COMP1786

Mobile Application Design and Development

Student's Name: Vu Thanh Nguyen
Student's ID: 001343200
Submission Date: 04/12/2025

Table of contents

Table of Figures

1. Brief statement of features you have implemented

Feature	Status	Your Comments
Functionality A	Fully completed ☒ Partially completed ☐ Having bugs/Not working ☐ Not implemented ☐	
Functionality B	Fully completed ☒ Partially completed ☐ Having bugs/Not working ☐ Not implemented ☐	
Functionality C	Fully completed ☒ Partially completed ☐ Having bugs/Not working ☐ Not implemented ☐	
Functionality D	Fully completed ☒ Partially completed ☐ Having bugs/Not working ☐ Not implemented ☐	
Functionality E	Fully completed ☒ Partially completed ☐ Having bugs/Not working ☐ Not implemented ☐	
Functionality F	Fully completed ☒ Partially completed ☐ Having bugs/Not working ☐ Not implemented ☐	
Functionality G	Fully completed ☐ Partially completed ☒ Having bugs/Not working ☐ Not implemented ☐	

Link to recorded video (if you record your application before submitting the report) The recorded video will demonstrate the implemented product. It is roughly 15 minutes. Enter the link into the below box
https://drive.google.com/drive/u/0/folders/1TsGf7-CtqCiGaQGBu7MTGclS4MLbIEuc

2. SCREEN SHOTS/DESIGN

In this section, the application is designed with the main color tones of blue and white, with convenient button arrangement for new and future users.

A. JavaCode

1. activity_main.xml (menu screen)

