STUDYBUDDY

Merve Karacaoğlu

Mobile Programming Project

WHAT IS STUDYBUDDY?

StudyBuddy is a mobile application designed to support students in managing their studies by allowing them to:

- Log study sessions
- Edit/delete study sessions
- Track progress with analytics and streaks
- Set weekly goals
- Organize tasks with subjects and tags
- Set personalized reminders
- Set Pomodoro timer
- Customize the app's theme

FUNCTIONAL REQUIREMENTS

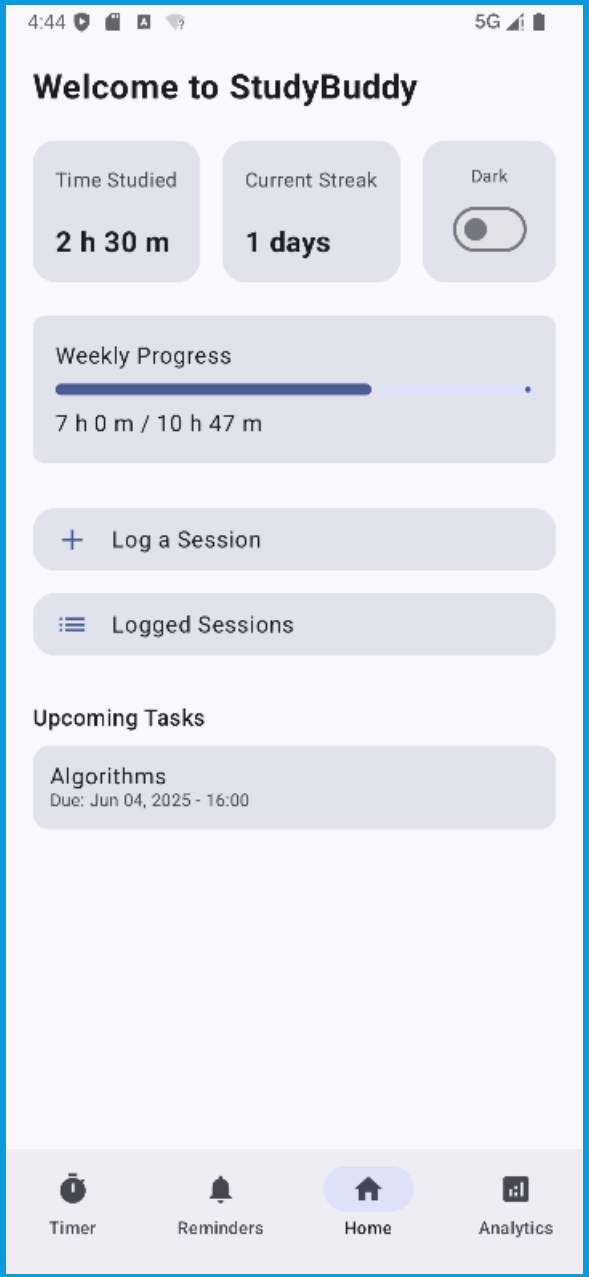
- The user must be able to **log a study session** including subject, duration, notes, completed checkbox, and tags.
- Sessions must be **stored locally** using the Room database and remain accessible offline.
- The app must provide a **dashboard** displaying total study time, streaks, goal progress and two buttons to navigate to log session and logged sessions screen.
- The user can **view a list of logged sessions**, with filters by completion and due date.
- The user can **set reminders** for study sessions or tasks and get a notification at set time.
- The user can set a **Pomodoro timer** for 25 minutes work, 5 minutes break and 15 minutes long break after 4 sessions
- Users must be able to **customize the app theme** and preferences (dark/light mode).
- The app must use **bottom navigation** to switch between Dashboard, Analytics, Reminders, and Timer.

NON-FUNCTIONAL REQUIREMENTS

- The app should function **entirely offline**, with persistent local storage.
- Architecture must follow **MVVM** with clear separation between layers.
- Code should be **modular and maintainable**, supporting future extension.
- All UI strings should be **externalized using string resources** for localization support.
- The interface should be **reliable**, **easy to use and intuitive**.

