powers your application—primarily cloud data centers—is a major source of its carbon footprint. Choosing a sustainable cloud provider is a critical step in green software engineering.

- **Choose Green Cloud Providers:** Select cloud vendors (AWS, Azure, Google Cloud, etc.) that publicly commit to and actively invest in 100% renewable energy for their data centers. Many providers now offer reporting tools that allow you to estimate the carbon emissions associated with your specific usage.
- **Select Low-Carbon Regions:** When deploying your application, choose cloud regions where the local electrical grid uses a high percentage of renewable energy (hydro, wind, solar). Cloud providers often publish the carbon intensity of different regions.
- **Optimize Cloud Resources:** Even with green energy, minimizing resource usage is vital. Employ strategies like:
 - **Auto-scaling:** Scale down or shut down virtual machines and containers during low-traffic periods (energy proportionality).
 - **Serverless Computing:** Use serverless functions (e.g., AWS Lambda, Azure Functions) that only consume energy while executing code, leading to greater efficiency than always-on servers.
 - **Containerization:** Use efficient container orchestration (like Kubernetes) to maximize hardware utilization, reducing the total number of physical servers needed.

5.6.2 Remote Work and Digital Collaboration Tools to Reduce Carbon Footprint

The carbon footprint of a software company isn't just in its code and servers; it also includes the emissions from employee commuting and office operations. Remote and digital-first work models significantly reduce this impact.

- **Reduce Commuting Emissions:** Remote work eliminates or drastically reduces the need for employees to travel to a central office, cutting down on vehicle fuel consumption and public transport energy use.

Digital-First Operations:

- **Paperless Processes:** Utilize digital documentation, e-signatures, and cloud storage to minimize paper consumption and printing energy.
- **Efficient Collaboration Tools:** Leverage project management, video conferencing, and chat tools (Slack, Teams, Jira) that enable effective communication without the need for physical travel or large, resource-intensive office spaces.
- **Optimize Office Space:** For hybrid models, implement smart office technology (e.g., smart lighting, HVAC controls) and downsize office space to reduce the energy consumption from heating, cooling, and lighting per employee.
- **Reduce Business Travel:** Prioritize video conferencing over inter-city or international business flights, which have a very high carbon impact per person.

5.6.3 Reuse of Existing Components and Libraries

In software, "waste" often refers to the duplicated effort of writing code that already exists, which leads to increased maintenance overhead, larger application size, and unnecessary consumption of developer time and computational resources.

- **Component-Based Architecture:** Design applications using modular components that can be easily shared and reused across different parts of the application or even different projects. This follows the principle of "Don't Repeat Yourself" (DRY).
- **Leverage Open-Source Libraries (OSS):** Utilize well-maintained, battle-tested open-source libraries and frameworks for common tasks (e.g., logging, data validation, UI components). This minimizes the need to write, debug, and secure new code from scratch, saving computational resources and developer time.
- **Internal Component Repository:** For large organizations, establish an internal component library or code repository to easily share and discover reusable code assets, themes, and utility functions across internal teams.
- **Avoid Code Bloat (Legacy Removal):** Regularly audit the codebase to identify and remove unused code, features, or deprecated libraries. This reduces application size, compilation time, and memory footprint, which directly contributes to resource efficiency.
- **Upgrade, Don't Rewrite:** Focus on upgrading and maintaining existing, stable software systems rather than initiating complete rewrites unless absolutely necessary. Rewrites consume significant resources and often lead to new inefficiencies.

Exercise

Multiple Choice Questions (MCQs):

1. Which of the following best describes sustainable coding?
 - a) Writing code that uses maximum memory and CPU power
 - b) Writing code that minimizes energy and resource consumption
 - c) Writing code with complex algorithms for faster results
 - d) Writing long code for better readability
2. What is the main goal of optimizing API calls in a mobile app?
 - a) To increase network requests
 - b) To reduce data transfer and improve speed
 - c) To store all data permanently on the device
 - d) To disable background synchronization
3. Which of the following caching strategies serves stale content while updating it in the background?
 - a) Lazy Loading
 - b) Pre-fetching
 - c) Stale-While-Revalidate
 - d) On-Demand Fetching
4. The term Power Usage Effectiveness (PUE) is related to:
 - a) Data compression
 - b) Data center efficiency
 - c) API optimization
 - d) Cloud storage encryption
5. On OLED screens, dark mode helps conserve energy because:
 - a) The brightness level increases automatically
 - b) Black pixels consume no power
 - c) The display uses a white backlight
 - d) It switches off animation processing
6. Which of the following practices helps reduce a software project's carbon footprint?
 - a) Increasing server load time
 - b) Rewriting all code from scratch
 - c) Using renewable energy-powered cloud providers
 - d) Disabling caching

