Assignment 3: Keeping Track App – Exercise Activity Tracker
Name: Dang Khoa Le
Student ID: 103844421
Tutorial: Wednesday 4.30 PM

A. Overview:

The **Exercise Activity Tracker** app is a mobile multi-fragment CRUD (Create, Read, Update, Delete) application developed on Android. This app provides users with a simple interface to manage physical activities, enabling the addition, modification, and deletion of exercises. The app is built with a combination of modern mobile application architecture components such as ViewModels, LiveData, and Room persistence, which ensures effective interaction with stored data. Additionally, the UI features fragments and RecyclerView, designed to enhance user interaction and experience.

B. Planning:

User Stories:

1. As a user, I want to be able to log my daily physical activities with details like activity name, duration, and time, so that I can track my fitness progress.
2. As a user, I want to update or delete the activities I logged previously, so that I can modify incorrect entries or remove unwanted activities from my history.

Use Cases:

Use Case 1: Add an Activity

- **Actors**: User
- **Description**: The user navigates to the "Add Activity" screen, fills out the form with details (e.g., activity name, time, and duration), and submits the form. The system validates the input and adds the activity to the database.
- **Preconditions**:
  - The app is running, and the user is on the main screen.
  - The user has valid inputs for the form fields.
- **Main Flow**:
  1. The user clicks the **Add Activity** button from the main screen.
  2. The app navigates to the **Add Activity** fragment.
  3. The user fills out the activity name, date, and time using input fields.
  4. The system validates the input format for date and time (e.g., time must be in 24-hour format).
  5. If validation passes, the user submits the form.
  6. The system saves the activity in the Room database and updates the **RecyclerView** on the main screen to reflect the newly added activity.
  7. A success message (e.g., **Toast**) confirms that the activity was added successfully.
- **Alternative Flow (Validation Failure)**:
  - If validation fails (e.g., the date or time format is incorrect), the system displays an error message, allowing the user to correct the input.
- **Postconditions**:
  - The new activity is saved and displayed in the main list, which reflects the updated state of the Room database.

Use Case 2: Update an Activity

- **Actors**: User
- **Description**: The user selects an existing activity from the list, edits the details, and submits the updated information. The system validates the updated information and modifies the existing record in the database.
- **Preconditions**:
  - At least one activity exists in the list for the user to update.
  - The app is running, and the user is on the main screen.

- **Main Flow**:
    1. The user selects an activity from the **RecyclerView**.
    2. The app navigates to the **Update Activity** fragment, with the current activity details pre-populated in the form fields.
    3. The user modifies the details (e.g., changes the activity name or updates the duration).
    4. The system validates the updated input fields (e.g., checks for correct date format, duration is a valid number).
    5. If the inputs are valid, the user submits the form.
    6. The system updates the existing record in the database with the new details.
    7. The updated activity replaces the old one in the list, and the **RecyclerView** is refreshed.
    8. A success message (e.g., **Toast**) confirms the successful update.
- **Alternative Flow (Validation Failure)**:
    - If validation fails, the system displays an error message, allowing the user to correct the inputs (e.g., invalid time, name format or missing required fields).
- **Postconditions**:
    - The updated activity is reflected in the main list, and the Room database now contains the modified details.

Time Logs:

- Day 1-3 (average 2 hours a day): Initial planning and researches for Room, DAO, Database, ViewModel, RecylerView and UI design approach.
- Day 3-7 (average 2 hours a day): Designing a basic app. Implement basic CRUD functions allow interaction while data storage with ROOM integration. App contains 3 basic fragments, with Your Activity fragment allowing view and Filter (by date) with the list of RecyclerView exercises, Add Activity fragment bring up a form for new activity with details to be submitted, and Update Activity fragment updating new details matching that specifically exercise id selected to be update.
- Day 8 (4 hours): Redesigning UI elements at Add Activity and Update Activity fragments. Opt-in dark/light mode which can be changed upon the daynight icon on the ToolBar, initialise toolbar_menu.xml for this. Restructure the UI for RecyclerView item from exercise_item.xml. Add color changing and fade-out animation with AlphaAnimation when deleting an exercise item on ExerciseAdapter.kt.
- Day 10-12 (7 hours approximately): Opt in method to filter out exercise RecyclerView items displayed at default is from the future (excluding past events, which can still being viewed from date filtering option). Utilise Toast to feedback user upon adding and updating exercise activities, use Snackbar with UNDO option to notify item deletion with data restoration when change of mind. Add more Logs to enable better debugging.
- Day 13-15 (Average 2 hours a day): Refine the algorithms to push the closest upcoming events on top. Implement checkers to check for valid datetime information, datetime range, and reflect invalid mistakes with Toast. Utilise Coroutines to append a set of feedback message before propagating (launching) Toast feedback messages with 1 second delay. Implement more logs to enhance debugging.
- Day 15-17 (7 hours approximately): Using different approaches (Mockito, Data Binding, scale down Android utility versions to ensure compatibility across usage dependencies etc) for direct UI testing on the targeted fragments but didn't work, hence use a more traditional approach such as "Record Espresso Test". Implement UI Testing Cases for Your Activity, Add Activity and Update Activity fragments, the so called test cases are to test the existance of UI elements and their expected behaviors, ensure fragments navigation, and ensure CRUD actions are as expected. Add app logo icon.
- Day 18-21 (9 hours): Create another app "ExerciseActivityTrackerVariant" to test the app's CPU usage without using LiveData (manually, modify ViewModel, DAO, create Repository and apply minor changes to fragment controllers). For the investigation, record CPU and memory usage from the app's Profiler and compare efficiency for Android app with LiveData. Add more logs and enhance app reliability.

Sketches:

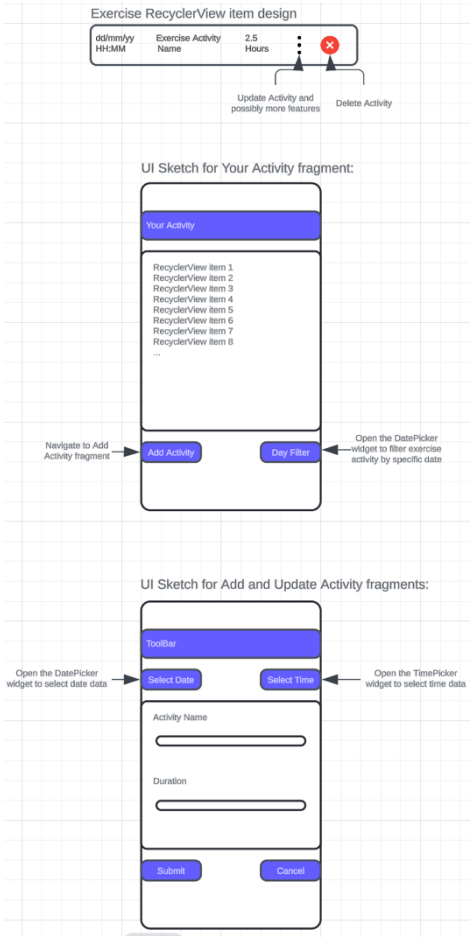

Figure. Initial Design Sketch

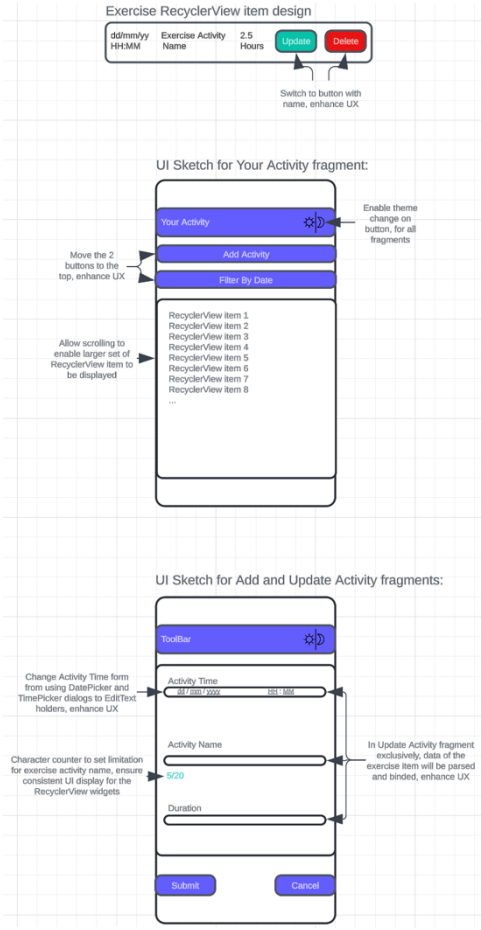

Figure. Final Design Sketch

Justifications of Changes in Design:

In the process of improving the design of the "Exercise Activity Tracker" app, key changes were made to enhance the user interface (UI) and user experience (UX). Below are the justifications for the major changes implemented between the initial and final design sketches:

1. **Usage of "Update" and "Delete" Buttons with Named Buttons:**
   - Initial Design: The original design used icons (e.g., kebab button and trash bin) to represent the update and delete functionalities in each RecyclerView item. While common, icons alone may not be universally understood by all users, especially when multiple icons are present.
   - Final Design: Switching to labelled buttons "Update" and "Delete" enhances UI/UX by making the functionalities more explicit and user-friendly. This change improves clarity and allows users to easily understand the actions associated with each button, reducing the learning curve for first-time users.

2. **Move "Add Activity" and "Filter By Date" Buttons to the Top:**
   - Final Design: By moving these buttons to the top of the screen, the most frequently used functionalities are now more accessible. This enhances user flow and makes it easier for users to access critical features without the need for searching. This change aligns with UX best practices, where frequently used actions should be placed in prominent, easy-to-reach areas of the interface.

3. **Remove DatePicker and TimePicker in Add/Update Fragments:**
   - Initial Design: The DatePicker and TimePicker widgets were used to input date and time data for activities. While functional, they reduced readability and understandability when hidden by default. Scrolling through months and years to select distant dates also became cumbersome for users.
   - Final Design: Replacing the DatePicker and TimePicker dialogs with more visible EditText input fields provides a more intuitive experience. Users can now directly input or adjust dates and times with more precision, avoiding the frustration of scrolling through calendars

for distant future dates. This enhances both readability and efficiency, as the input is now more accessible and visible, improving UX.

4. **Addition of Toggle Theme Button on Toolbar:**
   - Final Design: A theme toggle button has been added to the Toolbar to allow users to quickly switch between light and dark themes. This not only provides customization options for users who prefer different visual modes but also improves UI accessibility, especially in varying lighting conditions.
5. **Allow Scrolling in RecyclerView:**
   - Final Design: Enabling scrolling for the RecyclerView allows a larger set of exercise activities to be displayed, providing a more seamless navigation experience for users. Users can now scroll through their entire activity list without feeling restricted by screen size.
6. **Add Character Counter for Activity Name Input:**
   - Final Design: A character counter was added to limit the length of the activity name input to 20 characters. This ensures UI consistency within the RecyclerView widget, as longer text could cause layout issues. The counter dynamically displays the remaining characters, guiding users to provide concise activity names while maintaining the integrity of the UI.
7. **Pre-filled Input Fields in Update Activity Fragment:**
   - Final Design: In the UpdateActivityFragment, fields such as date, time, name, and duration are now pre-filled based on the data from the selected activity. This ensures that users are aware of the current data and can make more informed decisions when editing. By displaying the existing values, users are assured they are updating the correct activity, reducing potential errors and improving UX.

C. Key Features

- **Fragments for Modular Design**: The app utilizes multiple fragments like YourActivityFragment, AddActivityFragment, and UpdateActivityFragment, ensuring modularity and code separation.
- **Room Database for Data Persistence**: The app uses Room[1] for storing exercise data, ensuring persistent data between app sessions.
- **RecyclerView with Dynamic LiveData**[2]: The RecyclerView[3] displays all exercise activities, allowing users to view, update, or delete exercises from the list, in real-time.
- **ViewModel Integration:** The app uses the ViewModel[1] architecture component to store and manage UI-related data. The ViewModel is lifecycle-aware, meaning it survives configuration changes such as screen rotations. This ensures that data is not lost when the activity or fragment is recreated.
- **CRUD Operations**: The app fully supports Create, Read, Update, and Delete operations on exercise activities.
- **Concurrency**: Utilise Coroutines[13] to ensure consistent flow with delays either with backend data management and UI concurrency (Toast message display subsequentially). UI animation with flow also applied with exercise item's deletion action.
- **Theme Changing**: The app support light and dark theme changing[5].
- **Date Filter**: Using DatePicker widget as a dialog[9] to allow user filter exercise by date.
- **Date Sorting**: Using SimpleDateFormat[11] to sort out closest datetime data to now first.
- UI Animation: Red-over[14] RecyclerView item and Fade-out[15] animation on deletion.