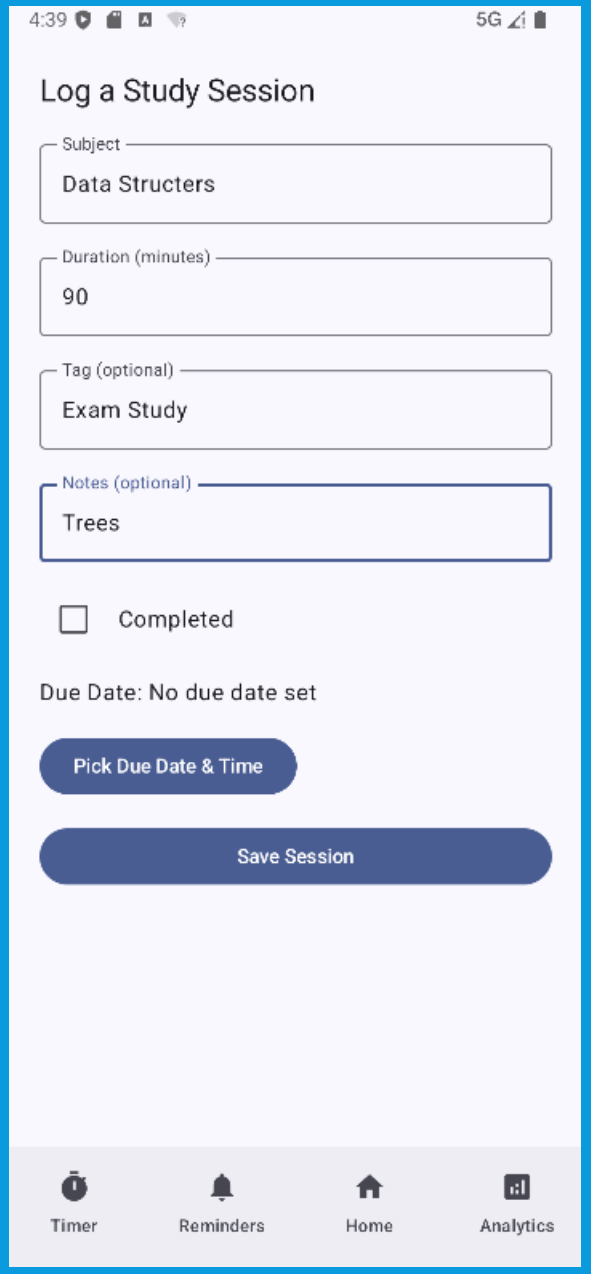
FEATURES :HOME

- Main page of the application
- Shows the time spent studying
- Shows how many days without a break sessions are logged
- Toggle button to switch the theme to dark/light
- Weekly progess bar according the goal set and sessions logged.
- Log a Session button to navigate to the screen to log sessions
- Logged Sessions button to navigate to the screen to view, edit and delete the logged sessions.