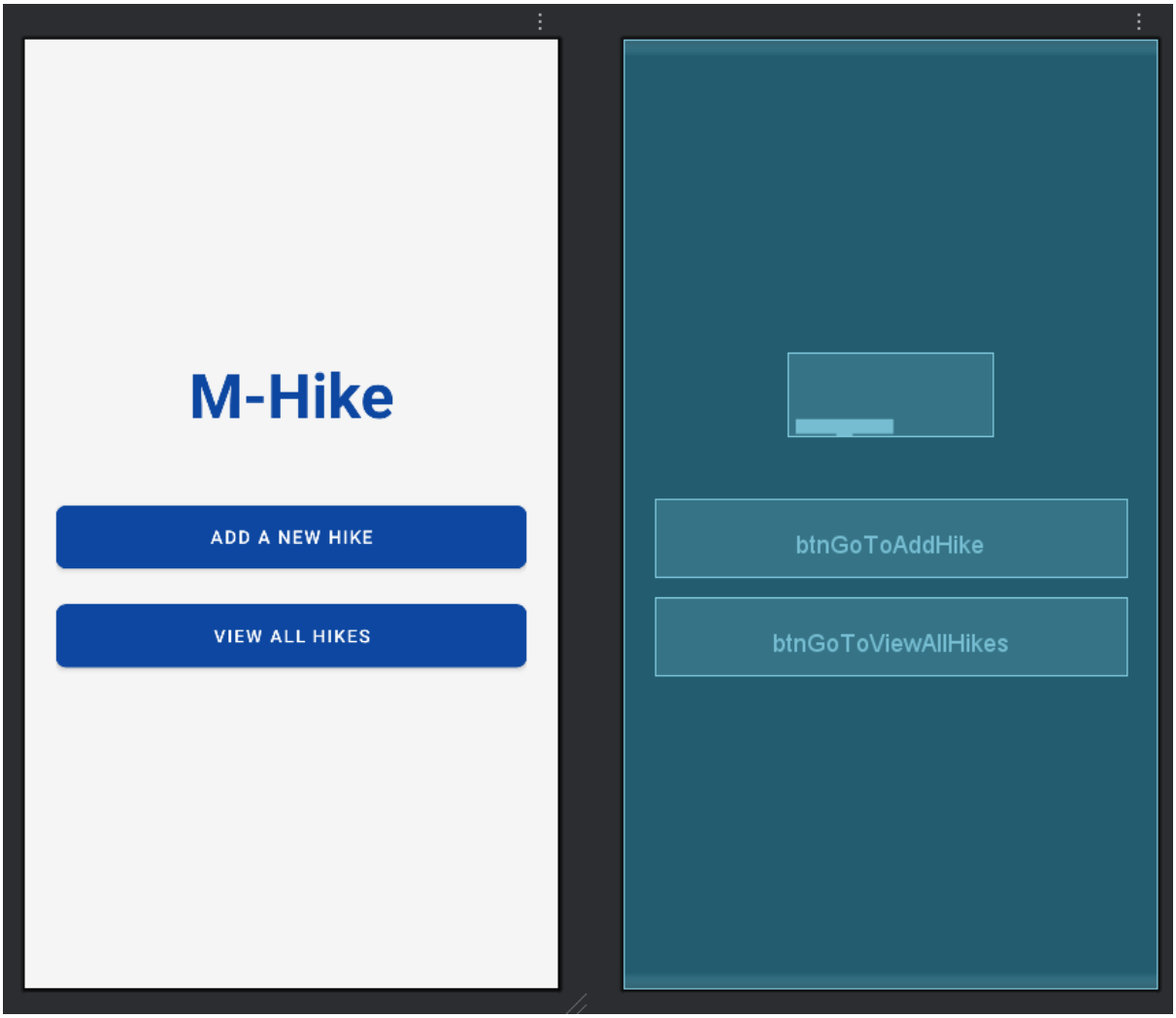

Figure 1Design of menu screen

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="24dp"
    android:background="@color/windowBackground"
    tools:context=".MainActivity">
```

```
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="M-Hike"
        android:textSize="48sp"
        android:textStyle="bold"
        android:textColor="@color/colorPrimary"
        android:layout_marginBottom="48dp"/>

    <com.google.android.material.button.MaterialButton
        android:id="@+id/btnGoToAddHike"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="ADD A NEW HIKE"
        app:cornerRadius="8dp"
        android:minHeight="60dp"
        android:layout_marginBottom="16dp"/>

    <com.google.android.material.button.MaterialButton
        android:id="@+id/btnGoToViewAllHikes"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:cornerRadius="8dp"
        android:text="VIEW ALL HIKES"
        android:minHeight="60dp"/>