D. Technical Usage

This app leverages a well-structured multi-fragment architecture with the following technical components:

1. **App Architecture:**
   - The architecture of the app follows the Model-View-ViewModel (MVVM) design pattern[1], ensuring a clear separation of concerns between the UI, business logic, and data handling. This architecture facilitates scalability and maintainability.
   - The app uses a multi-fragment architecture to handle different user interactions. The main components of the app include:

    - **YourActivityFragment**: Handles displaying the list of activities with a RecyclerView[3].
    - **AddActivityFragment**: Used to create a new exercise entry.
    - **UpdateActivityFragment**: Allows the user to update existing exercise details.
- The app makes extensive use of ViewModel[1] (ExerciseViewModel.kt) to handle data and manage the lifecycle of the fragments, ensuring that data persists across configuration changes.

2. Database:

- The app uses a Room database[1] (ExerciseDatabase.kt) to store exercise data persistently. Room is an abstraction layer over SQLite, providing type safety and easier database interactions.
- The database class is annotated with @Database, linking the Exercise entity and ExerciseDAO for performing CRUD operations.
- This database class also includes the ability to get a single instance, utilizing the Singleton design pattern for efficiency.

3. Entity:

- Exercise.kt is the entity class[1] representing the data model for an exercise item. It defines attributes such as id, name, date, time, and duration, which map to the corresponding columns in the SQLite database.
- The @Entity annotation in the Exercise.kt file ensures that this class represents a table in the database, while the @PrimaryKey annotation is used for defining the primary key of the table.

4. DAO:

- The DAO (Data Access Object) interface[1] (ExerciseDAO.kt) defines methods for interacting with the Exercise entity in the database. It contains methods for inserting, updating, deleting, and querying exercise data.
- The DAO handles the communication between the application and the Room database. For instance, it provides methods like insertExercise(), getAllExercises(), and deleteExercise() to manage data.

5. View Model:

- The ViewModel[1] (ExerciseViewModel.kt) mediates between the UI and the data repository. It allows the YourActivityFragment, AddActivityFragment, and UpdateActivityFragment to retrieve, observe, and modify data without directly accessing the database.
- It makes use of LiveData[2] to update the UI reactively whenever data changes. For example, when new exercises are added, the RecyclerView in YourActivityFragment.kt is updated automatically via observing LiveData[2] from the ViewModel.

6. Main Activity:

- The MainActivity.kt class serves as the host for the app's fragments. It uses the NavController to handle navigation between fragments.
- It also initializes the toolbar and handles fragment transactions for navigating between different parts of the app (e.g., from the list of exercises to adding a new exercise) and toggle theme changes[4].

7. Layout Files:

activity_main.xml This is the main layout for the app's MainActivity. It sets the foundation for the entire app, including the NavHostFragment, which hosts the different fragments (such as YourActivityFragment, AddActivityFragment, and UpdateActivityFragment) within the app.

- NavHostFragment: This acts as a container for hosting different fragments.
- Toolbar: The custom toolbar defined with the @id/toolbar is used to display the app's navigation bar, where options like the dark/light mode toggle button can be found.

exercise_item.xml This layout is designed for each item in the RecyclerView[3], representing an individual exercise. It includes:

- TextViews for displaying the exercise's datetime, name, and duration.

- Buttons for Update and Delete operations, which trigger corresponding actions (e.g., navigating to the UpdateActivityFragment or removing an exercise).
- This item view integrates well with the ExerciseAdapter, allowing for dynamic interaction with exercise data.

Figure. An exercise RecyclerView items with TextViews and buttons (applied color) elements.

fragment_your_activity.xml This layout is used in the YourActivityFragment and includes:

- RecyclerView for listing exercises.
- Buttons like Add Activity and Filter by Date to navigate to AddActivityFragment and allow users to filter their exercises by date.
- Material Design components are used to ensure the UI looks clean and modern.

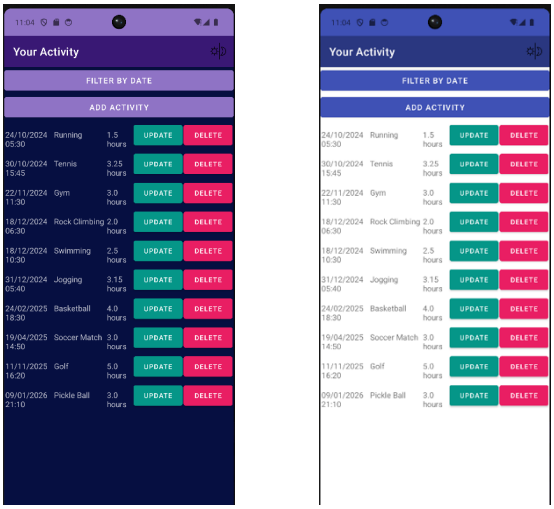

Figure. Your Activity Fragment in dark and light mode, showcasing Toolbar, buttons and RecyclerView items.

fragment_add_activity.xml This layout is responsible for collecting user input to add a new exercise. It contains:

- EditText components to capture the exercise's date and time (day, month, year, hour, and minute).
- TextWatcher to track input length in the activityNameInput, ensuring that the character count is displayed.
- Submit and Cancel buttons, which trigger the appropriate actions, allowing the user to add new data or discard the adding action.
- **Design Strategy:** The reason behind choosing a set of EditText holders (dd/mm/yyyy MM:HH) instead of using DatePicker and TimePicker for users to select their datetime input is to reduce the number of interaction times (e.g., clicks), which allow enhance UX and reduce effort to select data, especially when the desired date is very far away in the future, which require user to manually scrolling the calendar to find the date. In addition, the usage of different EditText holders also deliver a more direct approaches for user when it comes to input information, unlike DatePicker and

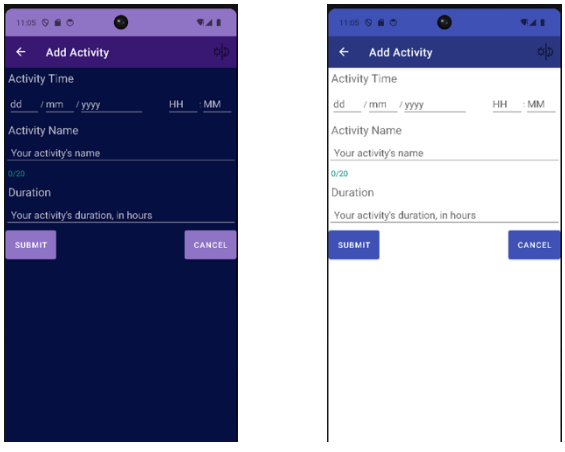

Figure. Add Activity Fragment in dark and light mode, showcasing Toolbar, TextView and different EditText input forms.

TimePicker, which the information are hidden at default, which also make them confuse and harder to use at the first time.

fragment_update_activity.xml This layout is almost identical to the AddActivityFragment, except it's used for updating an existing exercise. It includes:

- The same EditText fields for the user to modify the exercise details (except it will automatically set the existing details data to these fields, ensuring user is on the right exercise item as well as possibly evoke their sensibility when determining which (any) of the items should be altered).
- A TextWatcher that dynamically tracks changes to the activityNameInput.
- Submit and Cancel buttons, allowing the user to update or discard the changes.

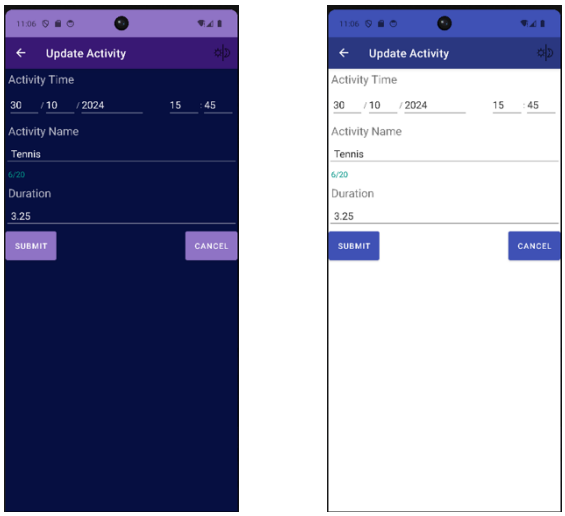

Figure. Update Activity Fragment in dark and light mode, showcasing Toolbar, TextView and different EditText input forms (with pre-set details).

8. Menu (toolbar_menu.xml):

This file defines the Menu used in the app's toolbar[10], specifically for handling the toggle between dark and light mode[5]. The menu consists of:

- A **day_night_toggle** item with an icon that allows the user to switch between themes[4]. This is linked to the MainActivity's onOptionsItemSelected method, which triggers the toggleTheme function.

Figure. Toolbar with fragment's id (name), toggle dark/light mode button of the right side and the theme color for dark and light mode.

9. Navigation (mobile_navigation.xml):

This file defines the navigation graph, mapping out how different fragments in the app are connected[7]:

- YourActivityFragment is the default starting fragment.
- AddActivityFragment and UpdateActivityFragment are accessible through actions defined in the navigation graph (action_yourActivityFragment_to_addActivityFragment and action_yourActivityFragment_to_updateActivityFragment).
- Argument of exerciseId (integer) is passed[8] to UpdateActivityFragment to allow accurate exercise item are being passed to be updated.
- Each fragment transition is represented as an action element, ensuring smooth navigation through the app using the NavController[7].

10. Values:

themes.xml: This file defines the app's theme. It applies material components and sets up global colors and styles, such as:

- **Theme.ExerciseActivityTracker**: The base theme for the entire app, which extends from Theme.MaterialComponents.DayNight.
- It ensures that both day and night modes are supported by default[5].

light-theme-mode (values/colors.xml): The colors.xml file defines all color resources used when the light theme is active (default theme). The theme color-set chosen for light mode is blue and white.

- **Primary and Secondary Colors**: These include primary_color, primary_variant_color, and secondary_color, which are used throughout the app to maintain consistency.
- **Background and Text Colors**: background_color, black, and white are the main colors defining the appearance of text and background elements.

dark-theme-mode (values-night/colors.xml): This file defines color values for the dark theme. The theme color-set chosen for light mode is purple and dark blue.

- Similar to the light theme's colors.xml, it includes primary_color, secondary_color, and background/text color definitions, but with values suited for a darker theme. This ensures that the UI adjusts dynamically based on the user's selected theme, providing an optimized user experience in both light and dark modes.

11.App Logo:

The app's logo was generated by DALL-E Gen AI tools[22]. Stored in res/drawable.
The logo signifies the atheism spirit of the app, inspiring the user to start their exercising activities, using some of the key theme color applied with the app UI.

E. Technical Analysis

1. Exercise.kt (Entity)

- **@Entity**: The Exercise class is annotated with @Entity to map it to a database table named "exercises".
- **Primary Key**:

```
@PrimaryKey(autoGenerate = true) val id: Int = 0,
```

This marks the id field as the primary key and instructs Room to automatically generate the key value for each new entry (e.g., automatically increment id value for each new entry created).

- **Other Fields**: The fields like name, time, duration represent various attributes of an exercise entry. These are mapped to columns in the SQLite table.
  - **activityTime**: The date and time of the exercise, as a non-nullable string.
  - **activityName** The name of the exercise, as a non-nullable string.
  - **duration**: Stores the duration of the exercise in hours, as float.

2. ExerciseDAO.kt (Data Access Object)

- **@Insert**:

```
@Insert
suspend fun insertExercise(exercise: Exercise)
```

This method inserts a new Exercise entry into the database. The suspend keyword makes it a coroutine-friendly method[13], meaning it can be executed asynchronously without blocking the main thread.