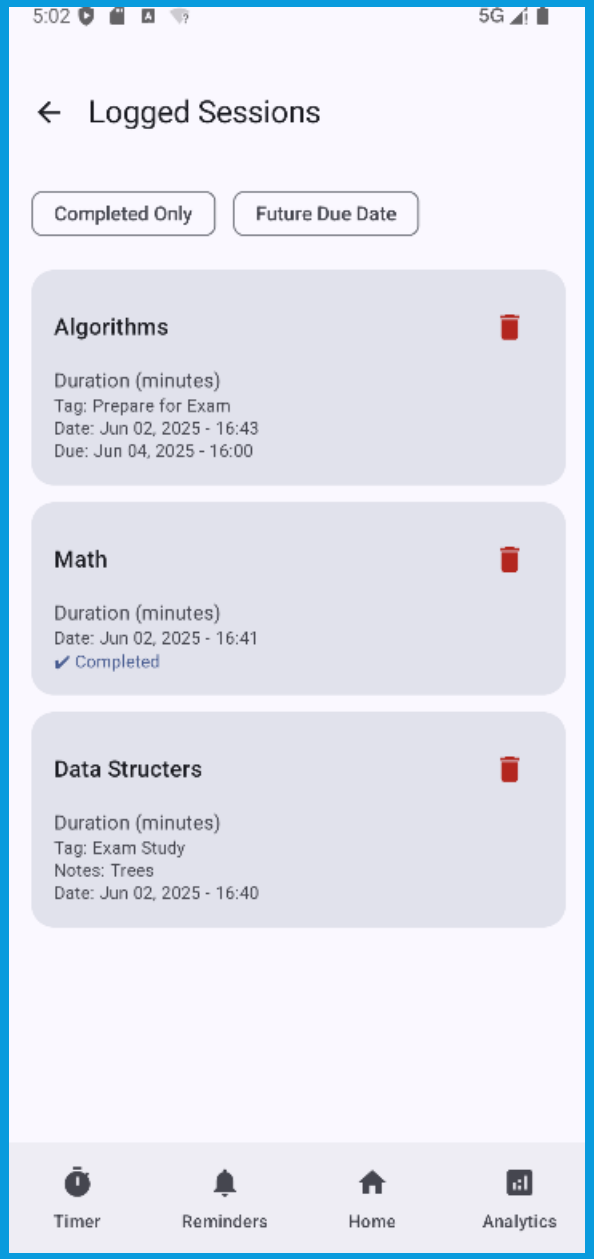
FEATURES :LOG A SESSION

- User can log sessions and customize subject, duration, tags and notes.
- Users can mark the sessions as completed. (The duration will appear on the home page (Time Studied) only if it is mark as completed)
- Users can set a due date for sessions and this will make the task appear on the home page under Upcoming Tasks.
- The user will be navigated back to the home page when clicked on Save Session



FEATURES :LOGGED SESSIONS

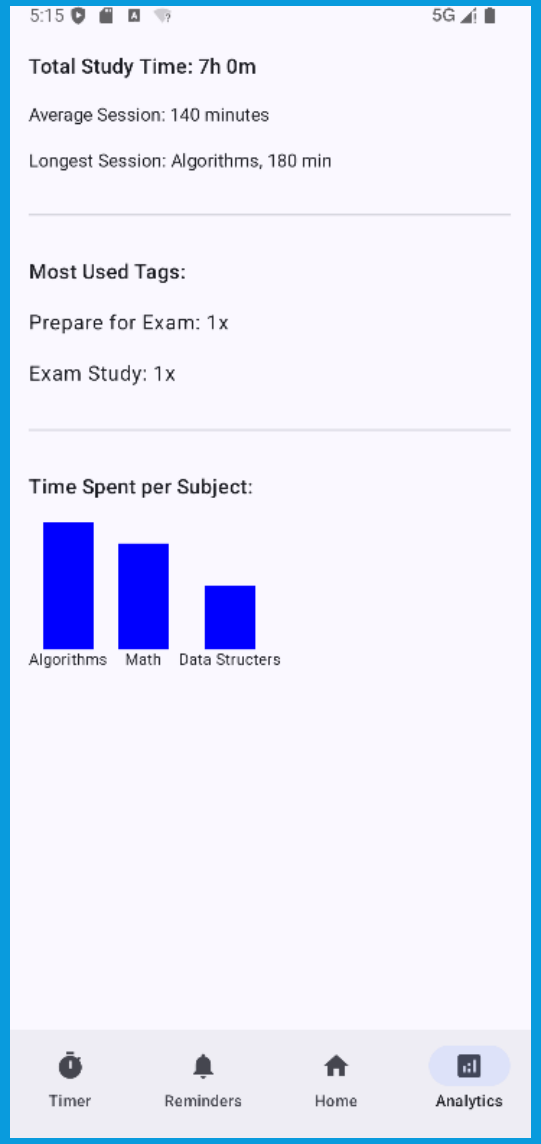
- User can see their Logged Sessions and all the information they logged.
- Users can filter the sessions by due date or if it has been completed.
- Users can edit sessions by clicking on them which will take them to another screen similar to log a session page.
- Users can delete sessions , a snack bar will appear to confirm the delete operation.