</LinearLayout>
```

This layout uses a LinearLayout as its main container, with its child elements arranged vertically. The android:gravity="center" attribute ensures that all the elements within it are centered on the screen.At the top, a TextView displays the title "M-Hike" in a large, bold font for emphasis. Below the title are two MaterialButtons: the first with the text "ADD A NEW HIKE" and the second with "VIEW ALL HIKES".Both buttons are styled consistently, spanning the full width of the screen and featuring rounded corners. Margins (marginBottom) are used to create appropriate separation between the title and the buttons, while the padding attribute on the LinearLayout creates an empty border around the entire interface.

2. activity_add_hike.xml

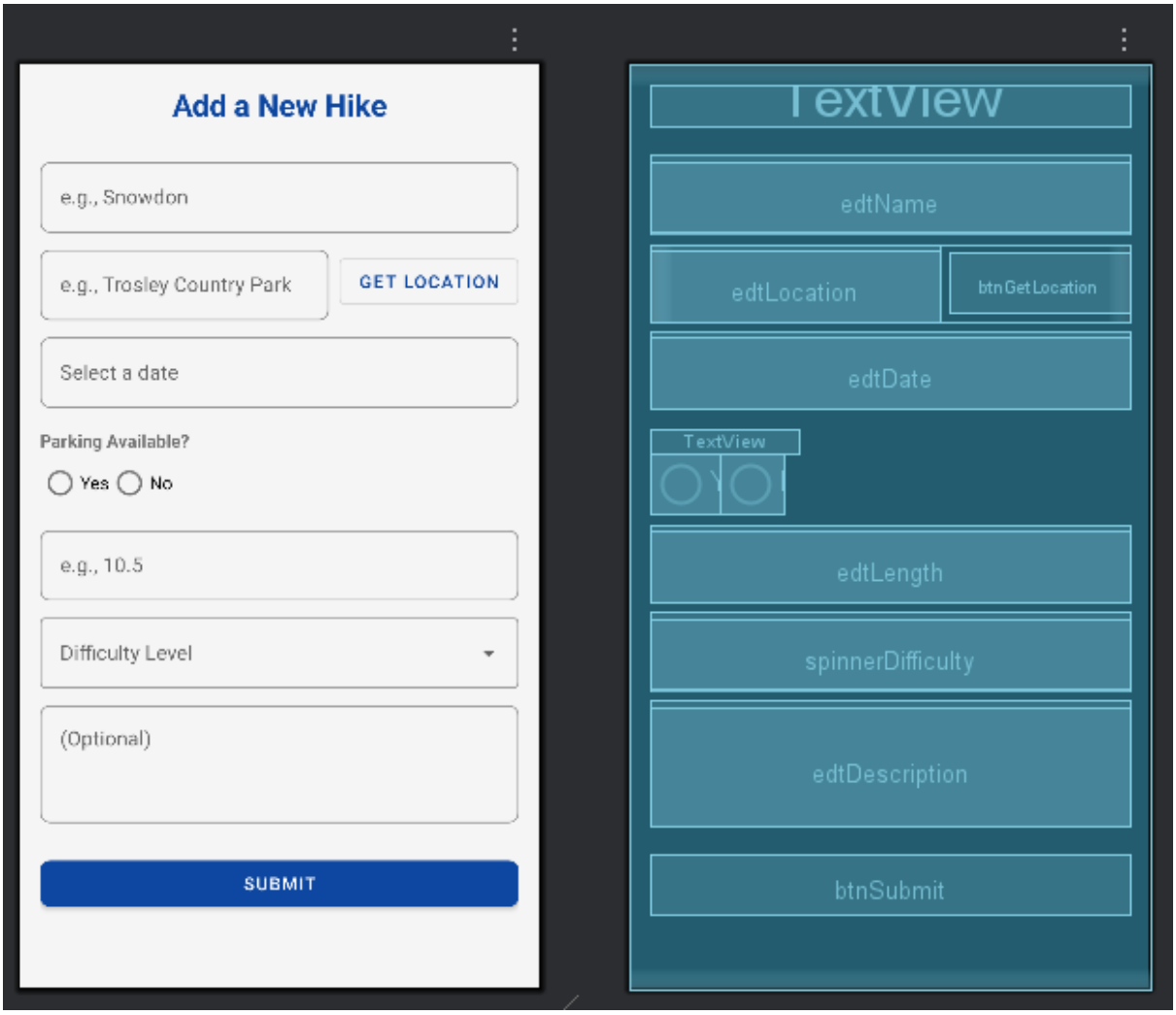

Figure 2 Form add hike screen

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fillViewport="true"
    android:background="@color/windowBackground">

    <LinearLayout
        android:orientation="vertical"
        android:padding="16dp"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <TextView
            android:id="@+id/txtFormTitle"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/form_title_add_hike"
            android:textSize="24sp"
            android:textStyle="bold"
            android:gravity="center"
            android:textColor="?attr/colorPrimary"
```

```
    android:layout_marginBottom="24dp" />

<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/name_layout"
    style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    app:boxCornerRadiusTopStart="8dp"
    app:boxCornerRadiusTopEnd="8dp"
    app:boxCornerRadiusBottomStart="8dp"
    app:boxCornerRadiusBottomEnd="8dp">
    <com.google.android.material.textfield.TextInputEditText
        android:id="@+id/edtName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/hint_hike_name"/>
</com.google.android.material.textfield.TextInputLayout>

<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:layout_marginBottom="8dp"
    android:gravity="center_vertical">
    <com.google.android.material.textfield.TextInputLayout
        android:id="@+id/location_layout"
        style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox"
        android:layout_width="0dp"
        android:layout_weight="1"
        android:layout_height="wrap_content"
        app:boxCornerRadiusTopStart="8dp"
        app:boxCornerRadiusTopEnd="8dp"
        app:boxCornerRadiusBottomStart="8dp"
        app:boxCornerRadiusBottomEnd="8dp">
        <com.google.android.material.textfield.TextInputEditText
            android:id="@+id/edtLocation"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="@string/hint_location"/>
    </com.google.android.material.textfield.TextInputLayout>
    <com.google.android.material.button.MaterialButton
        android:id="@+id/btnGetLocation"
        style="?attr/materialButtonOutlinedStyle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:text="Get Location"/>
</LinearLayout>

<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/date_layout"