- **@Update**:

```
@Update
suspend fun updateExercise(exercise: Exercise)
```

This method updates an existing Exercise entry in the database based on the ID. Again, it's coroutine-friendly[13] to avoid blocking the UI thread.

- **@Delete**:

```
@Delete
suspend fun deleteExercise(exercise: Exercise)
```

This method deletes an Exercise entry from the database.

- **Query to fetch all exercises**:

```
@Query("SELECT * FROM exercises")
fun getAllExercises(): LiveData<List<Exercise>>
```

This method returns a list of all exercises wrapped in LiveData, allowing the UI to reactively update when the data changes.

- **Query to fetch all exercises with matching activityTime**:

```
@Query("SELECT * FROM exercises WHERE activityTime LIKE :date")
fun getExercisesByDate(date: String): LiveData<List<Exercise>>
```

This method returns a list of all exercises matching activityTime wrapped in LiveData.

- **Query to fetch exercise matching id**:

```
@Query("SELECT * FROM exercises WHERE id = :id")
fun getExerciseById(id: Int): LiveData<Exercise>
```

This method returns a list of all exercises matching id wrapped in LiveData.

3. ExerciseDatabase.kt (Database)

- **@Database Annotation**:

```
@Database(entities = [Exercise::class], version = 1, exportSchema = true)
```

This specifies that the database handles the Exercise entity and is currently at version 1, and allowing export JSON files represent the database's schema history

- **Singleton Pattern**:

```
companion object {
    @Volatile
    private var INSTANCE: ExerciseDatabase? = null
```

The singleton pattern is used to ensure only one instance of the ExerciseDatabase is created. This is important to avoid having multiple connections to the database.

- **getDatabase method**: This method returns the singleton instance, creating the database if it doesn't already exist:

```
fun getDatabase(context: Context): ExerciseDatabase {
    return INSTANCE ?: synchronized(this) {
        val instance = Room.databaseBuilder(
            context.applicationContext,
            ExerciseDatabase::class.java,
            "exercise_database"
        ).build()
        INSTANCE = instance
        instance
    }
}
```

4. ExerciseViewModel.kt (ViewModel)

- **LiveData**:

```
val allExercises: LiveData<List<Exercise>> = exerciseDao.getAllExercises()
```

This property observes the list of all exercises, wrapped in LiveData. Whenever the data changes, the UI automatically updates.

- **Insert/Update/Delete Methods**: These methods allow adding, updating, and removing exercises from the database, with coroutines being used to ensure that the operations happen on a background thread.

```
fun insertExercise(exercise: Exercise) = viewModelScope.launch {
    exerciseDao.insertExercise(exercise)
}

fun updateExercise(exercise: Exercise) = viewModelScope.launch(Dispatchers.IO) {
    delay(1000)
    exerciseDao.updateExercise(exercise)
}
fun deleteExercise(exercise: Exercise) = viewModelScope.launch(Dispatchers.IO) {
    delay(1000)
    exerciseDao.deleteExercise(exercise)
}
```

- **Get Exercise by Date**: This method allow getting a list of exercise items corresponding to a date value, helping the filtration logic when filtering exercise item by the date.

```
fun getExercisesByDate(date: String): LiveData<List<Exercise>> {
    return exerciseDao.getExercisesByDate("%$date%")
}
```

- **Get Exercise by ID**: This method allow getting an exercise items corresponding to its id value, helping the exercise item being forwarded from Your Activity to Update Activity fragment effectively.

```
fun getExerciseById(id: Int): LiveData<Exercise> {
    return exerciseDao.getExerciseById(id)
}
```

5. MainActivity.kt:

This file manages the main application activity, serving as the entry point for the app. It hosts the navigation between fragments and handles UI elements such as theme toggling. Below is an explanation of each important method:

- **onCreate(savedInstanceState: Bundle?)**:
    - Initializes the activity's UI by calling setContentView() with the activity_main layout.
    - Sets up the Toolbar as the ActionBar[10] using setSupportActionBar().
    - Initializes the **NavController** by finding the NavHostFragment[7]. This sets up navigation to handle fragment transitions and allows ActionBar titles to dynamically update based on the current fragment.
    - **Key aspect**: Navigation is integrated using Android's Navigation component[7], linking the toolbar with the fragment-based navigation system[6].

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    // Set up the toolbar as the ActionBar
    val toolbar: androidx.appcompat.widget.Toolbar = findViewById(R.id.toolbar)
    setSupportActionBar(toolbar)
    // Set up navigation with the ActionBar to handle fragment title
    val navHostFragment =
        supportFragmentManager.findFragmentById(R.id.nav_host_fragment) as
androidx.navigation.fragment.NavHostFragment
    val navController = navHostFragment.navController
    setupActionBarWithNavController(navController)
    // Ensure the fragment's title is displayed
    supportActionBar?.setDisplayShowTitleEnabled(true)
}
```

- **onCreateOptionsMenu(menu: Menu?)**:
    - Inflates the menu options in the toolbar, adding the day/night theme toggle button.
    - **Key aspect**: It ensures the options menu is linked with the toolbar, allowing interaction with items like the day/night mode toggle.
- **onOptionsItemSelected(item: MenuItem)**:
    - Responds to user interaction with the options menu. When the theme toggle button is clicked (R.id.day_night_toggle), the app switches between light and dark mode using the toggleTheme() function.
    - **Key aspect**: Theme switching is a core feature handled here, and this method effectively controls the app's appearance.

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.day_night_toggle -> {
            toggleTheme()
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

- **toggleTheme()**:
    - Toggles between dark and light mode by checking the current uiMode. If the app is in night mode, it switches to day mode, and vice versa.
    - **Key aspect**: Uses AppCompatDelegate.setDefaultNightMode() to switch between themes dynamically at runtime.

- **onSupportNavigateUp()**:
    - Handles back navigation using the NavController[7]. It ensures proper behavior when navigating up in the fragment stack.
    - **Key aspect**: Ensures a seamless experience when navigating between fragments in the app.

6. YourActivityFragment.kt:

This fragment displays the list of exercises using a RecyclerView and allows the user to filter and manage activities. Below is an explanation of its critical methods:

- **Key Variable Declarations**:

  exerciseViewModel:
    - Inflates the fragment's layout, initializes the RecyclerView, and sets up the ViewModel (ExerciseViewModel).
    - Declares a lateinit variable of type ExerciseViewModel. The ViewModel is an architecture component designed to store and manage UI-related data. Here, it holds a reference to the exercise-related data (a list of exercise activities) for this fragment.
    - **Purpose**: ExerciseViewModel will be used to observe the exercise data and manage interactions (such as adding, deleting, or filtering exercises). It ensures data is retained across configuration changes like device rotation.

  adapter:
    - Declares a lateinit variable of type ExerciseAdapter. This adapter is responsible for binding the exercise data to the RecyclerView, displaying each exercise item.
    - **Purpose**: The adapter handles rendering of the data in the RecyclerView, and it includes the logic to handle update and delete actions for each exercise item.

- **onCreateView()**:
    - Inflates the fragment's layout, initializes the RecyclerView, and sets up the ViewModel (ExerciseViewModel).

      Set Up:

```
inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
```

RecyclerView Initialization:

```
val view = inflater.inflate(R.layout.fragment_your_activity, container, false)
val recyclerView = view.findViewById<RecyclerView>(R.id.recyclerView)
recyclerView.layoutManager = LinearLayoutManager(requireContext())
```

ViewModel Initialization:

```
exerciseViewModel = ViewModelProvider(this)[ExerciseViewModel::class.java]
```

    - If the fragment is restored after a configuration change (such as screen rotation or theme change), it restores the saved exercises (activityTime, activityName, duration, id) from the savedInstanceState.

```
if (savedInstanceState != null) {
    val savedExercises = savedInstanceState.getParcelableArrayList("savedExercises", Bundle::class.java)?.map { bundle ->
        Exercise(
            activityTime = bundle.getString("activityTime", ""),
            activityName = bundle.getString("activityName", ""),
            duration = bundle.getFloat("duration"),
            id = bundle.getInt("id")
        )
    }
```

    - The exerciseViewModel.allExercises.observe() method sets up an observer on the LiveData object. Whenever the list of exercises changes (e.g., new exercises are added or deleted), the UI is automatically updated.

  Filtering: The exercises are filtered to include only upcoming exercises (excluding past activities – in both date and time). It uses a SimpleDateFormat[11] to parse and compare the exercise times with the current date.

  Sorting: The filtered exercises are sorted by their date and time.

  Adapter Update: After filtering and sorting, the ExerciseAdapter is updated with the new list of upcoming exercises (today and excluding past events), ensuring non-null value, using isSameDate

and hasPassedToday helper methods, ensuring data is only from today, then rearrange with the mentioned sorting and filtering algorithms to show the latest to now activity first.

Update Activity: Once user click on Update button, the navigation action actionYourActivityFragmentToUpdateActivityFragment redirect user to Update Activity fragment, alongside with passing the RecyclerView id of that exercise item chosen.

Update Activity: Once user click on Delete button, the system call deleteExercise function to remove that chosen exercise, alongside with triggering the showUndoSnackbar method, which show a SnackBar widget indicating removal action and passing the exercise item allowing retrieval (insert back) of deleted item.

```
exerciseViewModel.allExercises.observe(viewLifecycleOwner) { exercises ->
    val upcomingExercises = exercises.filter { exercise ->
        val exerciseDateTime = SimpleDateFormat(
            "dd/MM/yyyy HH:mm",
            Locale.getDefault()
        ) // get datetime in correct format from Locale
            .parse(exercise.activityTime) // Parse both date and time from the activity
        val currentDateTime = Calendar.getInstance().time // Current date and time
        // Only include activities where the date is today but the time is in the future, or any future dates (exclude past event)
        exerciseDateTime?.after(currentDateTime) ?: false ||
                (exerciseDateTime != null && isSameDay(exerciseDateTime, currentDateTime) && !hasPassedToday(
                    exerciseDateTime,
                    currentDateTime
                )) // External helper functions, scenario if exercise item is today but the time has passed
    }.sortedBy { exercise -> // Sorted
        SimpleDateFormat("dd/MM/yyyy HH:mm", Locale.getDefault())
            .parse(exercise.activityTime)
    }
    // Initialize the adapter with filtered and sorted data
    adapter = ExerciseAdapter(upcomingExercises, { exercise ->
        // Navigate to Update Activity fragment, passing the corresponding exercise id
        val action = YourActivityFragmentDirections
            .actionYourActivityFragmentToUpdateActivityFragment(exercise.id)
        Log.d(
            "YourActivityFragment",
            "Update activity button clicked with exercise activity id " + exercise.id
        ) // Logs
        findNavController().navigate(action)
    }, { exercise ->
        // Delete exercise activity
        exerciseViewModel.deleteExercise(exercise) // Delete with concurrency applied to delay 1s
        showUndoSnackbar(
            view,
            exercise
        ) // Show SnackBar widget notify user upon deletion and allow restoration (undo)
        Log.d("YourActivityFragment", "Delete activity button clicked") // Logs
    })
    recyclerView.adapter = adapter
}
```

- **Key aspect**: Utilizes ViewModel and LiveData to maintain and observe exercise data, ensuring the RecyclerView is updated dynamically when exercise data changes.

- **filterByDate()**:
  - DatePickerDialog: When the user clicks the "Filter by Date" button, this block shows a DatePicker dialog[9] where the user can select a date. The selected date is formatted into a string (e.g., dd/MM/yyyy).
  - Filtering Exercises: The getExercisesByDate() function in the ExerciseViewModel fetches exercises for the selected date. It observes the result and updates the RecyclerView with the filtered exercises by calling adapter.updateExerciseList().
  - **Key aspect**: Implements date-based filtering for better user interaction with the exercise list.

