**SUMMER INTERNSHIP**

|  |  |  |
| --- | --- | --- |
| Name of the Student | : | Pulidindi Prasanth Babu |
| Name of the Course | : | Google Android Development Virtual Internship |
| Name of the College | : | Vishnu Institute of Technology |
| Registration Number | : | 22PA1A05D6 |
| Period of Internship | : | 10 Weeks (July 2024 – September 2024) |
| Name and Address of Intern Organization | : | AICTE-EduSkills Virtual Internship  (Cohort-8) |



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**VISHNU INSTITUTE OF TECHNOLOGY (AUTONOMOUS)**

**(Approved by AICTE, Accredited by NBA & NAAC and permanently affiliated to JNTUK)**

**BHIMAVARAM – 534 202**

**2024 – 2025**

**GOOGLE ANDROID DEVELOPMENT VIRTUAL INTERNSHIP**

*An Internship report submitted in partial fulfillment of the requirements for the degree of*

##### BACHELOR OF TECHNOLOGY

In

##### COMPUTER SCIENCE AND ENGINEERING

###### Submitted by

###### PULIDINDI PRASANTH BABU

###### Reg.No : 22PA1A05D6

**Under the Faculty Guidance of**

##### Mr.D.Shankar

###### Assistant Professor

###### 

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**VISHNU INSTITUTE OF TECHNOLOGY (AUTONOMOUS)**

**(Approved by AICTE, Accredited by NBA & NAAC and permanently affiliated to JNTUK)**

**BHIMAVARAM – 534 202**

**2024 – 2025**

## VISHNU INSTITUTE OF TECHNOLOGY (AUTONOMOUS)

**(Approved by AICTE, Accredited by NBA & NAAC and permanently affiliated to JNTUK)**

**BHIMAVARAM- 534202**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**



**DECLARATION**

I, **PULIDINDI PRASANTH BABU, REGD.NO: 22PA1A05D6** a student of **III Year** **BACHELOR OF TECHNOLOGY** in **COMPUTER SCIENCE AND ENGINEERING, VISHNU INSTITUTE OF TECHNOLOGY** do hereby declare that I have completed the mandatory internship from the period of **July 2024 to September 2024** in **Google** **Android Development Virtual Internship** under the Faculty Guidance of **Mr.D.Shankar**, **Department of Computer Science and Engineering, Vishnu Institute of Technology.**

**PULIDINDI PRASANTH BABU**

**Date:**

## VISHNU INSTITUTE OF TECHNOLOGY (AUTONOMOUS)

**(Approved by AICTE, Accredited by NBA & NAAC and permanently affiliated to JNTUK)**

**BHIMAVARAM- 534202**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**



**CERTIFICATE**

This is to certify that the Summer Intern Project Report entitled “**ANDROID DEVELOPMENT**”, is being submitted by **PULIDINDI PRASANTH BABU, REGD.NO: 22PA1A05D6** submitted in fulfillment for the degree of “**BACHELOR OF TECHNOLOGY”** in “**COMPUTER SCIENCE AND ENGINEERING”** is a record of summer Intern Project work carried out by her under my supervision during the academic year 2023-2024 and it has been found worthy of acceptance according to the requirements of the university.

##### Internal Examiner Head of the Department

**External Examiner**

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S. No** | **Topics** | **Page.No** |
| **1.** | **Android Application**   * Write simple Kotlin programs that display text output. * Download and install Android Studio. * Build an Android app with a simple user interface that displays text and images. * Run the app on a device or emulator**.** |  |
| **2.** | **Building app UI**   * Use conditionals, function types, classes, and lambda expressions in Kotlin. * Understand how composition and recomposition works. * Add a button to an app UI and respond to user taps. * Create an app that works with data entered by the user. * Learn how to use state to display data and reflect the changes automatically when the data gets updated. |  |
| **3.** | **Display Lists and use Material Design**   * Use data classes, functions, and collections in Kotlin. * Create a scrollable list in an app that displays both text and images. * Add click listeners to interact with list items. * Add an app bar to the app and modify the app theme. * Use Material Design to build modern and intuitive user interfaces, using colors, shapes and typography. |  |
| **4.** | **Navigation and app architecture**   * Explain activities and their lifecycles. * Understand Modern Android architecture. * Use StateFlow and UDF pattern to work with state and events. * Add a ViewModel to save data and state. * Set up and use the Navigation component with Compose. * Understand what responsive UI is. * Use window class sizes to build for different screen sizes. * Add a navigation drawer to an app. |  |
| **5.** | **Connect to the Internet**   * Describe the basics of concurrency and how to use coroutines in an Android app. * Define and understand the data layer in Modern Android app architecture. * Implement a repository to centralize data access. * Use Retrofit to retrieve data from a remote server. * Load and display images using the Coil library. * Implement dependency injection to decouple the classes, making it easier to test, maintain, and scale the app. |  |

|  |  |  |
| --- | --- | --- |
| **6** | **Data persistence**   * Learn the basics of SQL to insert, update, and delete data from a SQLite database. * Use the Room library to add a database to an Android app. * Use Database Inspector to test and debug database issues. * Use Preference DataStore to store user preferences. |  |
| **7** | **WorkManager**   * Define long running tasks that need to run in background work. * Add WorkManager to an Android app. * Create a Worker object and enqueue work. * Create constraints on WorkRequests. * Use the Background Task Inspector to inspect and debug WorkManager. |  |
| **8** | **Views and Compose**   * Understand the View-based UI toolkit and build app UI   using XML.   * Add a composable in an app built with Views. * Add Navigation component to the app and use it to navigate between fragments. * Use AndroidView to display views. * Add existing View-based UI components in a Compose app. |  |

**CHAPTER - 1**

**INTRODUCTION TO ANDROID DEVELOPMENT**

**Android Development**

Android development is the process of creating applications for devices that run the Android operating system, primarily using programming languages like Java and Kotlin, with Kotlin being the preferred choice due to its modern features. Developers typically use Android Studio, the official Integrated Development Environment (IDE), which provides essential tools for coding, debugging, and testing. The development process often involves using frameworks and libraries, such as Jetpack for simplifying common tasks, Retrofit for network operations, and Room for database management. User interfaces are designed using XML layouts, and common architectural patterns like MVC, MVP, and MVVM help structure the code effectively. Android apps can interact with various APIs and services, including Google Play Services for enhanced functionality. Testing is crucial, with frameworks like JUnit and Espresso used to ensure quality and performance. Finally, apps can be published on the Google Play Store, allowing developers to manage versions and user feedback effectively. Overall, Android development blends software engineering principles with mobile-specific design and user experience considerations.

**Introduction to Kotlin**

Key Features of Kotlin in Android Development:

**Concise Syntax**: Kotlin reduces boilerplate code significantly, which allows developers to write more expressive and readable code compared to Java. This leads to fewer errors and faster development cycles.

**Interoperability with Java**: Kotlin is 100% interoperable with Java. This means you can use both Kotlin and Java in the same Android project. Existing Java codebases can incrementally migrate to Kotlin without rewriting the entire codebase.