    style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    app:boxCornerRadiusTopStart="8dp"
```

```
    app:boxCornerRadiusTopEnd="8dp"
    app:boxCornerRadiusBottomStart="8dp"
    app:boxCornerRadiusBottomEnd="8dp">
    <com.google.android.material.textfield.TextInputEditText
      android:id="@+id/edtDate"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:focusable="false"
      android:clickable="true"
      android:hint="@string/hint_hike_date"/>
  </com.google.android.material.textfield.TextInputLayout>

  <TextView
    android:text="@string/label_parking"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="8dp"
    android:textStyle="bold"/>
  <RadioGroup
    android:id="@+id/radioParking"
    android:orientation="horizontal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

    <RadioButton
      android:id="@+id/rdoYes"
      android:text="@string/yes"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>

    <RadioButton
      android:id="@+id/rdoNo"
      android:text="@string/no"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
  </RadioGroup>

  <com.google.android.material.textfield.TextInputLayout
    android:id="@+id/length_layout"
    style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="8dp"
    android:layout_marginBottom="8dp"
    app:boxCornerRadiusTopStart="8dp"
    app:boxCornerRadiusTopEnd="8dp"
    app:boxCornerRadiusBottomStart="8dp"
    app:boxCornerRadiusBottomEnd="8dp">
    <com.google.android.material.textfield.TextInputEditText
      android:id="@+id/edtLength"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:inputType="numberDecimal"
      android:hint="@string/hint_length"/>
  </com.google.android.material.textfield.TextInputLayout>

  <com.google.android.material.textfield.TextInputLayout
```

```
        style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox.ExposedDropdownMenu"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="8dp">
        <AutoCompleteTextView
            android:id="@+id/spinnerDifficulty"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:inputType="none"
            android:hint="@string/label_difficulty"/>
    </com.google.android.material.textfield.TextInputLayout>

    <com.google.android.material.textfield.TextInputLayout
        style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="8dp"
        app:boxCornerRadiusTopStart="8dp"
        app:boxCornerRadiusTopEnd="8dp"
        app:boxCornerRadiusBottomStart="8dp"
        app:boxCornerRadiusBottomEnd="8dp">
        <com.google.android.material.textfield.TextInputEditText
            android:id="@+id/edtDescription"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:gravity="top"
            android:inputType="textMultiLine"
            android:lines="3"
            android:hint="@string/hint_description"/>
    </com.google.android.material.textfield.TextInputLayout>

    <com.google.android.material.button.MaterialButton
        android:id="@+id/btnSubmit"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/btn_submit"
        app:cornerRadius="8dp"
        android:layout_marginTop="16dp"/>

  </LinearLayout>
</ScrollView>
```