FEATURES : ANALYTICS

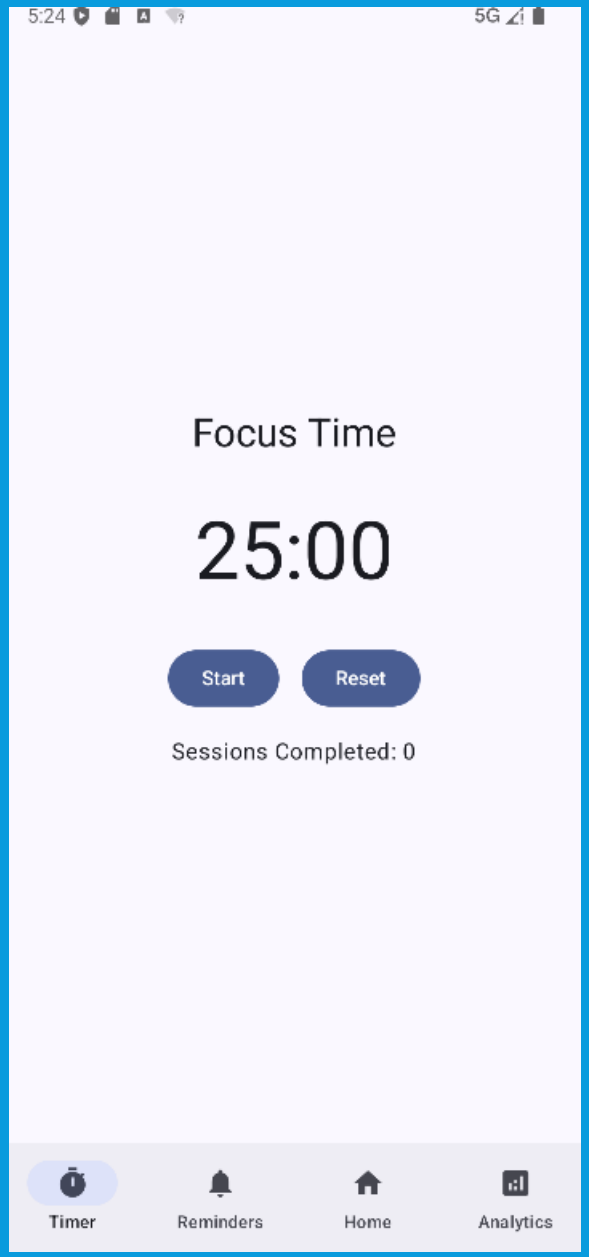
Displays:

- Total study time
- Average session
- Longest Session
- Most Used Tags
- Bar chart to display time spent per subject