**Null Safety**: Kotlin eliminates the common problem of null pointer exceptions by making all types non-nullable by default. If you want to allow a variable to be null, you must explicitly declare it as nullable (String?).

**Coroutines for Asynchronous Programming**: Kotlin provides built-in support for coroutines, which is a more efficient and simpler way to manage asynchronous tasks in Android compared to Java's thread management. Coroutines simplify working with background tasks like network calls and file operations.

**Higher-Order Functions and Lambda Expressions**: Kotlin treats functions as first-class citizens, allowing developers to pass functions as parameters or return them. This makes it easy to use functional programming techniques and simplifies Android tasks such as setting listeners or configuring adapters.

**Android Extensions**: Kotlin Android Extensions allow developers to directly access views from their XML layouts without using findViewById(), making code cleaner and more readable. This feature is largely superseded by Jetpack View Binding, but it demonstrates Kotlin’s focus on reducing boilerplate code.

**Data Classes**: In Kotlin, you can declare data classes, which are simple classes used to hold data. These classes automatically provide getters, setters, equals(), hashCode(), and toString() methods, all without the need for additional code.

**Download and install Android Studio**

Download Android Studio

1. Open any web browser and navigate to the Android studio development page.

This is the Android Developers website, where you can download Android Studio. This page automatically detects your operating system.

1. Click Download Android Studio. The Terms and Conditions page with the Android Studio License Agreement opens.
2. Read the License Agreement.
3. At the bottom of the page, if you agree with the terms and conditions, select the I have read and agree with the above terms and conditions checkbox.
4. Click Download Android Studio to start the download.
5. When prompted, save the file to a location where you can easily locate it, such as the Downloads folder.
6. Wait for the download to complete. This may take a while and may be a good moment to enjoy some tea!

Install Android Studio on Windows

1. Open the folder where you downloaded and saved the Android Studio installation file.
2. Double-click the downloaded file.
3. If you see a **User Account Control** dialog about allowing the installation to make changes to your computer, click **Yes** to confirm the installation.

The **Welcome to Android Studio Setup** dialog displays.

1. Click **Next** to start the installation.
2. Accept the default installation settings for all steps.
3. Click **Finish** when the installation is done to launch Android Studio.
4. Choose your preference of light or dark theme when Android Studio first launches. Screenshots in this course use the light theme, but choose whichever one you prefer.
5. During the installation, the setup wizard downloads and installs additional components and tools needed for Android app development. This may take some time depending on your internet speed. During this time, you may see a **User Account Control** dialog for **Windows Command Processor**. Click **Yes** to accept the dialog.
6. You may also receive a **Windows Security Alert** about adb.exe. Click **Allow Access,** if needed, to continue the installation.
7. When the download and installation completes, click **Finish**.
8. The **Welcome to Android Studio** window displays and you're ready to start creating apps!

**Build an Android app with a simple user interface that displays text and images.**

**Steps to Build the App**

**1. Set Up Android Studio and Create a New Project**

* **Open Android Studio** and start a new project by selecting **New Project**.
* Choose an **Empty Activity** and click **Next**.
* Give your project a name (e.g., "TextImageApp").
* Set the language to **Kotlin**.
* Select the minimum API level (you can choose API 21 or higher to support most devices).
* Click **Finish** to create the project.

**2. Design the User Interface (UI)**

Android’s UI is built using XML (Extensible Markup Language). In the layout file, you can define how the app looks, placing components like TextViews (for text) and ImageViews (for images).

* **Open res/layout/activity\_main.xml** in the app module.
* You can either use **Design mode** (drag-and-drop interface) or **Code mode** (edit the XML directly).

**3. Add an Image to the App**

* Place your image file (e.g., sample\_image.png) into the **res/drawable** directory. You can right-click on the **drawable** folder, select **New**, and then **Image Asset** or simply paste the image into the folder.

**4. Modify the MainActivity (Kotlin Code)**

After designing the UI, you need to connect the XML layout to Kotlin code in your MainActivity.kt file. This is where you can dynamically update UI elements (if necessary).

**5. Run the App**

* Connect your Android device via USB, or you can use an Android emulator within Android Studio.
* Click the **Run** button (green arrow) in Android Studio to build and launch the app.
* The app should display the text "Hello, welcome to my app!" along with the image you added in the center of the screen.

**6. Adding Interactivity (Optional)**

If you want to add interactivity, such as changing the text when an image is clicked, you can modify the MainActivity.kt like this:

**7.Testing and Debugging**

* Test the app on different screen sizes using the Android Studio emulator or real devices.
* Make sure the image and text are displayed correctly and that any interactions (like clicks) work as expected.

**Run the app on a device or emulator.**

**1. Set Up an Android Emulator**

Android Studio comes with the **Android Virtual Device (AVD) Manager**, which allows you to create and run an emulator that mimics a real Android device.

1. **Open the AVD Manager**:
   * In Android Studio, go to **Tools > Device Manager** or click the **Device Manager** icon from the toolbar (it looks like a phone).
   * This opens the AVD Manager where you can create and manage virtual devices.
2. **Create a New Virtual Device**:
   * Click **Create Device**.
   * Choose a device from the list (e.g., Pixel 3, Nexus 5X) and click **Next**.
   * Select a **system image** (Android version). You might need to download it if it’s not available. For example, select **Android 11.0 (R)** if it's compatible with your project.
   * Click **Next**, then **Finish** to create the virtual device.
3. **Launch the Emulator**:
   * After the virtual device is created, click the **Play** button next to the device in the AVD Manager to launch the emulator.
   * Wait for the emulator to boot up (it may take a minute for the first time).

**2. Run the App on the Emulator**

1. Once the emulator is running, make sure the device is selected in the **Target Device** dropdown menu (located at the top of the Android Studio window).
2. Click the **Run** button (green play arrow) or go to **Run > Run ‘app’** from the toolbar.
3. Android Studio will build the project, install the app on the emulator, and launch it. You should see your app running on the emulator.

**Running the App on a Physical Android Device**

**1. Enable Developer Options on the Device**

* Go to the **Settings** app on your Android device.
* Scroll down to **About phone**.
* Find **Build Number** and tap it **7 times**. This will enable **Developer Options**.
* Go back to the **Settings** menu and open **Developer Options**.
* Enable **USB Debugging**.

**2. Connect the Device to Your Computer**

* Connect your Android device to your computer using a USB cable.
* You may get a prompt on your phone asking for permission to allow **USB Debugging**. Click **OK** to authorize.

**3. Run the App on the Device**

1. In Android Studio, your connected device should appear in the **Target Device** dropdown (next to the Run button). If not, click the dropdown and select your device.
2. Click the **Run** button (green play arrow), or go to **Run > Run ‘app’**.
3. Android Studio will build the app and install it on your device.
4. After installation, the app will automatically launch on your phone, and you’ll be able to see it running.