This layout is designed as a scrollable form for adding a new hike, wrapped in a ScrollView to ensure it's accessible on any screen size. Inside, a LinearLayout arranges all the fields vertically.The form begins with a centered "Add a New Hike" title and is followed by a series of input fields styled using Material Design's TextInputLayout for a modern and user-friendly interface. At the end of the form is a "Submit" button for the user to save the entered information.Here are the input fields on the form:

- **Hike Name**: A text input field for the name of the hike.
- **Location**: A text input field for the location, accompanied by a "Get Location" button.
- **Date**: A non-editable text field that is intended to open a date picker when clicked.
- **Parking**: A RadioGroup allowing the user to select either "Yes" or "No".
- **Length**: A numerical input field for the hike's length.

- **Difficulty**: A dropdown menu (AutoCompleteTextView) for selecting a difficulty level.
- **Description**: A multi-line text input field for a more detailed description of the hike.

3. hike_list_item.xml

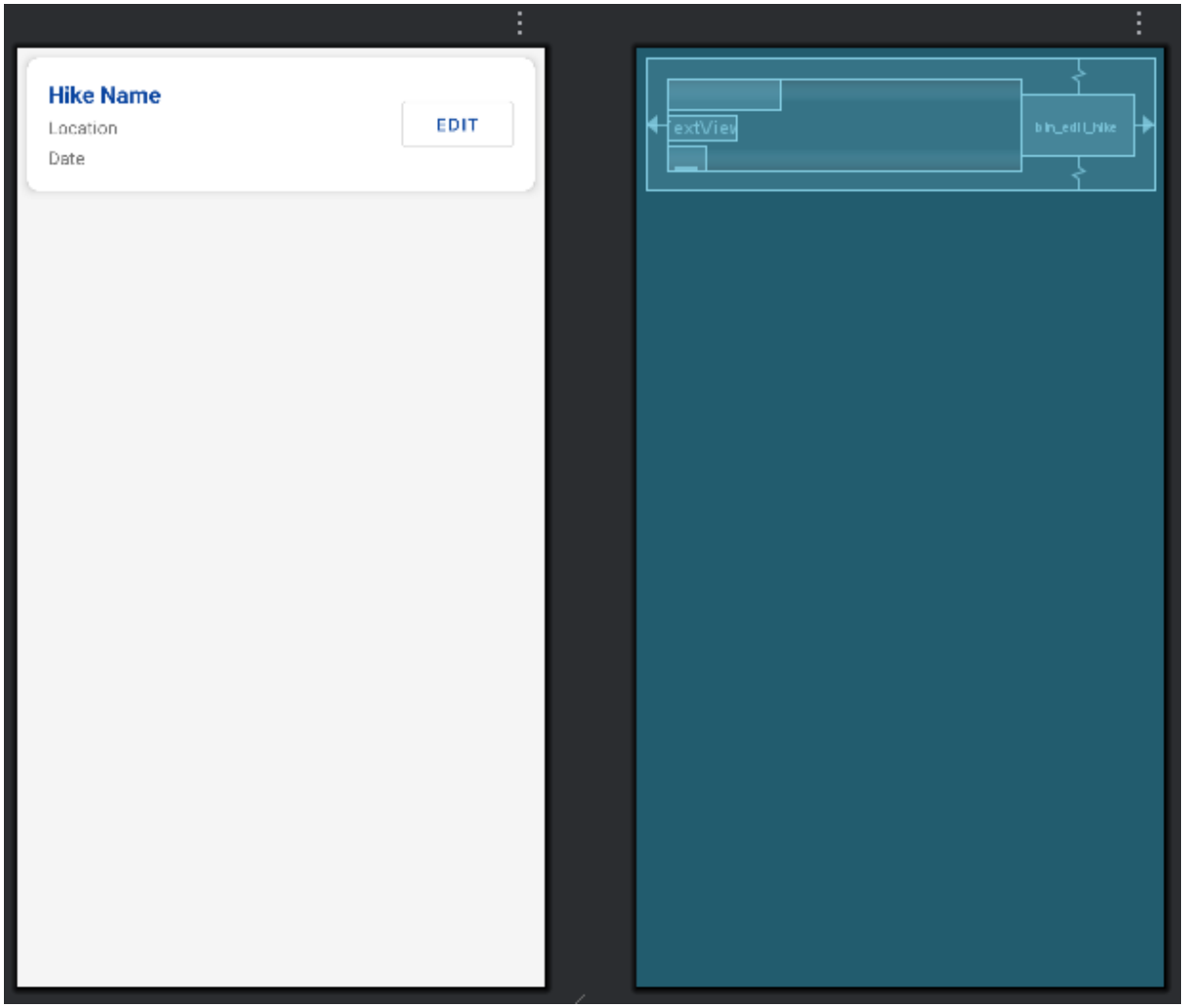

Figure 3 List hike screen

```
<?xml version="1.0" encoding="utf-8"?>
<com.google.android.material.card.MaterialCardView
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    app:cardElevation="4dp"
    app:cardCornerRadius="8dp">

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="16dp">
```

```
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:layout_alignParentStart="true"
        android:layout_toStartOf="@+id/btn_edit_hike">

        <TextView
            android:id="@+id/item_hike_name"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hike Name"
            android:textSize="18sp"
            android:textStyle="bold"
            android:textColor="?attr/colorPrimary"/>

        <TextView
            android:id="@+id/item_hike_location"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Location"
            android:layout_marginTop="4dp"/>

        <TextView
            android:id="@+id/item_hike_date"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Date"
            android:layout_marginTop="4dp"/>
    </LinearLayout>

    <com.google.android.material.button.MaterialButton
        android:id="@+id/btn_edit_hike"
        style="?attr/materialButtonOutlinedStyle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Edit"
        android:layout_alignParentEnd="true"
        android:layout_centerVertical="true"/>

  </RelativeLayout>

</com.google.android.material.card.MaterialCardView>
```

The hike_list_item.xml file defines the layout for a single hiking entry in a RecyclerView. The item is wrapped in a MaterialCardView, giving it a clean, modern look with rounded corners and subtle elevation. Inside, a RelativeLayout arranges elements efficiently. On the left, a vertical LinearLayout displays the hike's name, location, and date. The name is bold and larger to make it stand out, while the location and date appear below as supporting details. On the right side, an outlined MaterialButton serves as the "Edit" button, aligned to the end and centered vertically for easy access.