```
view.findViewById<Button>(R.id.filterByDateButton).setOnClickListener {
    // Initialise DatePicker widget and context of day/month/year
    val calendar = Calendar.getInstance()
```

```
    val year = calendar.get(Calendar.YEAR)
    val month = calendar.get(Calendar.MONTH)
    val day = calendar.get(Calendar.DAY_OF_MONTH)
    // Format the DatePicker dialog with parsed data
    val datePickerDialog = DatePickerDialog(
        requireContext(),
        { _, selectedYear, selectedMonth, selectedDay ->
```

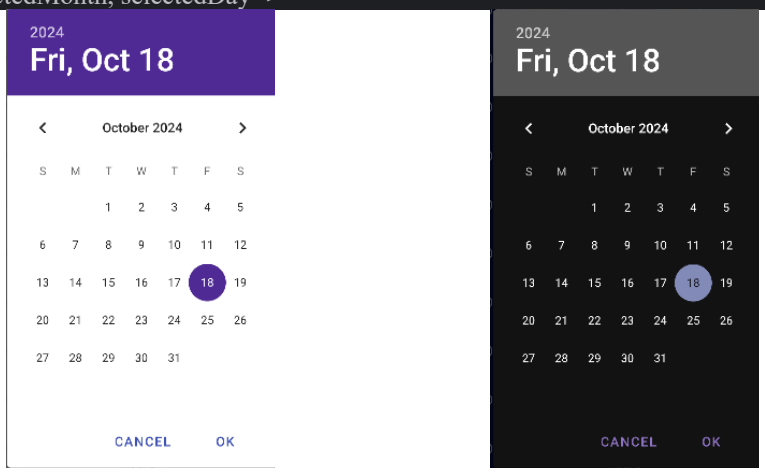

Figure. DatePicker widget in dark and light mode.

```
            val formattedDate = String.format("%02d/%02d/%04d", selectedDay, selectedMonth + 1, selectedYear)
            Log.d("YourActivityFragment - DatePicker", "Datetime selection $formattedDate") // Logs
            // Filter the exercise list based on the selected date // LiveData
            exerciseViewModel.getExercisesByDate(formattedDate).observe(viewLifecycleOwner) { exercises ->
                adapter.updateExerciseList(exercises)
            }
        },
        year, month, day
    )
    datePickerDialog.show() // Show DatePicker dialog
}
```

- **Navigate to Add Activity**:
    - To redirect to Add Activity fragment, we use the navigation action action_yourActivityFragment_to_addActivityFragment.
- **showUndoSnackbar()**:
    - Displays a Snackbar when an exercise is deleted, offering an "Undo" option to restore the deleted exercise. Deleted item will be reinserted back.
    - **Key aspect**: Provides user feedback and recovery options, especially when user delete data by accident, enhancing UX.

```
private fun showUndoSnackbar(view: View, deletedExercise: Exercise) {
    val snackbar = Snackbar.make(view, "Exercise activity removed", Snackbar.LENGTH_LONG)
    snackbar.setAction("UNDO") {
        // Re-insert the deleted exercise into the database when "UNDO" is clicked
        exerciseViewModel.insertExercise(deletedExercise)
        Log.d("YourActivityFragment - UNDO", "Exercise activity retrieved $deletedExercise") // Logs
    }
    snackbar.show()
}
```

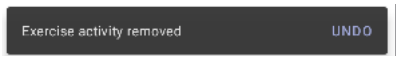

Figure. SnackBar widget notify deletion with UNDO option.

- **onSaveInstanceState(outState: Bundle)**:
    - Saves the current list of exercise features (activityTime, activityName, duration, id) to outState to persist data during configuration changes (Orientation change and Theme change).
    - **Key aspect**: Ensures data is retained during activity or fragment lifecycle events, preventing data loss on rotation or theme change.

- **Time Checker Helper methods:**
  - The 2 methods facilitate the helper algorithms to check if datetime value of the exercise item is at the same date as today (isSameDay) or if they has been passed / in the past (hasPassedToday).
  - **Key aspect**: Refining the exercise items that only contain items from today to the future.

```
private fun isSameDay(date1: Date?, date2: Date?): Boolean {
    val calendar1 = Calendar.getInstance().apply { time = date1!! } // nullable
    val calendar2 = Calendar.getInstance().apply { time = date2!! } // nullable
    return calendar1.get(Calendar.YEAR) == calendar2.get(Calendar.YEAR) &&
            calendar1.get(Calendar.DAY_OF_YEAR) == calendar2.get(Calendar.DAY_OF_YEAR)
}
private fun hasPassedToday(exerciseDateTime: Date?, currentDateTime: Date?): Boolean {
    return exerciseDateTime?.before(currentDateTime) ?: true
}
```

7. AddActivityFragment.kt:

This fragment allows the user to add a new exercise activity. It validates user inputs and adds a new exercise to the database.

- **ViewModel and Coroutine Scope Initialization:**

exerciseViewModel:.

  - A ViewModel is responsible for managing and holding UI-related data. It survives configuration changes such as screen rotations.
  - ExerciseViewModel is the main data provider in this fragment. It allows interaction with the database, including inserting new exercise entries.

charCountTextView:.

  - This TextView is used to display the character count for the activity name input field.
  - It's dynamically updated as the user types into the activityNameInput field.

coroutineScope:.

  - A coroutine scope using Dispatchers.Main[13] is initialized to handle UI-related tasks. It enables executing tasks asynchronously (like validation or showing Toast messages) on the main thread without blocking the UI.

```
private lateinit var exerciseViewModel: ExerciseViewModel
private lateinit var charCountTextView: TextView
private val coroutineScope = CoroutineScope(Dispatchers.Main)
```

- **onCreateView() - Setting up Views and Event Handlers:**
  - This function inflates the fragment's layout (fragment_add_activity.xml) and initializes the UI elements.
  - It connects the layout's UI components (e.g., buttons, EditText fields) to the Kotlin code.

```
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    val view = inflater.inflate(R.layout.fragment_add_activity, container, false)
```

- **ViewModel Initialization and Input Fields Setup:**

exerciseViewModel = ViewModelProvider(this)[ExerciseViewModel::class.java]:.

  - The ViewModelProvider instantiates ExerciseViewModel and associates it with this fragment's lifecycle, allowing data management and persistence.

Initializing Input Fields:

  - Each EditText field (dayInput, monthInput, etc.) corresponds to user input for the date and time of the exercise.
  - These fields will later be used to validate and create a new Exercise object when the user submits the form.

```
exerciseViewModel = ViewModelProvider(this)[ExerciseViewModel::class.java]

// Initialize the EditText fields, including Activity Name, Time Inputs and Duration Input
val dayInput = view.findViewById<EditText>(R.id.dayInput)
val monthInput = view.findViewById<EditText>(R.id.monthInput)
val yearInput = view.findViewById<EditText>(R.id.yearInput)
```

```
val hourInput = view.findViewById<EditText>(R.id.hourInput)
val minuteInput = view.findViewById<EditText>(R.id.minuteInput)
```

- **Handling Text Changes in the Activity Name:**

This block adds a text watcher to the activityNameInput field to monitor character count in real time.

- Character Count: Updates charCountTextView to reflect the current number of characters typed by the user (e.g., 5/20, initially 0/20).
- Color Change: If the character count exceeds 20, the text color of the charCountTextView changes to red to visually alert the user.
- TextWatcher: This is a built-in Android interface that allows monitoring text changes in an EditText. It has three main methods: afterTextChanged, beforeTextChanged, and onTextChanged.

```
val activityNameInput = view.findViewById<EditText>(R.id.activityNameInput)
charCountTextView =
   view.findViewById(R.id.wordCountTextView) // word count TextView (e.g., 0/20)
val durationInput = view.findViewById<EditText>(R.id.durationInput)
// Listen for activity name input changes and update character count
activityNameInput.addTextChangedListener(object : TextWatcher {
   override fun afterTextChanged(s: Editable?) {
      val charCount = s?.length ?: 0
      charCountTextView.text = "$charCount/20" // Count name character TextView
      // Dynamically update character counting text reflecting color when exceeding the limit
      val colorRes = if (charCount > 20) R.color.red else R.color.green
      charCountTextView.setTextColor(ContextCompat.getColor(requireContext(), colorRes))
   }
   override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {}
   override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {}
})
```

- **Submit Button - Validating Input and Adding a New Exercise:**

Purpose:

- When the user clicks the submit button, the app validates the form fields and adds a new exercise entry to the database if the input is valid.
- **CoroutineScope**: This coroutine[13] is used to execute tasks asynchronously on the main thread without blocking the UI. It also allows sequential validation and feedback processes.

Validation:

- Validation checks are performed to ensure that the input fields are valid.
- **Error Handling**: If validation fails, the set of error messages will be collected and the user is presented subsequentially with error messages via Toast notifications, and the exercise is not added.

Creating and Adding the Exercise:

- If the form passes validation, a new Exercise object is created with the provided data.
- The exercise is then inserted into the database using exerciseViewModel.insertExercise().

Feedback and Navigation:

- After adding the exercise, a Toast message informs the user that the new exercise was added successfully.
- The app then navigates back to the YourActivityFragment.

```
view.findViewById<Button>(R.id.submitButton).setOnClickListener {
   // Validate and provide feedback using launch coroutines (propagate Toast feedbacks with delays)
   coroutineScope.launch {
      val errorMessages =
         mutableListOf<String>()
// All validation logics …
// Display error Toast feedbacks sequentially (if not empty)
if (errorMessages.isNotEmpty()) {
   for (message in errorMessages) {
      showToastSequentially(message)
   }
   return@launch // Not proceed through
}

// Define new exercise item
```

```
val activityTime = "$day/$month/$year $hour:$minute"
val newExercise = Exercise(
    activityTime = activityTime,
    activityName = activityName,
    duration = duration!! // Assert non-null value since the usage of coroutine could return null value
)
```

- **Input Validation - Date, Time, and Duration Checks:**

Purpose: The function checks the validity of the user inputs

  - isValidDateTime(): This method ensures that the date and time entered by the user follow a valid format.
  - isValidRangeDateTime(): This method ensures the entered day, month, hour, and minute fall within valid .
  - Activity Name Validation: Ensures the name has between 2 and 20 characters.
  - Duration Validation: Ensures that a valid duration is not null.

```
            // Validate Datetime Format
            if (!isValidDateTime(day, month, year, hour, minute)) {
                errorMessages.add("Invalid Time Format") // Toast with concurrency
                Log.d("AddActivityFragment", "Invalid Activity Time") // Logs
            }
            // Validate Datetime Range
            if (!isValidRangeDateTime(day, month, hour, minute)) {
                errorMessages.addAll(getRangeErrors(day, month, hour, minute))
            }

            // Validate Activity Name character count
            val activityName = activityNameInput.text.toString()
            val charCount = activityName.length
            if (charCount > 20 || charCount < 2) {
                errorMessages.add("Invalid Activity Name") // Toast with concurrency
                Log.d("AddActivityFragment", "Invalid Activity Name") // Logs
            }

            // Validate Duration
            val durationText = durationInput.text.toString()
            val duration = durationText.toFloatOrNull()
            if (duration == null) {
                errorMessages.add("Duration cannot be empty") // Toast with concurrency
                Log.d("AddActivityFragment", "Invalid Activity Duration") // Logs
            }

            // Display error Toast feedbacks sequentially (if not empty)
            if (errorMessages.isNotEmpty()) {
                for (message in errorMessages) {
                    showToastSequentially(message)
                }
                return@launch // Not proceed through
            }
```

Error Handling:

  - If any of the checks fail, the corresponding error message is added to the errorMessages list, and the function exits early (returning before adding the exercise).

- **Utility Functions – DateTime Validation and Toast Feedbacks:**
  - isValidDateTime(): This method ensures that the date and time entered by the user follow a valid format (e.g., day, month, hour and minute have two digits, year has four digits).
  - getRangeErrors(): This method ensures the entered day, month, hour, and minute fall within valid ranges (e.g., day is between 1 and 31, month is between 1 and 12, hour is between 0 and 23 and minute is between 0 to 59). Return and append all error messages to the list of error message.