**Common Issues and Troubleshooting**

* **Emulator Performance Issues**: If the emulator is running slowly, try allocating more RAM to the virtual device, or enable **hardware acceleration**. Ensure **Intel HAXM** or **Android Emulator Hypervisor Driver** is installed for better performance.
* **Device Not Detected**:
  + Ensure **USB Debugging** is enabled.
  + Try using a different USB cable or port.
  + On your phone, check if you’ve granted permission to allow debugging with the connected computer.
* **APK Installation Failure**: If the app fails to install, check the logcat for error messages, and make sure the app's minimum SDK level matches or is lower than the device’s Android version.

By following these steps, you should be able to run your Android app either on a real device or on an emulator with ease.

**CHAPTER - 2**

**Building app UI**

**Use conditionals, function types, classes, and lambda expressions in Kotlin.**

**Conditionals:**

Kotlin provides conditional statements like if-else and when to control the flow of a program based on different conditions:

* **if-else**: A basic conditional statement to execute one block of code if a condition is true, and another if it's false.
* **when**: A more flexible alternative to if-else or switch statements from other languages. It checks for multiple conditions and executes the corresponding block of code when a match is found. It can handle various types of inputs and conditions, such as values, ranges, or even arbitrary expressions.

**2. Function Types:**

Kotlin treats functions as first-class citizens, meaning functions can be assigned to variables, passed as arguments, or returned from other functions. These are represented by **function types**. For example, a function that takes two integers and returns an integer has a function type like (Int, Int) -> Int. This allows you to write flexible code where you can pass in different behaviors (functions) without changing the structure of the code.

**3. Classes:**

Kotlin supports object-oriented programming through **classes**, which are blueprints for creating objects. A class can define properties (data) and methods (functions) to represent and operate on real-world entities. Kotlin allows you to define constructors for initializing the class with values and methods for performing actions with the class’s properties. Kotlin also supports powerful features like data classes, which are used to hold data and provide convenient built-in methods (e.g., equals, toString).

**4. Lambda Expressions:**

A **lambda expression** is essentially an anonymous function that can be treated as a value. Lambdas are often used to pass behavior as arguments to functions. For instance, higher-order functions (functions that accept other functions as parameters) often rely on lambda expressions to specify the logic without having to define a separate named function. Lambdas are concise, making them ideal for short, throwaway functions that perform simple tasks like filtering or transforming data.

**How They Work Together:**

* **Conditionals** allow decision-making in your code, responding dynamically based on input or state.
* **Function types** let you pass around functions as arguments, allowing for high levels of code flexibility and reuse.
* **Classes** let you encapsulate data and behavior, providing a structured way to model real-world concepts.
* **Lambdas** make your code concise and expressive, especially when working with collections or higher-order functions.

These features together help write clean, modular, and flexible Kotlin programs.

**Understand how composition and recomposition works:**

Composition and Recomposition are key concepts in declarative programming, particularly in modern UI frameworks like Jetpack Compose (for Android) and SwiftUI (for iOS). Let’s break down both:

**Key Ideas in Composition:**

* **Reusable Components**: Each part of the UI is broken down into small, reusable components (or composables, in Jetpack Compose).
* **Declarative UI**: You declare what the UI should look like for a given state, and the framework takes care of rendering it.
* **Hierarchical Structure**: UIs are built from smaller components that are composed together, creating a hierarchy. For example, a screen may be composed of a header, a body, and a footer, each of which is further composed of smaller elements like text, buttons, or images.

**Key Ideas in Recomposition:**

* **State-driven UI**: The UI is automatically updated when the state it depends on changes. For example, if a button click changes the state of a variable, only the components that depend on that variable are redrawn.
* **Efficient Rendering**: Instead of redrawing the entire UI, only the components that need updating are recomposed, making the UI more efficient.
* **Automatic Detection**: Frameworks like Jetpack Compose automatically detect which parts of the UI need to be recomposed based on changes to the state or data.

**Add a button to an app UI and respond to user taps Top of Form**

**Set up Jetpack Compose**: Ensure that your Android project is set up for Jetpack Compose. Add the required dependencies for Compose in your build.gradle file, if not already done.

**Create a Button**: Use the Button composable to add a button to your UI. This button takes an onClick lambda, where you define what should happen when the button is tapped.

**Respond to Taps**: You can maintain the state (e.g., a counter, a message) and update it when the button is clicked. The UI will automatically recompose when the state changes.

Bottom of Form

**Create an app that works with data entered by the user**

To create an app that works with user-entered data, you'll need to design a user interface (UI) that allows for data input, typically using components like text fields for typing information, selection widgets such as checkboxes or dropdowns, and buttons to submit or interact with the input. Once the input interface is in place, you need to capture the user's input and manage it through state variables, ensuring that the app dynamically reflects any changes in the UI as the user types or selects options. It's essential to validate the data to ensure correctness, such as checking for empty fields or improper formats (e.g., invalid emails).

When the user submits the data, the app should handle this action by either updating the UI, processing the data (e.g., saving it locally or sending it to a server), or performing calculations based on the input. To make the app responsive, state-driven UI updates ensure that any change in the data is immediately reflected in the interface. You may also want to store the data for future use, using local storage (e.g., a database or shared preferences) or sending it to remote storage (e.g., cloud servers).

Additionally, the app should provide feedback to the user, such as success messages when data is submitted correctly or error messages if the input is invalid. Proper feedback creates a smooth user experience. Finally, testing the app’s behavior with various types of input ensures it can handle all edge cases, such as empty fields or incorrect data formats. This approach allows you to build an app that effectively captures, processes, and responds to user input in a flexible and user-friendly way.

**Learn how to use state to display data and reflect the changes automatically when the data gets updated**.

1. **Understanding State**:
   * **State** refers to data that can change over time. In user interfaces, it is used to hold dynamic values that directly affect how the UI looks and behaves.
2. **State Management**:
   * Use state variables to store the data that will drive the UI. In modern frameworks like Jetpack Compose, this is typically done using constructs like mutableStateOf to declare state.
3. **Binding State to UI**:
   * Link your state variables to the UI components (e.g., Text fields, Buttons). When the state changes, the UI components using that state will automatically reflect the updated values.
4. **Recomposition**:
   * When the state changes, the UI is "recomposed," meaning only the parts of the UI that depend on that state are redrawn, ensuring efficient updates without redrawing the entire interface.
5. **User Interactions and State Updates**:
   * State is often updated in response to user actions (like typing in a text field or pressing a button). When the user input changes the state, the UI will automatically re-render to reflect those changes.

**CHAPTER - 3**

**Display Lists and use Material Design**

In Kotlin, data classes, functions, and collections are essential features that make the language concise and powerful. Let’s go over each one briefly and see how they are used.

**1. Data Classes**

Kotlin’s data class simplifies creating classes meant to hold data by automatically providing useful methods such as equals(), hashCode(), toString(), copy(), and componentN().

**2.Functions**

Functions in Kotlin are first-class citizens, meaning they can be stored in variables and passed as arguments to other functions.