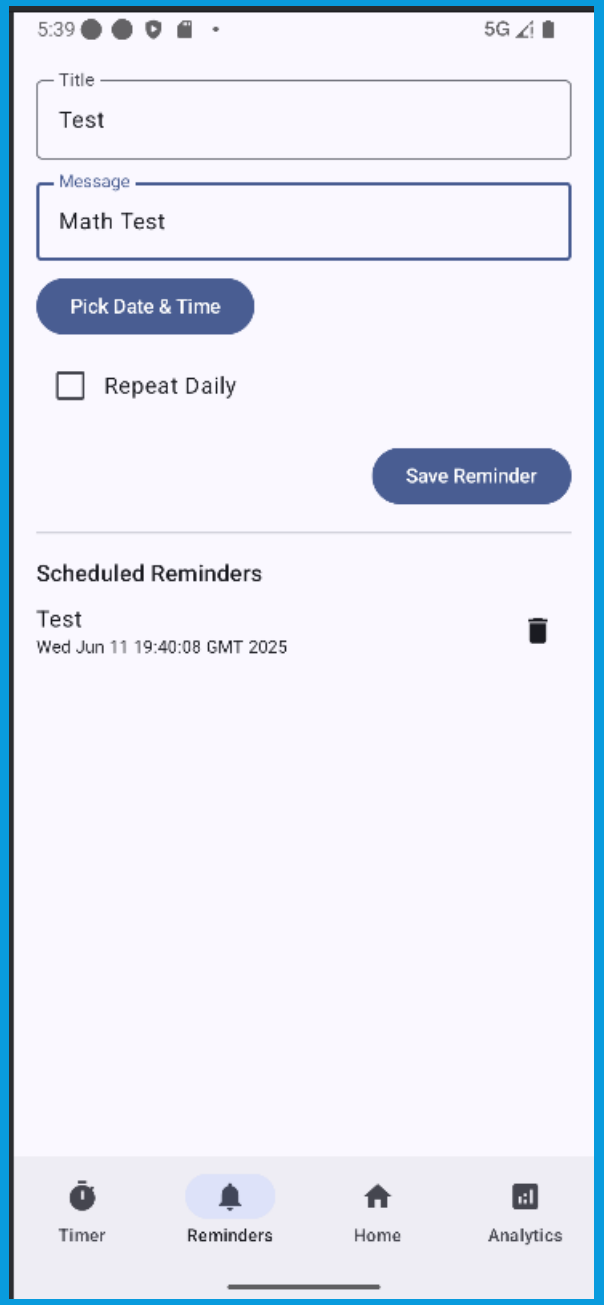
FEATURES : TIMER

- Timer which runs for 25 minutes and displays a notification on the notification panel of the timer.
- User can stop, reset, stop the timer from the notification.
- After 25 minutes the users can start a 5 minute timer for break.
- After 4 x 25 minutes there is a 15 minute break.
- Every 25 minutes, Sessions Completed is getting updated by 1.

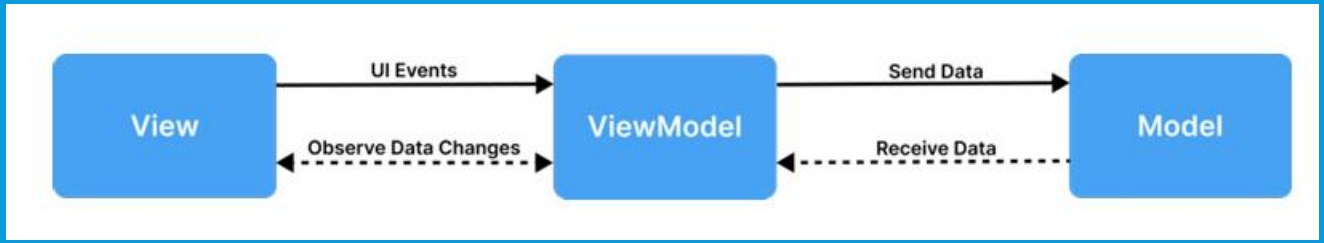


FEATURES :REMINDERS

- Users can set a reminder and customize a title and a message.
- Users can pick a date and time for the reminder and choose if the reminder should be repeated daily.
- The scheduled reminders will be displayed under Scheduled Reminders and can be deleted.