```
// Method to validate date and time format (e.g, validate correct format dd/mm/yyyy HH:MM, entry not null)
private fun isValidDateTime(day: String, month: String, year: String, hour: String, minute: String): Boolean {
    return day.length == 2 && month.length == 2 && year.length == 4 &&
            hour.length == 2 && minute.length == 2 &&
```

```
        day.toIntOrNull() != null && month.toIntOrNull() != null &&
        year.toIntOrNull() != null && hour.toIntOrNull() != null && minute.toIntOrNull() != null
}

// Collect and append error messages for invalid ranges to the list
private fun getRangeErrors(day: String, month: String, hour: String, minute: String): List<String> {
    val errorMessages = mutableListOf<String>()
    val dayInt = day.toIntOrNull() ?: return listOf("Invalid Time: Day must be a number")
    val monthInt = month.toIntOrNull() ?: return listOf("Invalid Time: Month must be a number")
    val hourInt = hour.toIntOrNull() ?: return listOf("Invalid Time: Hour must be a number")
    val minuteInt = minute.toIntOrNull() ?: return listOf("Invalid Time: Minute must be a number")

    if (dayInt !in 1..31) errorMessages.add("Invalid Time: Day must be between 1 and 31")
    if (monthInt !in 1..12) errorMessages.add("Invalid Time: Month must be between 1 and 12")
    if (hourInt !in 0..23) errorMessages.add("Invalid Time: Hour must be between 0 and 23")
    if (minuteInt !in 0..59) errorMessages.add("Invalid Time: Minute must be between 0 and 59")

    return errorMessages
}
```

- **Cancel Button – Navigate back:**
  - If the user clicks the cancel button, the app navigates back to the YourActivityFragment, discarding any data entered in the form.
  - findNavController()**:** This method is used to navigate between fragments, uses the navigation action action_yourActivityFragment_to_addActivityFragment.
- **Toast Feedback Mechanism:**
  - Displays error messages sequentially with a 1-second delay between each Toast. This method ensures that multiple error messages don't overlap or get missed by the user, allowing all displayed subsequentially.

8. UpdateActivityFragment.kt:

This fragment allows the user to update an existing exercise. It fetches the exercise details from the database, populates the form fields with the current data, and updates the exercise upon form submission.

- **ViewModel, Coroutine Scope, and Argument Passing Initialization:**
  - exerciseViewModel, charCountTextView, and coroutineScope[13] are similar features implemented in AddActivityFragment.kt.
  - args: navArgs() is a safe-args feature from Android's Navigation component[7]. It ensures that the exercise ID, passed from the previous fragment[8] (YourActivityFragment), is securely available in this fragment.

```
private val args: UpdateActivityFragmentArgs by navArgs()
```

- **onCreateView() – Fragment View Setup:**
  - The onCreateView method inflates the layout fragment_update_activity.xml and initializes the UI components (EditText fields, TextViews, Buttons).
  - This function is the entry point where the fragment's UI is set up and the ViewModel is prepared to fetch the exercise that needs to be updated.
- **Fetching the Exercise to Update:**
  - The ViewModel fetches the Exercise data by using the passed exercise ID (args.exerciseId).
  - **LiveData** is used here to observe changes in the database. If the data changes (for instance, the exercise gets updated by another process), the UI will reflect those changes automatically.

```
exerciseViewModel.getExerciseById(args.exerciseId).observe(viewLifecycleOwner) { exercise ->
    currentExercise = exercise
```

- **Populating the EditText Fields with Current Data:**

Purpose:

  - The exercise's activityTime (stored as a string in the format "dd/MM/yyyy HH:mm") is split into two parts:
    - date: Contains the day, month, and year.
    - time: Contains the hour and minute.

- Each part is then assigned to its respective EditText field (e.g., dayInput, monthInput, etc.), allowing the user to modify the values.
- Observe and pre-set text value of current exercise's name and duration.

Real-time Updating:

- This approach ensures that when the fragment opens, the existing exercise data is displayed for editing, enhance UX.

```
val dateTime = currentExercise.activityTime.split(" ")
val date = dateTime[0].split("/")
val time = dateTime[1].split(":")
// Get saved data from date and time of the exercise object with index
dayInput.setText(date[0])
monthInput.setText(date[1])
yearInput.setText(date[2])
hourInput.setText(time[0])
minuteInput.setText(time[1])
// Observe name and duration from exercise object
activityNameInput.setText(currentExercise.activityName)
durationInput.setText(currentExercise.duration.toString())
```

- **Common usage with Add Activity Fragment:**

Update Activity Fragment also utilise similar approach when handling user input, similarly to Add Activity Fragment, including Character Count for Activity Name input and checker, DateTime validations and appending error messages to a list, showcasing Toast message subsequentially by each 1 seconds reflecting errors with user's exercise item's details. Once information are valid, exercise RecyclerView item can be updated accordingly, and navigate back to Your Activity Fragment. Similarly, Cancel button can navigate back to Your Activity Fragment.

9. ExerciseAdapter.kt:

This ExerciseAdapter class is responsible for displaying the list of exercise activities in a RecyclerView. It manages how each item is displayed, responds to user interactions (like update and delete), and handles deletion animations for visual feedback.

- **Class Properties**:

exercises:

- List of Exercise objects passed to the adapter. Each Exercise represents a single item in the RecyclerView. The adapter will render this list by creating and binding ViewHolders for each item.

onUpdateClick:

- When clicking on Update button for a specific exercise, the function is passed as a parameter when initializing the adapter, allowing the parent component to define what happens when an item is updated.

onDeleteClick:

- When clicking on Update button for a specific exercise, it allows the parent to manage the delete logic.

```
private var exercises: List<Exercise>,
private var onUpdateClick: (Exercise) -> Unit,
private var onDeleteClick: (Exercise) -> Unit
```

- **ExerciseViewHolder Class**:

Purpose:

- This inner class holds references to the various UI components (TextViews and Buttons) of a single exercise item. The adapter uses this class to manipulate the views (e.g., setting the exercise name, time, and duration).

ViewHolder Design Pattern:

- The ViewHolder pattern helps improve performance by avoiding repeated calls to findViewById when scrolling through a list of items. The views are bound once and then reused as the RecyclerView scrolls.

```
inner class ExerciseViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
    val activityTime: TextView = itemView.findViewById(R.id.activityTime)
    val activityName: TextView = itemView.findViewById(R.id.activityName)
```

```
    val duration: TextView = itemView.findViewById(R.id.duration)
    val updateButton: Button = itemView.findViewById(R.id.updateButton)
    val deleteButton: Button = itemView.findViewById(R.id.deleteButton)
}
```

- **onCreateViewHolder()**:
    - This method is responsible for inflating the layout for a single exercise item (exercise_item.xml) and creating an instance of the ExerciseViewHolder.
    - The RecyclerView calls this method when it needs a new view holder to represent an item. The layout inflates the UI for the list item, which is then passed to the ExerciseViewHolder.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ExerciseViewHolder {
    val view = LayoutInflater.from(parent.context)
        .inflate(R.layout.exercise_item, parent, false)
    return ExerciseViewHolder(view)
}
```

- **onBindViewHolder()**:
    - Binding Data: This method binds data from the Exercise object at the current position in the list to the corresponding ViewHolder. For each exercise, the activityTime, activityName, and duration values are set to their respective TextViews.
    - Update Button: When the update button is clicked, the onUpdateClick function is invoked with the corresponding exercise item, allowing the parent component (e.g., the fragment) to handle navigation to the update screen.
    - Delete Button: When the delete button is clicked, the animateItemDeletion function is called to animate the removal of the item, and the onDeleteClick function is triggered to handle the actual deletion process (e.g., removing the exercise from the database).
    - UI Update: notifyDataSetChanged()[12] is called after deletion to refresh the entire list, reflecting the removal of the item. This ensures the RecyclerView stays up-to-date.
    - Dynamic Button Handling: Each exercise has its own update and delete buttons, and the onBindViewHolder() method connects these buttons to their respective logic (update or delete).

```
override fun onBindViewHolder(holder: ExerciseViewHolder, position: Int) {
    val exercise = exercises[position]
    holder.activityTime.text = exercise.activityTime
    holder.activityName.text = exercise.activityName
    holder.duration.text = "${exercise.duration} hours" // Adapting format e.g., 2.5 hours
    // Handle Update button
    holder.updateButton.setOnClickListener { onUpdateClick(exercise) }
    // Handle Delete button
    holder.deleteButton.setOnClickListener {
        // Trigger the animation
        animateItemDeletion(holder.itemView) {
            onDeleteClick(exercise)  // Notify the deletion to the parent class
            notifyDataSetChanged()   // Refresh the adapter once deletion is confirmed
        }
    }
}
```

- **getItemCount()**: This method returns the total number of items in the exercises list. The RecyclerView uses this information to determine how many items it needs to display and adapter needs this function to know when to stop creating or binding new ViewHolders.

- **Updating the Adapter Data:**
    - This method updates the exercises list with new data and then refreshes the entire adapter. The notifyDataSetChanged()[12] call ensures that the UI is re-rendered to display the updated list of exercises.

```
fun updateExerciseList(newExercises: List<Exercise>) {
    this.exercises = newExercises
    notifyDataSetChanged()
}
```

- **Retrieving Exercise**: This is a simple getter method that returns the current list of exercises being displayed in the RecyclerView.

- **Animation for Item Deletion:**
  - **Visual Feedback**: When the user deletes an exercise, this animation method (ArgbEvaluator)[14] provides a visual cue that the item is being removed. First, the background color gradually changes from white to red, and then the item fades out.
  - **Sequential Execution**: The animation is set up with a listener that triggers the fade-out effect (AlphaAnimation)[15] after the background color has fully changed.
  - **Concurrency**: The animation happens on the UI thread but does not affect other operations (onAnimationEnd function evoke when the effect has finished), allowing the user to see the deletion process without any lag or blocking.

```
private fun animateItemDeletion(view: View, onAnimationEnd: () -> Unit) {
    // Set default and destined color
    val colorFrom = ContextCompat.getColor(view.context, android.R.color.white)
    val colorTo = ContextCompat.getColor(view.context, android.R.color.holo_red_dark)
    // Method to gradually change color
    val colorAnimation = ObjectAnimator.ofObject(
        view, "backgroundColor", ArgbEvaluator(), colorFrom, colorTo
    )
    colorAnimation.duration = 1000 // 1 second to change color
    // Fading animation
    colorAnimation.addUpdateListener { animator ->
        if (animator.animatedFraction == 1f) {
            // After the color change, start the fade-out animation
            val fadeOut = AlphaAnimation(1f, 0f)
            fadeOut.duration = 1000 // 1 second to fade out
            fadeOut.fillAfter = true
            view.startAnimation(fadeOut)
            onAnimationEnd() // Notify parent after animation ends
        }
    }
    colorAnimation.start()
}
```

Figure. Red-over animation with the item on deletion.

F. Espresso and JUnit UI Testings

1. YourActivityTest.kt

- **Purpose:** This test class verifies the correct behavior of the "Your Activity" fragment, ensuring users can view and interact with their activities.

- **UIElementExist Test:**
  - Purpose: Ensure that all the key UI elements of the YourActivityFragment are present when the fragment is displayed.
  - Process:

  Step 1: The test first verifies that the customized toolbar displays the correct fragment title, "Your Activity". It checks that the toolbar contains the correct title text and that it is displayed correctly.
  Step 2: The test checks for the presence of the "Day-Night" toggle button (day_night_toggle). It verifies that this button is visible in the toolbar and is correctly rendered.
  Step 3: It checks for the "Filter by Date" button. It ensures that the button is present and correctly displayed in the fragment.
  Step 4: It checks for the "Add Activity" button to ensure it exists and is visible.
  Step 5: Lastly, the test checks that the RecyclerView (which lists exercise activities) is present and visible.

- **UIElementExist Test:**
  - Purpose: Ensure that when the user clicks the "Filter by Date" button, the DatePickerDialog is opened, allowing the user to select a date for filtering activities.

    - Process: The test identifies the "Filter by Date" button and performs a click action on it. After the click action, the test checks if the DatePicker widget is displayed on the screen by verifying that an android.widget.DatePicker class is visible.

- **AddActivityButtonOpenAddActivityFragment Test:**
    - Purpose: This test verifies that clicking the "Add Activity" button successfully navigates the user to the AddActivityFragment, where they can add a new exercise activity.
    - Process: The test The test identifies the "Add Activity" button and performs a click action on it. It checks the toolbar title, ensuring that the new fragment displays the correct title, "Add Activity", verifies that the fragment has successfully navigated to the AddActivityFragment.