**3. Collections**

Kotlin collections provide a rich API to work with lists, sets, and maps.

**Common collections:**

* **List**: An ordered collection of elements.
* **Set**: A collection that contains no duplicate elements.
* **Map**: A collection of key-value pairs.

Kotlin collections have many powerful functions for filtering, transforming, and reducing data.

**Create a scrollable list in an app that displays both text and images.**

**Design the User Interface Layout**:

* Use a RecyclerView or ListView to create a scrollable list that can display multiple items.
* Each item in the list should contain a layout with an ImageView for displaying images and a TextView for displaying text.
* The layout of the list item should define how text and images are arranged (e.g., image on the left and text on the right, or image above text).

**Create a Data Model**:

* Define a data class to represent the information for each item. This class should contain fields for both the text and the image data (such as a URL or resource ID).

**Create a ViewHolder and Adapter**:

* Implement a RecyclerView.Adapter or ListAdapter to bind the data to the views.
* Inside the adapter, create a ViewHolder class to hold references to the TextView and ImageView for each list item.
* In the onBindViewHolder method, set the text and load the images into the ImageView. You might use an image loading library like **Glide** or **Picasso** for efficient image loading.

**Populate the List with Data**:

* In your activity or fragment, create a list of data objects (instances of the data model) and pass this list to the adapter.
* Set the adapter to the RecyclerView or ListView to display the scrollable list of items.

**Handle Image Loading**:

* If you are using online images (URLs), ensure images are loaded asynchronously to avoid blocking the UI thread. Libraries like **Glide** or **Picasso** handle this efficiently, supporting features like caching, resizing, and error handling.

**Optimize for Performance**:

* To improve performance, use RecyclerView with view recycling, ensuring that the RecyclerView.Adapter efficiently reuses views as the user scrolls.
* Implement placeholder images and caching strategies for better performance when loading images from remote sources.

**Add click listeners to interact with list items**.

### ****Define Click Listener Interface****

Create an interface to handle click events. This allows the interaction logic to be separated from the adapter and handled in the activity or fragment.

**Modify ViewHolder to Implement Click Listener**

In your ViewHolder class, implement the View.OnClickListener interface and define the logic for item clicks. The ViewHolder will handle item clicks and pass the event back to the activity/fragment through the OnItemClickListener.

**Pass Click Listener to Adapter**

In the adapter, pass the OnItemClickListener instance when creating the ViewHolder. This ensures that each list item can trigger click events.

**Implement Listener in Activity or Fragment**

In your activity or fragment, implement the OnItemClickListener interface to handle the click events for each item in the list. You can define what happens when an item is clicked

**Adding Click Listener to Specific Views**

If you want to add click listeners to specific views inside the list item (such as a button or image), you can do so in the ViewHolder. Here’s how you can add a click listener for an individual view within the item layout

**Add an app bar to the app and modify the app theme.**

**Add an App Bar (Toolbar) to the Layout**

* Integrate a Toolbar into the app's main layout to act as the app bar.
* The Toolbar will appear at the top of the screen and serve as the main UI element for navigation and interaction, offering features like a title, navigation buttons, and action items

**Set Up the Toolbar in the Activity**

* In the activity, set the Toolbar as the app's action bar using the appropriate method.
* Customize the toolbar by setting a title, subtitle, and potentially a logo or navigation icon.

**Customize the App Bar**

* You can modify the app bar’s appearance and functionality by changing its background color, text appearance, or adding icons and buttons.
* Include a navigation button (like a hamburger icon for a navigation drawer) or other interactive items (like a search button or settings icon) to enhance user experience.
* Customize the app bar further by inflating menu items, which provide additional actions users can perform from the app bar.

**Modify the App's Theme**

* Adjust the app's theme by defining styles in the theme configuration files.
* Customize the theme by specifying colors for primary and secondary UI elements, such as the app bar, buttons, status bar, and navigation bar.
* Modify the text appearance and UI element shapes across the app to match your design preferences.

**Apply the Theme to the App**

* Set the customized theme globally for the app or activity by referencing the theme in the app's manifest file.
* Ensure consistency across the app by applying the chosen theme to all activities and components.

**Material Design to build modern and intuitive user interfaces, using colors, shapes and typography**

1.Color Palette

2. Shapes and Shadows

3. Typography

4. Layout and Spacing

5. Icons and Imagery

6. Navigation

7. Feedback and Interactions

**CHAPTER – 4**

**Navigation and app architecture**

**1. Activities and Their Lifecycles**

* **Activity:** An activity in Android represents a single screen in an app, like a window where the UI is drawn. It’s the entry point for user interaction. Activities manage the app’s UI and respond to user input.
* **Lifecycle:** The lifecycle of an activity includes multiple states such as onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy(). These methods handle transitions between states (like when the user opens the activity, navigates away, or the app is closed). Understanding activity lifecycles helps manage app resources, save state, and handle interruptions (like incoming calls).

**2. Modern Android Architecture**

* **Android Jetpack Architecture:** Modern Android apps are built using Android Jetpack components, which promote a modular and robust architecture. Key components include:
  + **ViewModel:** Manages UI-related data and survives configuration changes.
  + **LiveData/StateFlow:** Observes changes in UI data**.**
  + **Room:** Manages local databases.
  + **Repository:** A clean abstraction for data sources**.**
  + **Navigation Component:** Manages in-app navigation.
* **MVVM (Model-View-ViewModel):** A widely used design pattern in Android architecture. The ViewModel separates UI logic from business logic, making the code easier to maintain and test.

**3. Use StateFlow and UDF (Unidirectional Data Flow) Pattern**

* **StateFlow:** A Kotlin API for managing and observing state in a reactive way, much like LiveData. It emits values and helps manage UI state changes efficiently.
* **UDF (Unidirectional Data Flow):** A pattern where the state of the app flows in a single direction: from the source of truth (ViewModel) to the UI. The UI sends events (like user input) to the ViewModel, which updates the state, and the UI then reacts to those changes. This pattern is useful for maintaining consistency in complex UIs.

**4. Add a ViewModel to Save Data and State**

* **ViewModel**: A lifecycle-aware component that stores and manages UI-related data. It survives configuration changes, such as screen rotations, ensuring that data is not lost. It decouples the data handling logic from the activity or fragment, leading to more maintainable and testable code.

**5. Set up and Use the Navigation Component with Compose**

* **Navigation Component**: Android’s recommended way to handle navigation in an app. It simplifies fragment transactions, passing data between screens, and managing the back stack. In Jetpack Compose, the navigation component integrates smoothly, allowing declarative navigation based on composables.
* **Benefits:** The Navigation component supports deep linking, nested navigation graphs, and safe args for type-safe data passing between destinations.

**6. Understand What Responsive UI Is**

* **Responsive UI:** Designing an interface that adapts to different screen sizes, orientations, and devices (phones, tablets, desktops). It ensures the app looks and works well on any screen, from small mobile devices to large tablets or foldable phones. Responsive design involves using flexible layouts, percentage-based dimensions, and adaptive content scaling.