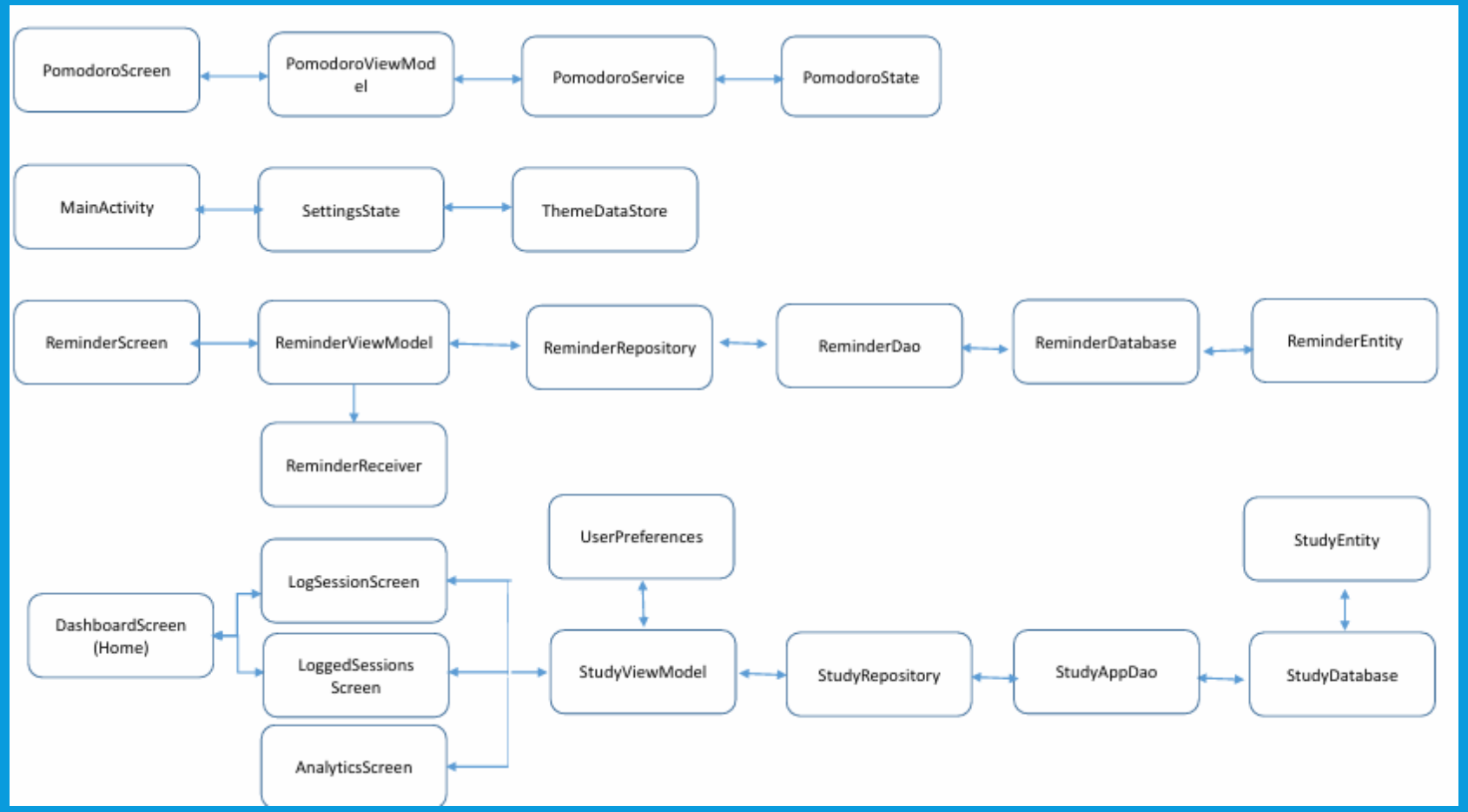


MVVM ARCHITECTURE



- **Model Layer**
  - Consists of data classes (e.g. StudySession entity), the Room database (StudyDatabase), and DAO interfaces (SessionDao).
  - Handles all data operations such as inserting, querying, and updating sessions.
  - Also includes the repository(StudyRepository), which acts as an abstraction layer between ViewModel and DAO.
- **ViewModel Layer**
  - Classes like StudyViewModel expose reactive data (StateFlow) to the UI.
  - They handle business logic such as calculating daily totals, loading upcoming sessions, and managing user preferences.
  - ViewModels survive configuration changes and are lifecycle-aware.
- **View Layer (Compose UI)**
  - Built entirely with **Jetpack Compose** for declarative UI.
  - Screens include DashboardScreen, LogSessionScreen, ReminderScreen, SettingsScreen, and LoggesSessionsScreen, Timer Screen.
  - UI observes ViewModel state using collectAsState() from StateFlow.