2. AddActivityTest.kt

- **Purpose:** This test class verifies the correct functionality of the 'Add Activity' feature in the app, ensuring user can add an exercise.
- **UIElementExist Test:**
    - Purpose: Ensures all relevant UI elements exist in the "Add Activity" fragment.
    - Process: It first clicks the "Add Activity" button to navigate to the fragment and then checks for the visibility of multiple UI components such as date inputs, activity name, duration input, and the submit and cancel buttons.
- **AddExerciseActivity Test:**
    - Purpose: Verifies that an exercise activity can be successfully added.
    - Process: The test fills in the date, time, name, and duration fields. It then clicks the "Submit" button and checks whether the new activity appears in the RecyclerView (in the "Your Activity" fragment). It ensures that the correct name, and datetime are displayed.

3. UpdateActivityTest.kt

- **Purpose:** This test class verifies the 'Update Activity' functionality, ensuring users can update an existing exercise.
- **updateActivityTestAll Test:**
    - Purpose: Ensures that an existing activity can be updated.
    - Process:

      Part 1: Hardcode an exercise item by navigating to the “Add Activity” fragment and fill in the EditText holders. Submit and click on “Update” button to navigate to “Update Activity” fragment.

      Part 2: Test UI elements exist, and the information provided matches item pre-created.

      Part 3: Modify and update new details for the updated item. Submit new details.

      Part 4: Check if updated details matches (activityTime, activityName and duration must be matched).

All implemented test cases pass. Each of these test cases is designed to test specific UI functionality within the app, verifying that users can interact with activities correctly, and that the expected UI behavior occurs under different scenarios.

G. Logs

Log messages are recorded to enable better efficient debugging and allow developer to keep track on the app process, allowing future improvements and integrations to develop the app.

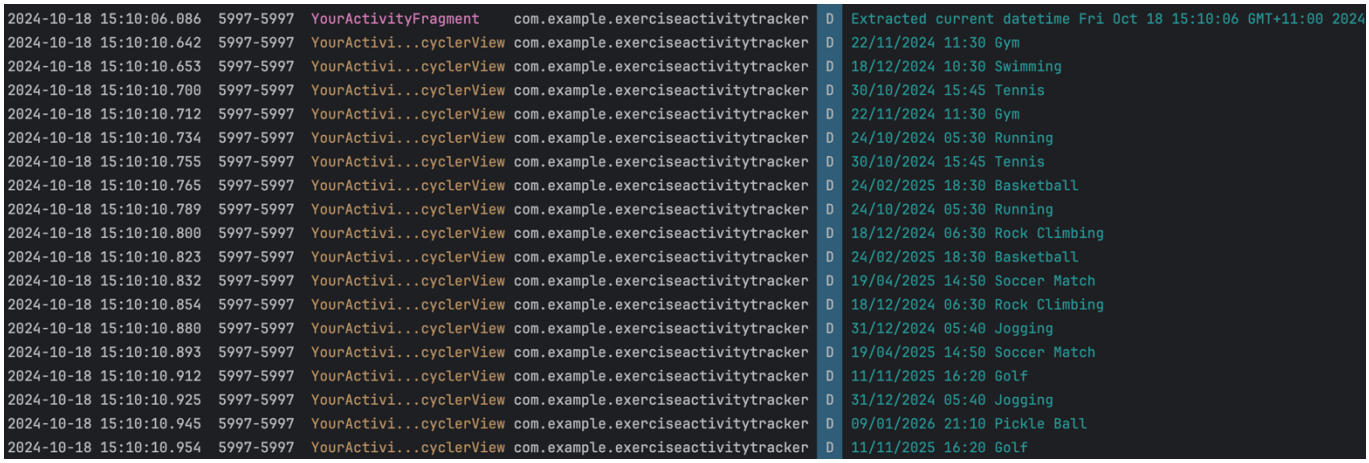

Figure shows logs initially when user start the app, logging the current time data and the set of exercise items (with datetime and name) obtained from the database to ensure data shown at YourActivityFragment initially is correct.

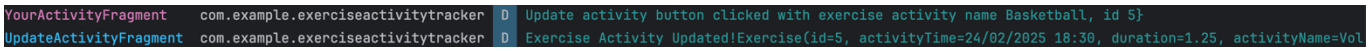

Figure shows logs when updating new item details, ensuring updated data matches and the id is corresponding.

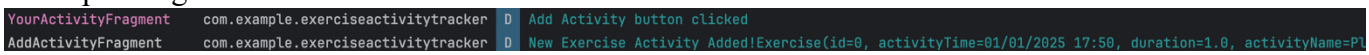

Figure shows logs when adding new item details, ensuring new data matches, notice that id changing is not made during this stage, since id auto-incrementation hasn't been called.

H. Quality Assurance (QA) Strategy:

The **Exercise Activity Tracker** app has undergone thorough testing and validation to ensure its quality and reliability. The QA process covered both functional and non-functional aspects to provide a smooth user experience. Below is an overview of the key QA processes:

1. Functional Testing:

Functional testing was performed to ensure that all user interactions and features of the app worked as expected. This included the following:

- **UI Testing (Espresso Tests):**
  - **YourActivityTest.kt:** Ensures that all key UI elements in the "Your Activity" screen (such as buttons, RecyclerView, and toolbar) are visible and functioning. It also verifies navigation between fragments and the functionality of features such as filtering by date.
  - **AddActivityTest.kt:** Tests the "Add Activity" screen, validating that all input fields are present and working, including form validation and the ability to submit a new activity.
  - **UpdateActivityTest.kt:** Ensures that users can successfully update an existing exercise. It checks that all input fields are pre-populated and validates the correctness of the updated data.
- **Use Case Testing:**
  - **Add an Activity:** Verified through user stories and direct testing that the user can add activities with proper input validation.
  - **Update an Activity:** The ability to update an existing activity was tested, ensuring that the changes are saved correctly in the database.

2. Non-functional Testing:

- **Performance Testing:** The performance of the RecyclerView in dynamically displaying exercise activities was tested with different data loads. The app remained responsive even with a significant number of exercises.

- **Data Persistence Testing:** Verified that data persists across app sessions using the Room database. Data entered by the user is saved, even after the app is closed and reopened.
- **Error Handling:** Tested various edge cases, such as invalid date formats and missing data. Appropriate error messages were displayed to guide the user in correcting their inputs.

3. Theming and UI Consistency:

- The app's light and dark themes were tested to ensure that all UI components, such as buttons and text, were visible and readable in both modes.
- Theme toggling was also validated for a smooth transition between light and dark themes.
- Deletion animation and UNDO option on SnackBar widget was tested and ensure visual impact is efficiently delivered to the user.

4. Bug Tracking:

All identified bugs were logged and addressed throughout the development process. Common issues such as incorrect form validation, misaligned UI elements, or incorrect data persistence were systematically fixed.

I. Challenges and Improvements

- **Challenges**:
    - **Fragment Navigation and Data Handling:** Managing navigation between the various fragments (YourActivityFragment, AddActivityFragment, UpdateActivityFragment), particularly in ensuring data consistency across the app. The app's reliance on shared ViewModel instances made it critical to ensure that the data flowing between fragments was well-managed and correctly observed.
    - **RecyclerView with LiveData:** Implementing RecyclerView to display dynamic data from the Room database involved challenges in synchronizing the UI with the ViewModel and LiveData. Ensuring the correct display of activities and handling the CRUD operations required coordination between the data updates and UI rendering. There were issues initially in handling data binding and updating RecyclerView after LiveData changes, requiring additional debugging efforts to ensure smooth transitions and real-time data reflection.
    - **Testing Challenges – UI Testing with Espresso:** Testing the app using Espresso involved several challenges, especially around testing dynamic data in RecyclerView and fragment navigation. Initially attempts tried using more sophisticated testing approaches, such as **Mockito** for mocking, **Data Binding** to tie UI elements to data models, and attempting to scale down Android utility versions to ensure compatibility with different dependencies. However, these approaches proved difficult due to integration issues and limited success in properly targeting the fragments for UI testing. Hence, switched to “Record Espresso Test”, where test cases were recorded for capturing interactions with the UI. This manual recording approach ensured compatibility and allowed direct testing of user interaction in the targeted fragments, but it required more manual effort and lacked the flexibility of automated unit testing.
    - **Edge Case Validation:** Ensuring that the app handled various edge cases, such as invalid data inputs (empty fields, incorrect datetime, name formats), required additional attention. Validating data at the time of CRUD operations and providing appropriate feedback to users was a critical part of ensuring a robust user experience. Handling these edge cases while maintaining a user-friendly interface was a challenge that was addressed during development and testing.

- **Improvements**:
    - **Enhanced Error Handling:** While the app handles basic error cases from users’ mistakes (e.g., empty fields, invalid inputs), I implemented more informative error messages and

listed them subsequentially. The app can also be improved in the future by providing suggestive solution (if possible).

- **UI and UX Enhancements:** UI measurements could benefit from additional visual enhancements, such as smooth animations and transitions between fragments to make navigation feel more seamless.
- **More Advanced Filtering and Sorting Options:** In the current implementation, users can only filter activities by date in the YourActivityFragment. Potentially, adding more sophisticated filters, such as filtering by activity type, duration, time range, or searching name would provide users with greater flexibility in viewing their activities, making the app more versatile for users with more complex tracking needs.

J. Reflection on Assignment 2

Architectural Evolution: Reflecting on Assignment 2 "Rent With Intent" app, we can observe a noticeable evolution in design patterns and technical architecture from Assignment 2 to Assignment 3 "Exercise Activity Tracker" app. The most significant change in this transition was moving from a two-activity structure with Parcelable intents for data transfer to a multi-fragment architecture with Room, ViewModel and LiveData integration. This shift was motivated by the need for better scalability, modularity, and separation of utilities.

Database and Data Handling: In Assignment 2, the app relied on in-memory data structures and manual data passing between activities using Parcelable objects, using the **MVI** pattern. For Assignment 3, I transitioned to a more sophisticated and persistent solution using the **Room database** for data storage. This choice was driven by the need for **persistent data storage** that could handle CRUD operations (Create, Read, Update, Delete) efficiently across multiple fragments.

The **Room** database's integration with **LiveData** and **ViewModel** has vastly improved data handling. Now, the UI components observe data changes directly from the database, ensuring that updates (such as adding a new exercise or modifying existing data) are immediately reflected in the app's views without manual intervention.

Error Handling and User Feedback: In Assignment 2, error handling was largely limited to Toast messages and user feedback during data entry errors. Assignment 3 improves upon this by incorporating better **validation mechanisms**, particularly for inputs like name, date and time in exercise activities. The introduction of **real-time error messages** during data entry has enhanced the overall user experience, ensuring that users receive immediate feedback for invalid entries, which aligns with current UI/UX practices.

Adoption of Fragment Navigation and Testing: While Assignment 2 primarily used Intents to pass data between activities, Assignment 3 adopts the **Navigation Component**, which simplifies navigation between fragments and ensures more cohesive handling of navigation actions and argument passing (using safe-args). This approach is much more aligned with modern Android development practices and provides a more flexible way to manage complex UI flows, such as navigating between the "Your Activity," "Add Activity," and "Update Activity" fragments.

In terms of testing, Assignment 2 relied heavily on manual testing, which proved to be insufficient as the app grew in complexity. For Assignment 3, I experimented with various UI testing frameworks such as **Mockito** and **Data Binding**, but due to compatibility issues with Android utility versions, I eventually resorted to **Espresso Record Test** for automated UI testing. This traditional yet reliable approach allowed me to test user interactions with the UI components of each fragment (e.g., checking the existence of UI elements, validating navigation between fragments, and ensuring that CRUD operations work correctly).

Architectural Diagrams:

- **Assignment 2**: The architecture was a simple **two-activity structure**[17] where data flow was managed using Intents and Parcelable objects.

- **Assignment 3**: The architecture follows a more sophisticated **MVVM (Model-View-ViewModel)** design pattern[16], incorporating fragments, ViewModel, and LiveData. The Room database serves as the data repository, and all data flow is managed through reactive streams (LiveData), ensuring that the UI stays updated with the latest changes in the data layer.