**7. Use Window Class Sizes to Build for Different Screen Sizes**

* **Window Class Sizes:** Android classifies screen sizes into categories (compact, medium, and expanded) based on their dimensions. These classes help design responsive UIs that adapt based on the available screen space. Developers can provide different layouts or behaviors for different window sizes, ensuring the app scales and works across a wide variety of devices.

**8. Add a Navigation Drawer to an App**

* **Navigation Drawer:** A side panel that slides in from the edge of the screen to display navigation options. It’s a common UI pattern for accessing different parts of the app, often used to house menu items like "Home," "Settings," or other major sections. The navigation drawer provides a consistent, easily accessible method of app navigation, especially when paired with Android’s Navigation component**.**

**CHAPTER – 5**

**Connect to the Internet**

**Basics of Concurrency and Coroutines in Android**

Concurrency in Android refers to the ability of the system to perform multiple tasks at the same time. Given the nature of mobile applications, managing tasks efficiently (like networking, database operations, or heavy computation) without blocking the main thread is critical to ensure smooth user experiences.

Coroutines are a feature of Kotlin that provide a more concise and readable way to handle asynchronous operations. Coroutines simplify concurrency by allowing code to be written sequentially while managing background tasks efficiently. With coroutines, you can manage threads, switch contexts, and write non-blocking code easily using suspending functions. In Android, coroutines integrate seamlessly with popular libraries and architectural patterns to improve performance and simplify code.

**2. Data Layer in Modern Android App Architecture**

The data layer is responsible for managing the app’s data sources, such as local databases (e.g., Room), remote APIs (e.g., Retrofit), or other sources like shared preferences. It serves as the single source of truth for the app's data and is often abstracted to ensure separation of concerns, making it easier to test and maintain.

In modern Android architecture (e.g., using the MVVM pattern), the data layer usually consists of repositories that retrieve data from multiple sources and data models that represent the business logic. This helps ensure that the app can easily handle data fetching, caching, and synchronization in a structured and scalable way.

**3. Implementing a Repository to Centralize Data Access**

A repository is a design pattern used in Android to centralize data access logic, making it easier to manage different data sources (such as local databases, network requests, or even in-memory caches). By abstracting the data source behind a repository, the app’s architecture becomes more modular and easier to test, as components don’t need to worry about where the data comes from.

Repositories are commonly used in conjunction with ViewModels in MVVM architecture, where the ViewModel interacts with the repository to fetch data, which is then exposed to the UI. This setup promotes clean code by decoupling the data-fetching logic from the UI logic.

**4. Using Retrofit to Retrieve Data from a Remote Server**

Retrofit is a popular HTTP client library used in Android to simplify making network requests to RESTful APIs. Retrofit abstracts much of the boilerplate code associated with making network calls, parsing JSON responses, and managing asynchronous tasks. It allows you to define API endpoints using simple interface methods, making the code clean and easy to maintain.

By using annotations, Retrofit maps API requests and responses to Java or Kotlin objects, streamlining network interactions and error handling. Additionally, Retrofit can work seamlessly with other libraries like Gson for JSON parsing, and it integrates well with coroutines for asynchronous network calls.

**5. Loading and Displaying Images Using the Coil Library**

Coil (Coroutine Image Loader) is an Android library designed for loading images into views (such as ImageView) in a fast, efficient manner. Built using Kotlin and coroutines, Coil provides a lightweight solution for handling image loading with minimal memory overhead, compared to other libraries like Glide or Picasso.

Coil simplifies the process of loading images from the internet, caching them, and managing memory efficiently. It supports various image formats and allows easy integration with your app’s UI. Additionally, Coil is highly customizable, supporting transformations, placeholders, error handling, and more.

**6. Implementing Dependency Injection in Android**

Dependency injection (DI) is a design pattern that promotes loose coupling between classes by providing a way to supply their dependencies externally, rather than letting classes create their own dependencies. In Android, DI is typically used to manage dependencies such as repositories, network clients, or services that various parts of the app require.

By implementing DI, your app becomes more modular, easier to test, and more maintainable. Libraries like Dagger and Hilt (a simplified version of Dagger for Android) are commonly used for dependency injection. DI makes it possible to inject dependencies into Android components (e.g., ViewModels, Activities, and Fragments) without having to manually instantiate or manage those dependencies, promoting cleaner and more testable code.

**CHAPTER – 6**

**Data Persistence**

**Basics of SQL for SQLite: Insert, Update, and Delete**

**SQL (Structured Query Language)** is a standard language used to communicate with databases, and SQLite is a popular lightweight database that is commonly used in mobile apps and embedded systems. Here’s a description of the basic SQL operations:

* **INSERT**: This command is used to add new records to a table. You specify the table name, the columns you want to insert values into, and the corresponding values.
  + Example: You insert a new user into a Users table with INSERT INTO Users (name, age) VALUES ('John', 25).
* **UPDATE**: This command updates existing records in a table. You need to specify the table, the column to be updated, the new value, and a condition to identify which records to update.
  + Example: Updating a user’s age in the Users table could look like UPDATE Users SET age = 26 WHERE name = 'John'.
* **DELETE**: This command removes records from a table. You need to specify the table and a condition to identify which records to delete.
  + Example: To remove a user from the Users table, you would use DELETE FROM Users WHERE name = 'John'.

**2. Using the Room Library to Add a Database to an Android App**

The **Room library** is a part of Android’s Jetpack architecture components, and it provides an abstraction layer over SQLite, allowing developers to manage databases more easily while taking care of much of the boilerplate code. Room helps developers to store, access, and manage data in local databases with the following components:

* **Entities**: These represent tables in the database. Each entity is a class that is annotated with @Entity. The fields in the class represent the columns in the table.
* **DAO (Data Access Object)**: This is an interface where you define methods for interacting with the database, such as inserting, updating, deleting, and querying data. The methods are annotated with SQL queries like @Insert, @Update, @Delete, and @Query.
* **Database**: The @Database annotation marks the class as a Room database and defines which entities the database contains. Room automatically handles the database creation and version management.

Room simplifies handling database operations, provides compile-time checks for SQL queries, and integrates seamlessly with LiveData and Kotlin Coroutines for observing and performing database operations asynchronously.

**3. Using Database Inspector to Test and Debug Database Issues**

**Database Inspector** is a powerful tool available in Android Studio that allows developers to inspect and interact with SQLite databases in real time while the app is running. Here’s how it helps:

* **Viewing Database Structure**: Developers can view the database structure, including tables, columns, and the types of data stored.
* **Query Execution**: You can run SQL queries directly in the Database Inspector to test or fetch data without modifying the app code. This is helpful for testing complex queries or debugging issues.
* **Real-time Updates**: Database Inspector shows real-time updates as the app runs. If the app inserts, updates, or deletes data, you can immediately see the effect in the database.
* **Viewing Data**: The tool allows you to view the data stored in each table and filter or sort it for analysis. You can also inspect data to ensure it's being saved or updated correctly.