Fill in the Blanks:

1. Optimized code helps reduce _____ load and battery consumption.
2. Using efficient data formats like _____ over XML can reduce processing time and bandwidth usage.
3. Following environmental regulations ensures compliance with _____ standards in digital sustainability.
4. Implementing _____ mode helps lower energy use on AMOLED screens.
5. Reusing existing components and _____ minimizes digital waste and resource consumption.

Short Answer Questions:

1. Explain two benefits of writing optimized and efficient code in mobile app development.
2. What are two methods developers can use to minimize redundant network requests?
3. How do green software engineering practices contribute to environmental sustainability?
4. Mention two ways users can control app performance settings to improve energy efficiency.
5. Describe how cloud computing solutions that use renewable energy sources promote sustainable development.
6. What are the environmental benefits of adopting remote work and digital collaboration tools?

Notes



Lined area for taking notes, consisting of multiple horizontal lines.





6. Employability Skills (30 Hours)

It is recommended that all training include the appropriate. Employability Skills Module. Content for the same can be accessed
<https://www.skillindiadigital.gov.in/content/list>














7. Annexure





Annexure I - QR Codes –Video Links



Annexure I

QR Codes –Video Links

Chapter No.	Unit Name	Topic	URL Links	QR code (s)
1. Role and Responsibilities of an Android Application Technician – Telecom Devices	Unit 1.1 – Introduction to the Program	Telecom Industry Overview	https://www.youtube.com/watch?v=jmymVdxbKWU&t=24s	 Telecom Industry Overview
		Telecommunication process	https://www.youtube.com/shorts/XpXZd7jC3Cs	 Telecommunication process
	UNIT 1.2: History of Communication	History of Communication	https://www.youtube.com/watch?v=E4xK2r4MO3U	 History of Communication
	UNIT 1.4: Networks	Mobile Telecommunication	https://www.youtube.com/watch?v=ZpcEOwdQ12U	 What is mobile telecommunication technologies
	UNIT 1.5: Channel Access Methods	Multiplexing	https://www.youtube.com/watch?v=cEOuzBRJ_gY	 Multiplexing
	UNIT 1.7: Legacy Mobile Operating Systems and Concepts	Virtual Private Networking (VPN)	http://youtube.com/watch?v=JFXBjIT5cGU	 What is VPN
	UNIT 1.8: Android Operating System and Version History	App Development Process	https://www.youtube.com/watch?v=4JqhErux_P8	 App Development Process

Chapter No.	Unit Name	Topic	URL Links	QR code (s)
	UNIT 1.10: Android Studio Installation and Setup	How to Install Android Studio	https://www.youtube.com/watch?v=6uZPVmOzUEA	 How to Install Android Studio
2. Setting up Android framework/ Development Environment and creating user interface (TEL/N2300)	UNIT 2.1: Creating Simple Android Project in Android Studio	Creating a Simple Android Project	https://www.youtube.com/watch?v=-NsdRbi4sY	 How to Create Your First Android Project in Android Studio
3. Configuring Value Added Services (VAS) in Android Applications for Telecom Devices (TEL/N2301)	UNIT 3.1: Managing Data within Android Applications	Android App Activity Lifecycle	https://www.youtube.com/watch?v=1sjA4e_wG3w	 Android App Activity Lifecycle
5. Follow Sustainability Practices in Telecom Cabling Operations	Unit 5.1 - Sustainability Practices in Telecom Cabling Operations	What Are Regulatory Compliance Requirements?	https://www.youtube.com/watch?v=WvyWmXrDTwM	 What Are Regulatory Compliance Requirements?





Telecom Sector Skill Council

Estel House, 3rd Floor, Plot No: - 126, Sector-44

Gurgaon, Haryana 122003

Phone: 0124-2222222

Email: tssc@tsscindia.com

Website: www.tsscindia.com