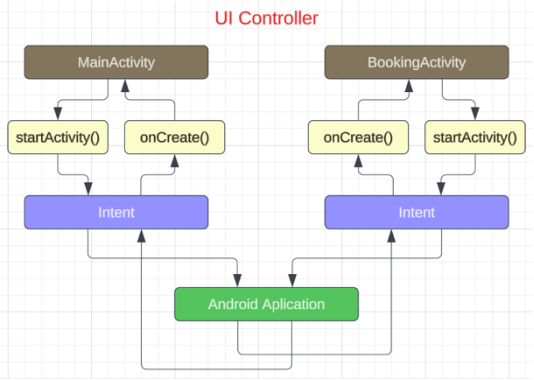

Figure. Assignment 2 Architecture UML Diagram.

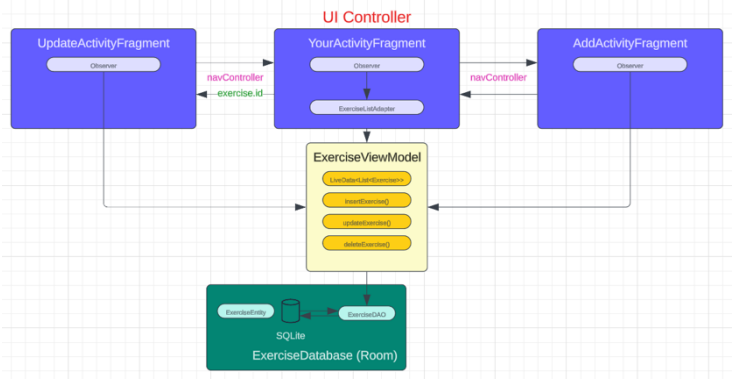

Figure. Assignment 3 Architecture UML Diagram.

Conclusion: Assignment 2 laid the groundwork for building a functional Android app, focusing on effective UI/UX design with basic functionality. However, Assignment 3 demonstrates a significant leap forward, leveraging more advanced Android components and architectures, particularly with the integration of **Room**, **ViewModel**, and **LiveData**. This transition ensures better scalability, maintainability, and a more responsive and user-friendly app.

K. Investigation

1. Setup

1. **Initial Setup:**
    - Set up 2 variants with the same dataset (pre-recorded RecyclerView item set) and identical list interactions (CRUD).
    - Variant 1 is the original app "ExerciseActivityTracker", while Variant 2 "ExerciseActivityTrackerVariant" will apply some changes to accommodate manual list operations[19], include modifying ViewModel, DAO, create Repository[20] and apply minor changes to fragment controllers, some methods are removed from the original app-program since they are not tested (e.g., date filter function is removed, animation is simpler/incomplete, and adjust the app logics with separated helper methods etc).
    - Each test case consist of:

- Launch Your Activity fragment
- Navigate to Add Activity fragment
- Add an exercise and navigate back to Your Activity (each test case having the same input)
- Navigate to Update Activity fragment (exercise id will be passed with SafeArgs[8])
- Update an exercise and navigate back to Your Activity (each test case having the same input)
- Delete that exercise (with animation)
    - Use Android Profiler to monitor and record CPU and memory usage during list operations.

2. **Scenario 1: Loading Data:**
    - Measure CPU and memory performance when loading the list of exercise activities.
3. **Scenario 2: Adding New Data:**
    - Track performance when adding 4 new entries into the list.
4. **Scenario 3: Updating Existing Data:**
    - Update 4 entries in the list and measure CPU spikes and memory usage.
5. **Scenario 4: Deleting Data:**
    - Measure the performance when deleting 4 entries from the list.

The app set up for Variant 2 can be found from "ExerciseActivityTrackerVariant" directory, where the key difference to the original program is that the data is not 'observable', or LiveData is not applied.

2. Observation from Original Program

Investigation Focus: Exploring CPU and Memory Performance Using Android Profiler[18] for "Exercise Activity Tracker" app.

CPU Usage Analysis

1. **Launch Your Activity:**
    - Test 1: 36%
    - Test 2: 31%
    - Test 3: 31%
    - Test 4: 34%

    Observation: CPU usage when launching Your Activity with a RecyclerView remains somewhat stable across tests, fluctuating between 31% and 36%. This indicates a moderate CPU demand due to loading and rendering dynamic data in the RecyclerView.

2. **Navigate from Your Activity to Add Activity:**
    - Test 1: 27%
    - Test 2: 31%
    - Test 3: 30%
    - Test 4: 27%

    Observation: Transitioning between activities appears to be lightweight, with CPU usage ranging from 27% to 31%. This suggests that navigation and fragment transitions are optimized, but can slightly vary depending on system performance or background processes.

3. **Add an Exercise Activity and Navigate Back to Your Activity:**
    - Test 1: 44%
    - Test 2: 24%
    - Test 3: 26%
    - Test 4: 31%

    Observation: Test 1 shows significantly higher CPU usage (44%) compared to the other tests (24-31%) during this step. This discrepancy might be due to a heavier load or data processing during Test 1. Generally, the activity creation and navigation process appears efficient in the other tests, with a relatively stable CPU range.

4. **Navigate from Your Activity to Update Activity (with Passed ID):**
    - Test 1: 33%
    - Test 2: 31%
    - Test 3: 35%
    - Test 4: 35%

    Observation: Navigating to the Update Activity incurs moderate CPU usage (31-35%), showing consistency across all tests. The ViewModel efficiently retrieves data with a low overhead, ensuring the performance remains stable during updates.

5. **Update an Exercise Activity and Navigate Back to Your Activity:**
    - Test 1: 34%
    - Test 2: 39%
    - Test 3: 25%
    - Test 4: 35%

    Observation: CPU usage shows some fluctuations when updating an activity, ranging from 25% to 39%. Test 2 showed slightly higher CPU usage (39%), possibly due to background processes or data synchronization loads at the time of testing. Generally, navigating from Your Activity to Update Activity would cost more CPU usage than to Add Activity fragment, this is because the app-system have to pass the exercise id to Update Activity using SafeArgs[8].

6. **Delete an Exercise Activity (with Animations):**
    - Test 1: 52%
    - Test 2: 28%
    - Test 3: 27%

- Test 4: 27%

Observation: Test 1 shows a significantly higher CPU usage (52%) during deletion compared to the other tests (27-28%). The deletion process includes animations, which might cause higher CPU spikes. In Test 1, the deletion, along with the animation, might have caused a CPU surge. Other tests indicate that the operation generally incurs low CPU usage.

Memory Usage Analysis

1. **Initial Memory Usage:**
   - Test 1: 86 MB
   - Test 2: 94.8 MB
   - Test 3: 105.9 MB
   - Test 4: 114.4 MB

   Observation: The initial memory usage increases from Test 1 to Test 4, beginning at 86 MB in Test 1 and reaching 114.4 MB by Test 4. This increase is a natural consequence of the app being tested continuously without resetting between tests. Since the app retains certain cached data and states across tests, the memory accumulates slightly between each iteration. The key focus should be on how the memory usage changes with specific actions rather than overall app memory consumption.

2. **When Adding an Exercise Item:**
   - Test 1: 87.5 MB (+1.5 MB)
   - Test 2: 100.5 MB (+5.7 MB)
   - Test 3: 107.7 MB (+1.8 MB)
   - Test 4: 115.8 MB (+1.4 MB)

   Observation: Memory usage increases when an exercise item is added. The difference between the initial memory and after adding an item is mostly consistent across tests. The increase is higher in Test 2, which could be due to background processes or larger dataset management during that test.

3. **When Updating an Exercise Item:**
   - Test 1: 92.9 MB (+5.4 MB)
   - Test 2: 104.5 MB (+4 MB)
   - Test 3: 113.3 MB (+5.6 MB)
   - Test 4: 120.5 MB (+4.7 MB)

   Observation: Memory usage jumps more significantly when updating an exercise item, which could be due to the ViewModel's involvement in fetching and processing data, combined with UI changes. The overall increase across the tests shows a steady pattern.

4. **When Deleting an Exercise Item:**
   - Test 1: 100 MB (+7.1 MB)
   - Test 2: 105.4 MB (+ 0.9 MB)
   - Test 3: 111 MB (+2.2 MB)
   - Test 4: 121.1 MB (+0.6 MB)

   Observation: Memory usage increases even during deletion, which is counterintuitive. Typically, deleting items should free memory. However, animations during deletion (e.g., RecyclerView item red-over and fade-out) might explain this behaviour.

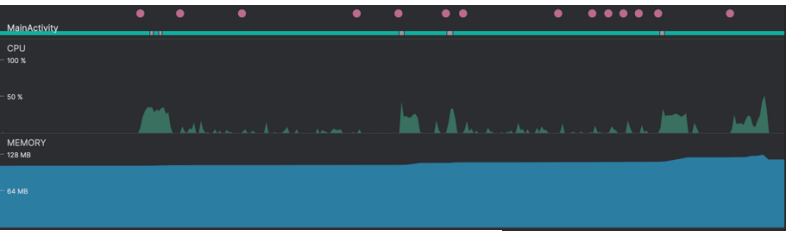

Figure. Test 1 CPU and Memory Profiler Record

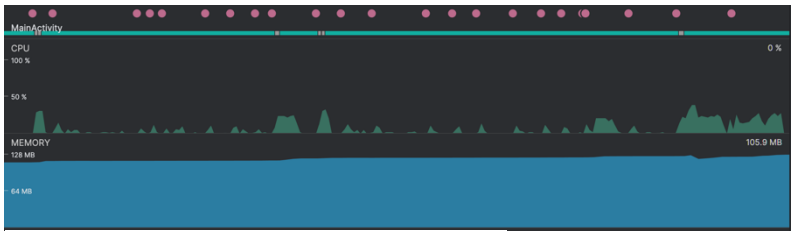

Figure. Test 2 CPU and Memory Profiler Record

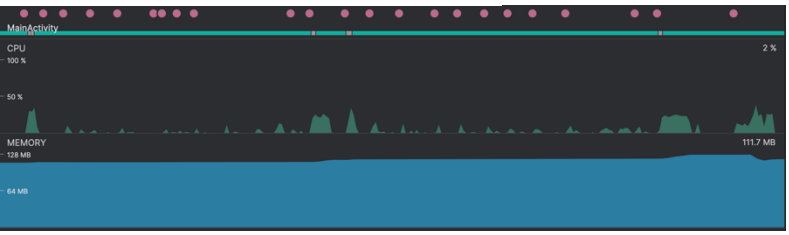

Figure. Test 3 CPU and Memory Profiler Record

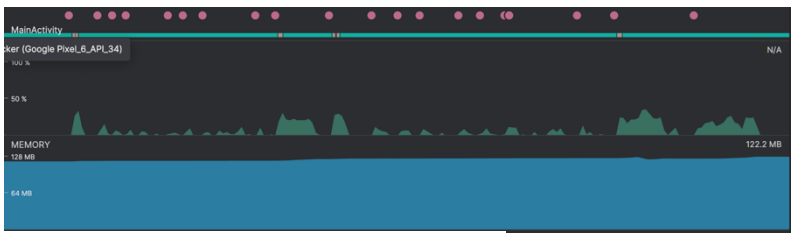

Figure. Test 4 CPU and Memory Profiler Record

Conclusion

CPU Usage:

- The CPU usage results indicate that launching and navigating through activities are fairly consistent, with most operations staying within a 25-36% range, except for some spikes when adding, updating, or deleting items, especially in Test 1.
- The largest CPU spike (52%) was seen during deletion in Test 1, which may be due to the animation involved. In contrast, other tests kept deletion CPU usage within a lower range (27-28%).

Memory Usage:

- Memory usage showed a consistent increase across tests, when adding, updating, or deleting items. While the initial memory usage started at 86 MB in Test 1, it gradually increased across tests, reaching 114.4 MB in Test 4. The increase is due to retained states, caching, and background processes, since the app was tested continuously without resetting.

3. Variant Testing

Investigation Focus: Exploring CPU and Memory Performance Using Android Profiler[18] for Two Variants of Data Handling in RecyclerView (LiveData vs Manual List Updates)

This investigation aim to explore the CPU and memory performance differences between two data-handling approaches within the app's RecyclerView component for listing exercise activities. We hypothesize that using **LiveData** accommodating with **ViewModel** will automatically manage data will offer better CPU and memory efficiency[21] compared to manually updating the list in the RecyclerView.