This tool helps developers identify database issues early and easily, reducing debugging time and improving development efficiency.

**4. Using Preference DataStore to Store User Preferences**

**Preference DataStore** is a newer and more efficient way to store key-value pairs or user preferences in Android, replacing the traditional SharedPreferences API. There are two types of DataStore implementations:

* **Preferences DataStore**: This is used for storing simple key-value pairs, similar to SharedPreferences. It is ideal for user settings like theme preferences or notification settings.
* **Proto DataStore**: This is used for storing more complex data using Protocol Buffers (a language-neutral, platform-neutral, extensible mechanism for serializing structured data).

Some of the key features of **Preference DataStore** include:

* **Asynchronous and Safe**: It works asynchronously using Kotlin Coroutines, meaning it avoids blocking the main thread. This makes it more efficient than SharedPreferences.
* **Consistency**: It guarantees data consistency with transactional data changes, ensuring no data corruption.
* **Lifecycle Awareness**: You can use it with Jetpack components like LiveData or Flow, making it easier to observe and update UI based on preference changes.

Preference DataStore is generally used for storing user-specific preferences and is ideal for apps that need to store and manage simple user settings across app sessions.

These topics cover fundamental concepts and tools for managing data persistence and local databases in Android apps, providing you with the necessary skills to work with SQLite, Room, and user preferences.

**CHAPTER – 7**

**Work Manager**

**Defining Long-Running Tasks in WorkManager**

In Android development, **WorkManager** is the recommended solution for scheduling long-running background tasks that are guaranteed to execute, even if the app is closed or the device is rebooted. It is specifically designed for tasks that should be deferrable (i.e., tasks that don't need to execute immediately) but must be guaranteed to finish eventually. WorkManager handles all the constraints, retries, and system optimizations needed to ensure that long-running tasks are executed reliably.

**Adding WorkManager to an Android App**

To integrate WorkManager into an Android app, developers need to add the required dependencies to the app's build configuration. Once configured, WorkManager provides APIs to schedule and manage background work in a systematic way. It abstracts away direct handling of background threads, alarms, and services, offering a simplified interface for managing background operations that need to be reliably completed.

**Creating a Worker Object and Enqueueing Work**

A **Worker** class is responsible for executing the background task. When creating a Worker object, the developer specifies the task that should run in the background. The doWork() method in the Worker class contains the logic of the task. Once the Worker is defined, a **WorkRequest** is created and then **enqueued** using the WorkManager instance. This ensures the task is added to the queue for execution based on the specified criteria, such as constraints or timing.

**Creating Constraints on WorkRequests**

Constraints are conditions that must be met before a WorkRequest is executed. WorkManager allows developers to specify constraints, such as requiring a network connection, charging state, or idle state, before the task is allowed to run. By defining these conditions, WorkManager ensures that tasks only execute under suitable circumstances, improving resource management and battery efficiency on the device.

**Using the Background Task Inspector for WorkManager**

Android Studio provides a **Background Task Inspector**, a tool that enables developers to monitor, inspect, and debug background tasks handled by WorkManager. This tool provides insights into scheduled tasks, their status, execution details, and any potential issues. It helps in understanding the behavior of background work and can be valuable for troubleshooting and optimizing long-running background tasks.

**CHAPTER – 8**

**View and Compose**

**1. View-based UI Toolkit and Building App UI using XML:**

The View-based UI toolkit is the traditional approach to creating user interfaces in Android. It involves defining the app's UI components (such as buttons, text fields, etc.) using XML layouts. The XML files describe how the interface should appear, and the corresponding Java or Kotlin code handles the functionality. The Android framework then translates the XML layout into actual UI elements that users interact with. Each UI component is a subclass of View (e.g., TextView, Button, etc.), and these views are laid out in a hierarchy, typically using containers like LinearLayout, RelativeLayout, or ConstraintLayout.

XML-based UIs offer flexibility and separation of concerns, as the UI structure is defined separately from the logic. However, this approach can sometimes result in verbose code and performance limitations, especially when dealing with complex, nested layouts.

**2. Adding a Composable in an App Built with Views:**

Jetpack Compose is Android's modern UI toolkit that simplifies UI development by using a declarative approach. Instead of building UIs using XML, Compose allows you to define UI components (called "composables") directly in Kotlin code.

In a project that primarily uses Views (XML-based), it's possible to integrate composables without completely rewriting the app. This hybrid approach allows gradual migration to Jetpack Compose. For example, you can add a composable inside a Fragment or Activity by using a ComposeView, which serves as a bridge between the traditional View system and Compose. This is useful when you want to leverage the benefits of Compose, such as its reactive nature, without overhauling the entire app.

**3. Adding the Navigation Component to the App and Using it to Navigate Between Fragments:**

The **Navigation component** is part of Android Jetpack and simplifies navigation within an app. It manages fragment transactions, deep links, and complex navigation patterns (e.g., bottom navigation, tab navigation) in a more structured way. It uses a navigation graph (defined in XML or Kotlin) to declare the app's navigable destinations (fragments, activities) and the actions that connect them.

By integrating the Navigation component, developers no longer need to manually manage the FragmentTransaction or keep track of the back stack. Instead, navigation is handled by the framework, ensuring smoother transitions and state management. This reduces boilerplate code, improves scalability, and allows the app to manage navigation even across processes and after configuration changes.

**4. Using AndroidView to Display Views:**

In Jetpack Compose, when there's a need to reuse existing view-based components (from the XML approach), you can use the AndroidView composable. This allows you to display traditional views (such as a MapView, WebView, or any custom view) inside a Compose UI.

AndroidView bridges the gap between the imperative UI system (Views) and the declarative UI system (Compose). It provides a way to incorporate legacy UI components or third-party view libraries that haven't been converted to Compose yet. By wrapping these Views inside AndroidView, developers can ensure a seamless transition to Compose without losing compatibility with existing components.

**5. Adding Existing View-based UI Components in a Compose App:**

When building a fully Compose-based app, you may still need to incorporate existing UI components from the View system. This could be because of legacy code or components that are not yet available in Compose. To achieve this, Compose offers a way to integrate existing views through AndroidView. This allows developers to reuse custom views, libraries, or view-based UI components within a Compose hierarchy.

Adding View-based components in a Compose app helps in gradually migrating to Compose without a full rewrite. It provides flexibility in combining both UI paradigms (Views and Compose) and ensures that legacy codebases can still benefit from Compose features while maintaining backwards compatibility with older view components.