ARCHITECTURE



JETPACK COMPONENTS USED

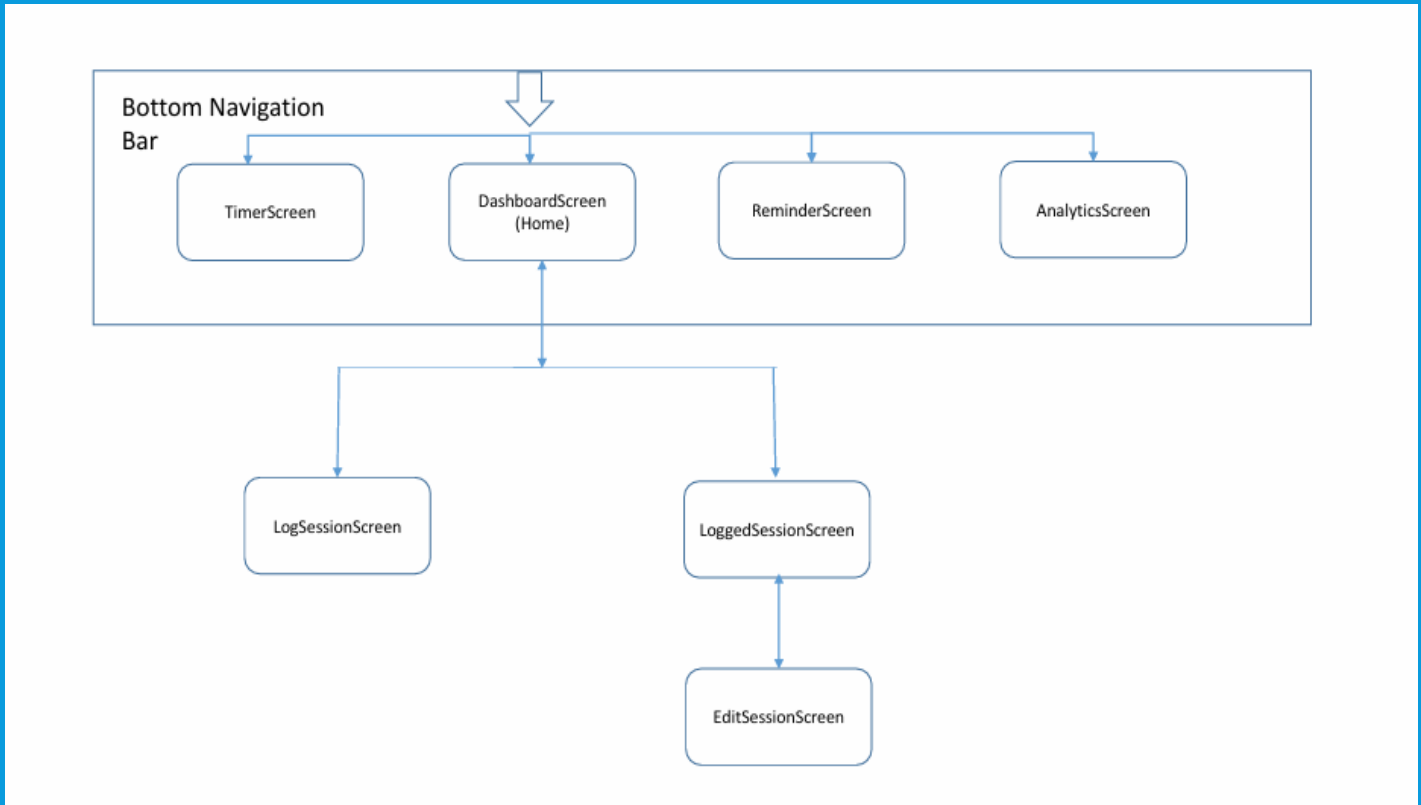
- **Jetpack Compose:** UI layer built entirely with Compose. Components are reactive and modular.
- **ViewModel:** Maintains screen state and encapsulates logic
- **Room:** Handles local data persistence for sessions and reminders.
- **StateFlow:** Used instead of LiveData for observing state in the ViewModel.
- **Navigation:** Jetpack Navigation Component manages screen transitions with NavHost and NavController.
- **DataStore:** Stores persistent settings like dark/light mode and weekly goals settings
- **Coroutines:** Enables background tasks (e.g., database access) without blocking the main thread.
- **AlarmManager:** Schedules notifications for reminders in the background.