Variants:

1. **Variant 1: LiveData with ViewModel**
    - The current app design leverages **LiveData** for observing and updating the list in RecyclerView.
    - The **ViewModel** ensures the data is retained during configuration changes and simplifies data management by observing changes in the Room database.
    - We will track CPU and memory usage when loading, adding, updating, and deleting exercise entries.
2. **Variant 2: Manual List Updates**
    - The second variant removes **LiveData** from the RecyclerView and instead uses manual list management for the same operations.
    - The list of exercises will be updated directly, without observing changes through **LiveData**.
    - We will repeat the same user actions (viewing, adding, updating, deleting) and track CPU and memory usage in this variant.

Expected Results:

- CPU Usage: We expect Variant 1 (**LiveData**) to consume fewer CPU resources[21] due to more efficient data handling.
- Memory Usage: **LiveData** will manage memory better because it automatically updates and retains the data, whereas manual updates may cause inefficient memory usage, especially during large updates or continuous scrolling in RecyclerView.

Results of Variant 2:

+ CPU Usage Analysis

1. **Launch Your Activity:**
   - Test 1: 28%
   - Test 2: 29%
   - Test 3: 40%
   - Test 4: 21%
2. **Navigate from Your Activity to Add Activity:**
   - Test 1: 28%
   - Test 2: 29%
   - Test 3: 40%
   - Test 4: 24%
3. **Add an Exercise Activity and Navigate Back to Your Activity:**
   - Test 1: 28%
   - Test 2: 22%
   - Test 3: 44%
   - Test 4: 30%
4. **Navigate from Your Activity to Update Activity (with Passed ID):**
   - Test 1: 26%
   - Test 2: 28%
   - Test 3: 29%
   - Test 4: 25%
5. **Update an Exercise Activity and Navigate Back to Your Activity:**
   - Test 1: 26%
   - Test 2: 25%
   - Test 3: 34%
   - Test 4: 27%
6. **Delete an Exercise Activity (with Animations):**
   - Test 1: 25%
   - Test 2: 26%
   - Test 3: 26%
   - Test 4: 19%

+ Memory Usage Analysis

1. **Initial Memory Usage:**
   - Test 1: 100 MB
   - Test 2: 114.9 MB
   - Test 3: 125.4 MB
   - Test 4: 129.8 MB
2. **When Adding an Exercise Item:**
   - Test 1: 101.6MB (+1.6 MB)
   - Test 2: 117 MB (+2.1 MB)
   - Test 3: 127.1 MB (+1.7 MB)
   - Test 4: 132.5 MB (+2.7 MB)
3. **When Updating an Exercise Item:**
   - Test 1: 102.3 MB (+0.7 MB)
   - Test 2: 120.3 MB (+3.3 MB)
   - Test 3: 127.6 MB (+0.5MB)
   - Test 4: 139 MB (+6.5 MB)

4. **When Deleting an Exercise Item:**
    - Test 1: 105.7 MB (+3.4 MB)
    - Test 2: 123.5 MB (+ 3.2MB)
    - Test 3: 129.6 MB (+2.0 MB)
    - Test 4: 140.4 MB (+1.4 MB)

The usage results indicate that launching and navigating through activities for Variant 2 are less consistent than for Variant 1 (using LiveData), with more gaps and fluctuations occur, which make it less predictable.

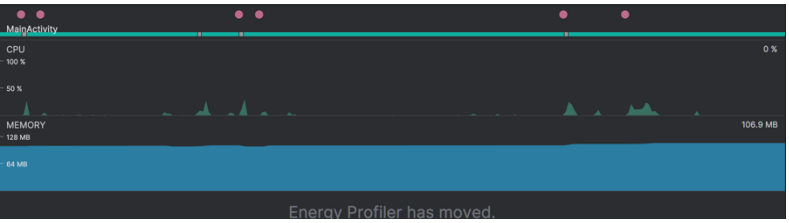

Figure. Test 1 CPU and Memory Profiler Record

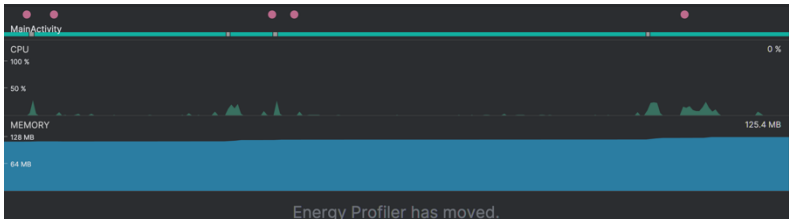

Figure. Test 2 CPU and Memory Profiler Record

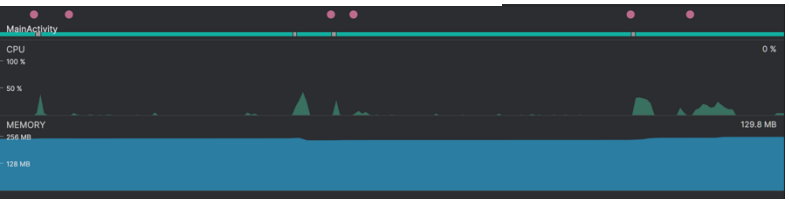

Figure. Test 3 CPU and Memory Profiler Record

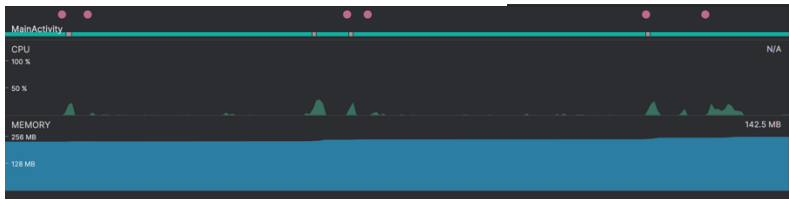

Figure. Test 4 CPU and Memory Profiler Record

Comparison of Average Test Case Results between Variant 1 and Variant 2:

1. **Variant 1:**
    - Launch Your Activity: 33.00%
    - Navigate from Your Activity to Add Activity: 28.75%
    - Add an Exercise Activity and Navigate Back to Your Activity: 31.25%
    - Navigate from Your Activity to Update Activity: 33.50%
    - Update an Exercise Activity and Navigate Back to Your Activity: 33.25%
    - Delete an Exercise Activity: 33.5%
    - Memory Usage when Adding Exercise: +2.60 MB
    - Memory Usage when Updating Exercise: +4.93 MB
    - Memory Usage when Deleting Exercise: +2.70 MB
2. **Variant 2:**
    - Launch Your Activity: 29.50%
    - Navigate from Your Activity to Add Activity: 30.25%
    - Add an Exercise Activity and Navigate Back to Your Activity: 31.00%
    - Navigate from Your Activity to Update Activity: 27.00%
    - Update an Exercise Activity and Navigate Back to Your Activity: 28.00%
    - Delete an Exercise Activity: 24.00%
    - Memory Usage when Adding Exercise: +2.03 MB
    - Memory Usage when Updating Exercise: +2.75 MB
    - Memory Usage when Deleting Exercise: +2.50 MB

Relection:

The investigation compared the performance of two variants of the "Exercise Activity Tracker" app, Variant 1 (using LiveData) and Variant 2 (manual data management), focusing on CPU and memory usage during common operations like adding, updating, and deleting exercises.

1. **CPU Usage:**
    - Variant 1 (LiveData) exhibited more consistent and stable CPU usage, with most operations ranging between 28% and 36%. LiveData's ability to automatically manage data helped reduce fluctuations in performance.
    - Variant 2 (Manual Updates) showed more variation in CPU usage, with operations ranging from 19% to 44%. Deletion operations used less CPU due to simpler animations, but other actions like updating exercises had more unpredictable performance.

2. **Memory Usage:**
    - Variant 1 had slightly higher memory usage, likely due to LiveData retaining observability features, but it handled memory more predictably.
    - Variant 2 showed lower memory usage increases, particularly when adding and updating data, but it experienced more memory spikes in certain tests.
3. **Animation Impact:**
    - Deletion animations in Variant 1 caused higher CPU spikes, while Variant 2 used less CPU due to simpler animations (from ExerciseAdapter.kt) during deletion.
4. **Test Case Limitation:**
    - The small number of test cases (4) might limit the accuracy of the comparison, especially for larger datasets or more complex scenarios.

Conclusion: LiveData (Variant 1) provided more predictable performance with stable CPU and memory usage, while manual list management (Variant 2) had less consistency but performed better in certain cases like deletion.

L. Acknowledgement

1. Android Developers. (2023). Room with a view: Kotlin. https://developer.android.com/codelabs/android-room-with-a-view-kotlin#0
2. Android Developers. (2024). LiveData. https://developer.android.com/topic/libraries/architecture/livedata
3. Android Developers. (n.d.). RecyclerView. https://developer.android.com/develop/ui/views/layout/recyclerview
4. Stack Overflow. (2021). Switch button for changing theme in Android. https://stackoverflow.com/questions/68881424/switch-button-for-change-theme-in-android
5. Pathak, A. (2023). Implementing light to dark and vice versa theme switch in Android. Medium. https://medium.com/@myofficework000/implementing-light-to-dark-and-vice-versa-theme-switch-in-android-16f2916b761f
6. GeeksforGeeks. (2024). Fragment lifecycle in Android. https://www.geeksforgeeks.org/fragment-lifecycle-in-android/
7. Medium. (2023). Implementing navigation in your Android app with Android Navigation Component. https://medium.com/@muhammadumarch321/implementing-navigation-in-your-android-app-with-android-navigation-component-ff22a3d300a
8. Chet Haase. (2020). Navigating with SafeArgs. Medium. https://medium.com/androiddevelopers/navigating-with-safeargs-bf26c17b1269
9. Muhammadumarch. (2023). Quick and easy: A basic guide to implementing DatePicker in Android. Medium. https://medium.com/@mrizqi070502/quick-and-easy-a-basic-guide-to-implementing-date-picker-in-android-114b36394953
10. Rizqi, M. (2021). How to create custom AppBar/ActionBar/Toolbar in Android Studio Java. Medium. https://medium.com/swlh/how-to-create-custom-appbar-actionbar-toolbar-in-android-studio-java-61907fa1e44
11. GeeksforGeeks. (2021). Date and time formatting in Android. https://www.geeksforgeeks.org/date-and-time-formatting-in-android/
12. Baumgartner, L. (2016). How to notifyDataSetChanged. Treehouse. https://teamtreehouse.com/community/how-to-notifydatasetchanged
13. Gonzalez, N. (2023). Getting started with Kotlin coroutines in Android. Medium. https://nickand.medium.com/getting-started-with-kotlin-coroutines-in-android-bfa8283fcf60
14. Stack Overflow. (2024). Change view color with alpha animation. https://stackoverflow.com/questions/27780973/change-view-color-with-alpha-animation
15. GeeksforGeeks. (2021). Android fade in/out in Kotlin. https://www.geeksforgeeks.org/android-fade-in-out-in-kotlin/
16. Burman, U. (2018). Android Room persistence library: A login example with LiveData. Medium. https://medium.com/@umang.burman.micro/android-room-persistence-library-a-login-example-with-livedata-b6019fe23b0
17. Android Developers. (n.d.). Intents and filters. https://developer.android.com/guide/components/intents-filters
18. Gelormini, D. (2022). Practical Android profiling. Medium. https://medium.com/ww-tech-blog/practical-android-profiling-58ece24934f7
19. Stack Overflow. (2020). Can you use room and not use a LiveData object? https://stackoverflow.com/questions/62476766/can-you-use-room-and-not-use-a-livedata-object
20. MuindiStephen. (2023). Why you should use Flows instead of LiveData in Room Database? Medium. https://medium.com/@stephenmuindi241/why-you-should-use-flows-instead-of-livedata-in-room-database-77aa96d26873
21. Anand, G. (2023) LiveData In Android. Medium. https://medium.com/@anandgaur22/livedata-in-android-eecbbed63c51
22. Gen AI: DALL-E was used to create the app's logo, using the prompt: Create a logo for my app - Exercise Activity Tracker, which is a app for user to CRUD their plannings for exercise activities, the logo should be able to deliver high atheism spirit with multiple exercises, app logo should also be vivid with major tones of dark blue, dark purple and deep red colors to be applied. The logo shouldn't be too complex with too many details to deliver direct impression to its user.