ACTIVITY LOG

|  |  |  |  |
| --- | --- | --- | --- |
| Day | Brief Description of Daily Activity | Learning outcomes | Person in-Charge  Signature |
| Day-1,2 | Introduction to Android Development concepts. | Gaining knowledge on fundamentals. |  |
| Day-3,4 | Installing and Exploring android studio and performing basic tasks. | How to perform particular task. |  |
| Day-5,6 | Gone through simple Kotlin programs whish display text output and executing the Kotlin program to in android app emulator or real device. | The elements which required in executing Kotlin programs. |  |
| Day-7,8 | Studied about conditionals, function types, and lambda expressions in Kotlin. | Getting idea on all types of expressions. |  |
| Day-9,10 | Creating an Android app with a simple UI to display text and images. | Created a simple user interface. |  |
| Day-11,12 | Adding buttons to the UI and respond to user interactions. | Learned programming to create buttons. |  |
| Day-13,14 | Building and testing a simple Android app with interactive UI. | Tested the user interface. |  |

WEEKLY REPORT

Week 1 & 2

Objectives:

* Gained fundamental knowledge on Android development and its concepts.
* Creating and installing android studio.

Detailed Report:

During the first week, the focus was on setting up the development environment and gaining familiarity with Kotlin and Android Studio. Android Studio was downloaded and successfully installed, providing access to a powerful IDE for Android development. After installation, time was spent exploring the various features and layout of Android Studio, including its interface, project structure, and tools. A basic "Hello World" app was created as the first Android application, which helped in understanding the process of building and running apps. Kotlin was introduced as the primary programming language for Android development, and simple programs were written using println() to display text output on the console. Additionally, Log messages were generated using Log.d() for debugging purposes. The Kotlin programs were run on both an emulator and a real device, verifying their functionality across different environments. By the end of the week, familiarity with Kotlin syntax was achieved, along with basic competence in using Android Studio to create and test applications. This foundational knowledge serves as a critical step toward more advanced Android app development.

In the second week, the focus shifted to building a basic understanding of Kotlin's conditionals, function types, and lambda expressions, and applying this knowledge to develop a simple Android app with an interactive UI. Conditionals such as if, when, and else were explored, enabling decision-making in Kotlin programs. Function types and lambda expressions were introduced to streamline code, especially for handling events and callbacks efficiently. Moving on to UI development, an Android app was created with a layout that included both text and images, designed using XML. Buttons were added to the UI, which required learning about event listeners and handling user interactions. Tap events were captured using Kotlin, with actions defined using lambda expressions for concise event handling. The app was thoroughly tested on an emulator and a physical device to ensure proper functionality. This process solidified understanding of building user interfaces and integrating interactivity into apps. By the end of the week, a fully functional Android app with a simple yet responsive UI was successfully built, marking a significant step forward in Android development skills.

ACTIVITY LOG

|  |  |  |  |
| --- | --- | --- | --- |
| Day | Brief Description of Daily Activity | Learning outcomes | Person in-Charge  Signature |
| Day-1,2 | Implementing Kotlin classes and data collections. | Getting idea on various data collections. |  |
| Day-3,4 | Adding UI components like EditText to allow users to enter data. | The process of giving data. |  |
| Day-5,6 | Using state management to update the UI based on user input. | Giving input and communicating with user interface. |  |
| Day-7,8 | Learning how to use data classes and collections to build a scrollable list in the app. | Learnt about classes, collection and list. |  |
| Day-9,10 | Adding click listeners to list items for user interaction. | Making UI easy for client access. |  |
| Day-11,12 | Implementing an app bar and customize the app theme using Material Design guidelines. | Customizing the application. |  |
| Day-13,14 | Created a dynamic list-based app with a modern UI using Material Design. | Developed a proper user interface. |  |

WEEKLY REPORT

Week 3 & 4:

Objective:

* Creating an Android app with a simple UI to display text and images with adding buttons to the user interface and make it to respond to reactions.
* Making it in a way to build and testing a simple Android app with interactive UI.

Detailed Report:

During third Week , the focus was on implementing Kotlin classes and managing user input in Android apps. First, Kotlin classes were studied in depth, with attention to object-oriented principles and the use of data collections such as List and Map to store and manage information. The Android app was enhanced with UI components like EditText, which allowed users to enter text data. The concept of state management was introduced, enabling dynamic updates to the UI based on the user’s input. For instance, when a user entered their name in an EditText, the app would display the name in real time using TextView. This required managing the app’s state effectively to reflect user-driven changes without restarting the activity. The implementation of state-aware UI components using Kotlin made the app more interactive and personalized. By the end of the week, the app could successfully take user input, process it, and update the display dynamically, improving user experience. This week’s activities solidified skills in state management, laying the groundwork for more complex user interaction patterns.

In fourth week I have started concepts in advanced UI concepts, focusing on building a scrollable list and integrating Material Design guidelines for a modern app look. The week began by learning how to use Kotlin data classes and collections to organize and present data efficiently. These were applied to create a scrollable list using RecyclerView, enabling the app to display a dynamic list of items. Each list item was made interactive by adding click listeners, allowing users to interact with individual items and receive immediate feedback. Moving forward, an app bar (using Toolbar) was implemented to enhance the app's structure and navigation. Material Design principles were studied and applied, particularly for customizing the app's theme, colors, and typography, which provided a more polished, consistent look. The app was enhanced with a cohesive and modern UI, using components like floating action buttons, styled buttons, and a refined color scheme. By the end of the week, a list-based app was created, complete with responsive user interaction and a modern UI that adhered to Material Design standards. This week marked a significant improvement in both design aesthetics and user functionality.

ACTIVITY LOG

|  |  |  |  |
| --- | --- | --- | --- |
| Day | Brief Description of Daily Activity | Learning outcomes | Person in-Charge  Signature |
| Day-1,2 | Learning about Android activities and lifecycle methods. | Lifecycle of android studio. |  |
| Day-3,4 | Adding ViewModel to save and manage UI-related data. | Saving the data in android studio. |  |
| Day-5,6 | Setting up the Navigation component using Jetpack Compose for screen navigation. | Understood the screen navigation principle. |  |
| Day-7,8 | Understanding and implementing responsive UI for different screen sizes using window classes. | Implemented responsive user interface. |  |
| Day-9,10 | Learning concurrency basics in Android and Implementing Retrofit to retrieve data from a remote API. | Connecting user interface to API. |  |
| Day-11,12 | Display fetched images using the Coil library. | Using coil library. |  |
| Day-13,14 | Understanding and applying dependency injection for better app architecture. | Making  Application architecture. |  |

WEEKLY REPORT

WEEK 5&6

Objectives:

* Implemented a multi-screen app with structured navigation and responsive design.
* Built an app that retrieves and displays data from an external source.

Detailed Report:

In Fifth Week, the focus was on learning Android activities, lifecycle methods, and structuring an app with proper navigation. The first task was understanding the lifecycle of an activity and how lifecycle methods like onCreate(), onStart(), and onPause() work, ensuring that the app behaves consistently across different states. The ViewModel architecture component was introduced to save and manage UI-related data across configuration changes, avoiding unnecessary recomputations. Next, the Jetpack Navigation component was set up, using Jetpack Compose to handle seamless screen transitions and structured navigation between different activities or fragments. Emphasis was also placed on making the app responsive to different screen sizes. Android window classes, such as compact, medium, and expanded, were explored to create a responsive layout, adjusting UI elements dynamically based on device screen size. By the end of the week, a multi-screen app with structured navigation and responsive design was built, ensuring a smooth user experience across different devices. The implementation of ViewModel and navigation components helped enhance app performance and maintainability.