JETPACK COMPOSE

- To build a modern and easy to use interface
- All screens are built using Jetpack Compose
- Compose observes StateFlow from ViewModel via collectAsState() for state managment
- UI is seperated into modular compsables for testability and reusability
- Compose works with MaterialTheme
- Each composable screen works with Navigation Compose
- Flexible layout components

JETPACK NAVIGATION COMPOSE FOR SCREEN MANAGMENT

- NavHost to define the navigation graph: specify destinations and how to navigate between them
- NavController to manage navigation: transitions across routes
- Supports nested navigation.
- Ensures a clear seperation of navigation logic from UI.



ROOM

Used to persist study sessions and reminders locally.

Main Components:

- Database: Abstract class that extends RoomDatabase, contains a list of entities and methdos to get the DAOs (StudyDatabase, ReminderDatabase)
- DAO (Data Acess Objects): Interface to Access data, contains methods for queries, inserts, updates and deletions (StudyAppDao, ReminderDao)
- Entity: Represents a database table by defining colums via fields (StudyEntity, ReminderEntity)

Repository: Abstracts database access between the main components and view model.

VIEWMODEL

- Maintains screen state and encapsulates business logic
- Preserves the states shown in the UI during configuration changes
- Reactive and Lifecycle Aware
- Exposes StateFlow to Compose UI
- UI automatically updates with collectAsState()
- Holds immutable (external read-only) and mutable (internal) version of state
- Private functions are called in viewModelScope to launch a coroutine on a background thred
- Fetches data from the Repository and exposes actions

DATASTORE FOR USER PREFERENCES

- Used to save data persistenly across app restarts
- UI updates dynamicaly without manual refreshes and reloading
- Stores preferences like: Dark/Light Theme, Weekly Goal
- Uses suspend functions to save preferences in the background. (non-blocking, asynchronously)
- Preferences are exposed as Flow, allowing the UI to automatically respond to changes.
- ViewModel (weekly goal) or MainActivity (theme) collects preference flows and exposes them to UI using collectAsState()

ALARM MANAGER AND BROADCAST RECEIVER FOR REMINDERS

- User sets a remainder, remainder details are passed to ReminderViewModel
- ViewModel creates an Intent with title/message, wraps intent with PendingIntent so AlarmManager can fire even if the app is not running.
- Schedules alarm with AlarmManager
- AlarmManager triggers the PendingIntent at the right time, sends a broadcast to ReminderReceiver.
- ReminderReceiver receives the Intent, extracts the data and creates a notification and shows it using NotificationManager, NotificationChannel.

FOREGROUND SERVICE FOR TIMER

- Reliable timing with LifecycleService, Notifications and Broadcasts
- User starts the timer. PomodoroViewModel sends an Intent with an action to PomodoroService.
- The foreground service: runs in the background even if the app is closed or screen is off, uses a coroutine loop in serviceScope to count down time every second, posts a persistent notification, send a local broadcast (intent) to update the UI with the current timer state.
- Notification include Pause, Reset and Stop buttons. Each actions sends an intent back to the service to handle accordingly.
- Viewmodel listens to updates using LocalBroadcastManager. Updates UI state based on intent extras.

TESTING

Unit Testing:
ReminderViewModelTest:
To verify that ReminderViewModel correctly loads reminders from the database.
ReminderDao was mocked using mockito-kotlin.
Test frameworks that are used are JUnit4 for assertions and turbine for testing Kotlin Flows.
StudyViewModelTest:
To validate the StudyViewModel logic that interacts with StudyRepository and exposes state to the UI.
StudyRepository was mocked to simulate database operations.
Test frameworks that are u
sed: Junit4, Turbine for collecting flows, Mockito for mocking behavior, Robolectric for Android components without an emulator.

Integration Testing:
StudyIntegrationTest:
The StudyIntegrationTest class verifies the full integration of: Room database (StudyDatabase), Study Repository, StudyViewModel. An **in-memory database** is used to simulate real database interactions without affecting production data.

THANK YOU FOR LISTENING