In Sixth I have shifted my focus to network operations, introducing concurrency in Android and retrieving data from external sources. Concurrency was explored using coroutines, a key feature of Kotlin that simplifies asynchronous programming and allows for non-blocking operations. The next step was integrating Retrofit, a popular HTTP client, to fetch data from a remote API. Retrofit was used to parse JSON data and retrieve information such as images and text, which could be displayed within the app. The Coil image-loading library was implemented to display fetched images efficiently, taking advantage of caching and asynchronous image loading. Alongside network operations, dependency injection (DI) was introduced, focusing on using Dagger/Hilt to inject dependencies like Retrofit instances and ViewModels, enhancing app architecture. This made the codebase cleaner, more modular, and easier to test. By the end of the week, a functional app that fetched data from an external source and displayed it was built. The integration of coroutines, Retrofit, and Coil, along with dependency injection, provided a robust architecture for handling network operations in Android apps.

ACTIVITY LOG

|  |  |  |  |
| --- | --- | --- | --- |
| Day | Brief Description of Daily Activity | Learning outcomes | Person in-Charge  Signature |
| Day-1,2 | Learning the basics of SQL for database operations. | Required SQL knowledge for application. |  |
| Day-3,4 | Using the Room library to create a local database for the Android app. | How to create local database. |  |
| Day-5,6 | Adding functionality to insert, update, and delete data and  Use Database Inspector to debug and test database-related issues. | Applying required commands for database. |  |
| Day-7,8 | Studying how to implement background tasks in Android using Work Manager. | Concepts on work manager in doing tasks. |  |
| Day-9,10 | Creating and enqueue a Worker object to handle background tasks. | How to handle background tasks. |  |
| Day-11,12 | Setting constraints on WorkRequests (e.g., network conditions). | What are networking conditions and its types. |  |
| Day-13,14 | Using Background Task Inspector to debug WorkManager. | Debugging and avoding errors in database. |  |

WEEKLY REPORT

Week 7&8

Objectives:

* Developed an app that handles background tasks efficiently using WorkManager.
* Successfully integrated data persistence into the app using SQLite and Room.

Detailed Report:

During seventh week my focus was on data persistence in Android apps, specifically using the Room library for database operations. The week started with an introduction to SQL, covering essential database concepts such as creating tables, inserting, updating, and deleting data using structured queries. These SQL basics were important for understanding how data is managed under the hood when using Room, which is an abstraction layer over SQLite. Next, the Room library was integrated into the Android app to create a local database. Entities, DAOs (Data Access Objects), and the database itself were set up to store and manage app data. CRUD (Create, Read, Update, Delete) operations were implemented, allowing users to add, modify, and remove data from the local database through the app’s UI. The Database Inspector tool in Android Studio was used to view the database’s content, inspect SQL queries, and debug database-related issues in real time. By the end of the week, data persistence was successfully integrated into the app, ensuring that user data is stored locally and remains available even when the app is closed. This solidified the understanding of database management within Android apps.

In week eighth I have focused on implementing background tasks in Android using the WorkManager API, which provides a flexible and reliable way to schedule background tasks. The week began by studying the basics of background work in Android and the advantages of using WorkManager, such as its ability to handle tasks that must be guaranteed to run, even after app restarts or device reboots. A Worker class was created and enqueued to perform background tasks such as data syncing or cleanup processes. The WorkRequest object was configured with specific constraints, like network availability or battery status, to ensure that tasks were executed only under suitable conditions. For debugging, the Background Task Inspector tool in Android Studio was employed to monitor and troubleshoot background tasks during runtime. This ensured that tasks were being enqueued and executed as expected. By the end of the week, the app was capable of handling background tasks efficiently using WorkManager, making it more robust and reliable. This allowed the app to perform long-running operations without affecting the user experience, ensuring seamless performance.

ACTIVITY LOG

|  |  |  |  |
| --- | --- | --- | --- |
| Day | Brief Description of Daily Activity | Learning outcomes | Person in-Charge  Signature |
| Day-1,2 | Exploring the traditional View-based UI toolkit, building app UI with XML. | Creating app using xml to UI. |  |
| Day-3,4 | Using Compose in combination with existing Views. | Creating UI with various combination. |  |
| Day-5,6 | Adding the Navigation component to an app with multiple fragments. | Creating an app with multiple required components. |  |
| Day-7,8 | Integrating existed View-based UI components into a Compose-based app. | Developed app with composed base application. |  |
| Day-9,10 | Developing a small final project integrating various components. | Started small project. |  |
| Day-11,12 | Review and test all features in the project. | Working out with all applications with implementing all tasks. |  |
| Day-13,14 | Receive feedback and improvements based on code review. | Completing the app and improvising it. |  |

WEEKLY REPORT

WEEK 9 & 10

Objectives:

* Built a hybrid app combining View-based UI and Jetpack Compose.
* Completed a comprehensive Android app integrating all the learned concepts and tools.

Detailed Report:

During ninth week I focused on understanding both the traditional View-based UI system in Android and the newer Jetpack Compose framework. The week began with an exploration of the View-based UI toolkit, where XML layouts were used to design the user interface. Concepts like ConstraintLayout, LinearLayout, and RecyclerView were revisited, helping to build a better understanding of traditional Android UI components. The transition to Jetpack Compose, a modern, declarative UI framework, was introduced, with emphasis on how it simplifies UI development by using Kotlin code instead of XML. A key activity involved integrating Jetpack Compose with existing View-based components, allowing for a hybrid app that leveraged both approaches. The Jetpack Navigation component was also used to enable seamless navigation between multiple fragments or screens, combining Compose and Views. Additionally, traditional Views were integrated into a Compose-based app, showcasing how the two UI systems can coexist. By the end of the week, a hybrid app combining both View-based UI and Jetpack Compose was successfully built, demonstrating the flexibility of using both frameworks in Android development.

Tenth week was dedicated to the final project, which involved integrating all the concepts and tools learned throughout the course. The goal was to develop a comprehensive Android app that utilized state management, networking, data persistence, background tasks, and both View-based and Compose-based UI components. The project required combining key libraries and frameworks like Room for data persistence, Retrofit for networking, WorkManager for background tasks, and Jetpack Compose for building a modern UI. Throughout the week, time was spent reviewing and testing each feature in the project to ensure that they worked seamlessly together. Debugging tools such as the Database Inspector, WorkManager Inspector, and network monitoring tools were used to troubleshoot and optimize performance. After development, a code review session was held to receive feedback on the project’s architecture, design patterns, and code quality. Based on the feedback, improvements were made to streamline the code and enhance performance. By the end of the week, the final project was completed, integrating all learned components into a well-functioning, cohesive Android app, marking a successful conclusion to the course.

CERTIFICATE