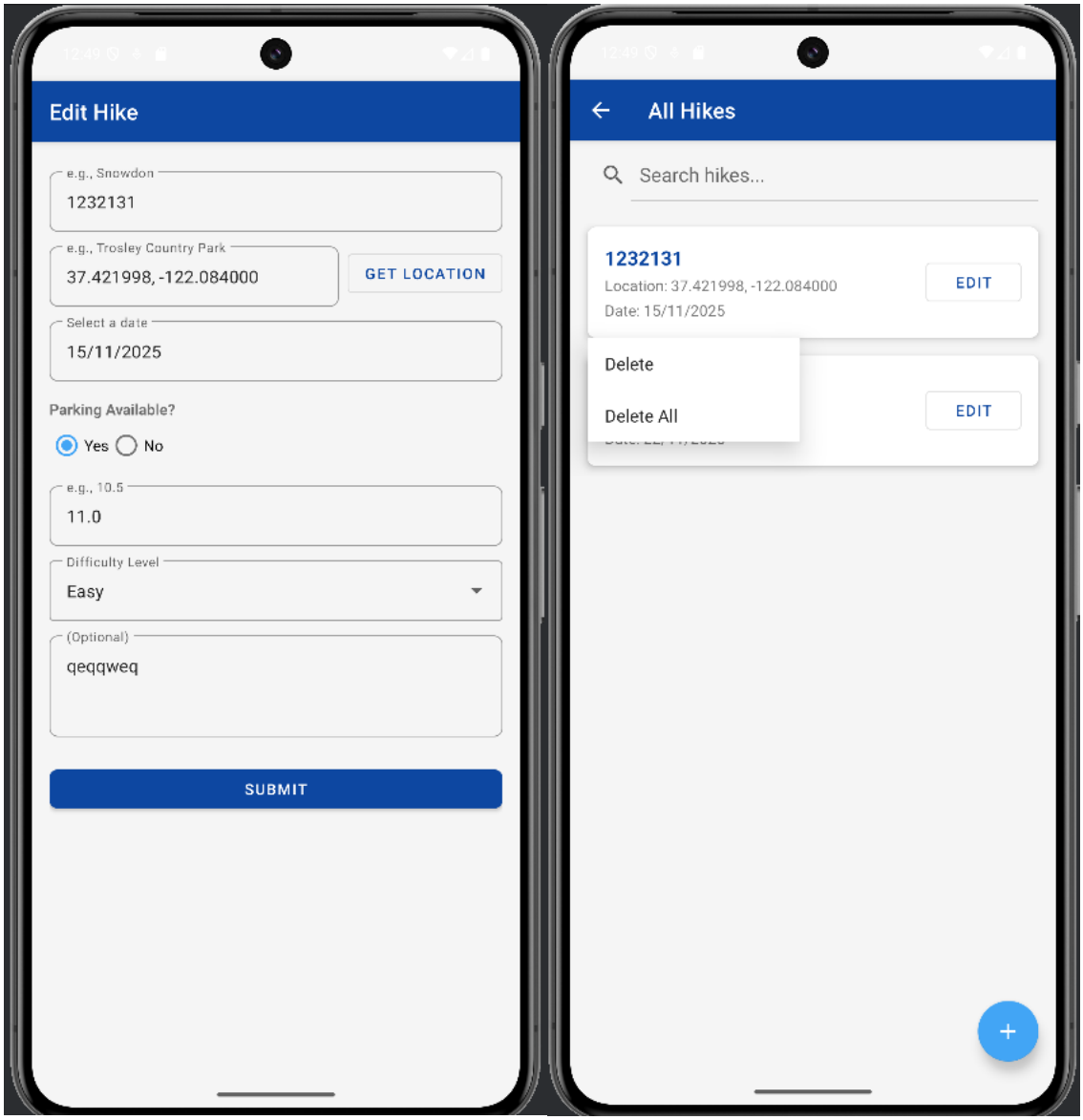

Figure 4 Edit screen and delete hike screen

When you click edit, the same information will appear as when you added it and the old information you entered will appear so you can easily change it when you want to edit it and click submit to close the form and change the information you want successfully.

Besides tapping to edit, the list supports long-press actions. When the user presses and holds an item, the app can show options such as Delete for removing that specific hike or Delete All to clear the entire list. This interaction keeps the layout simple while providing powerful management options.

4. activity_hike_list.xml

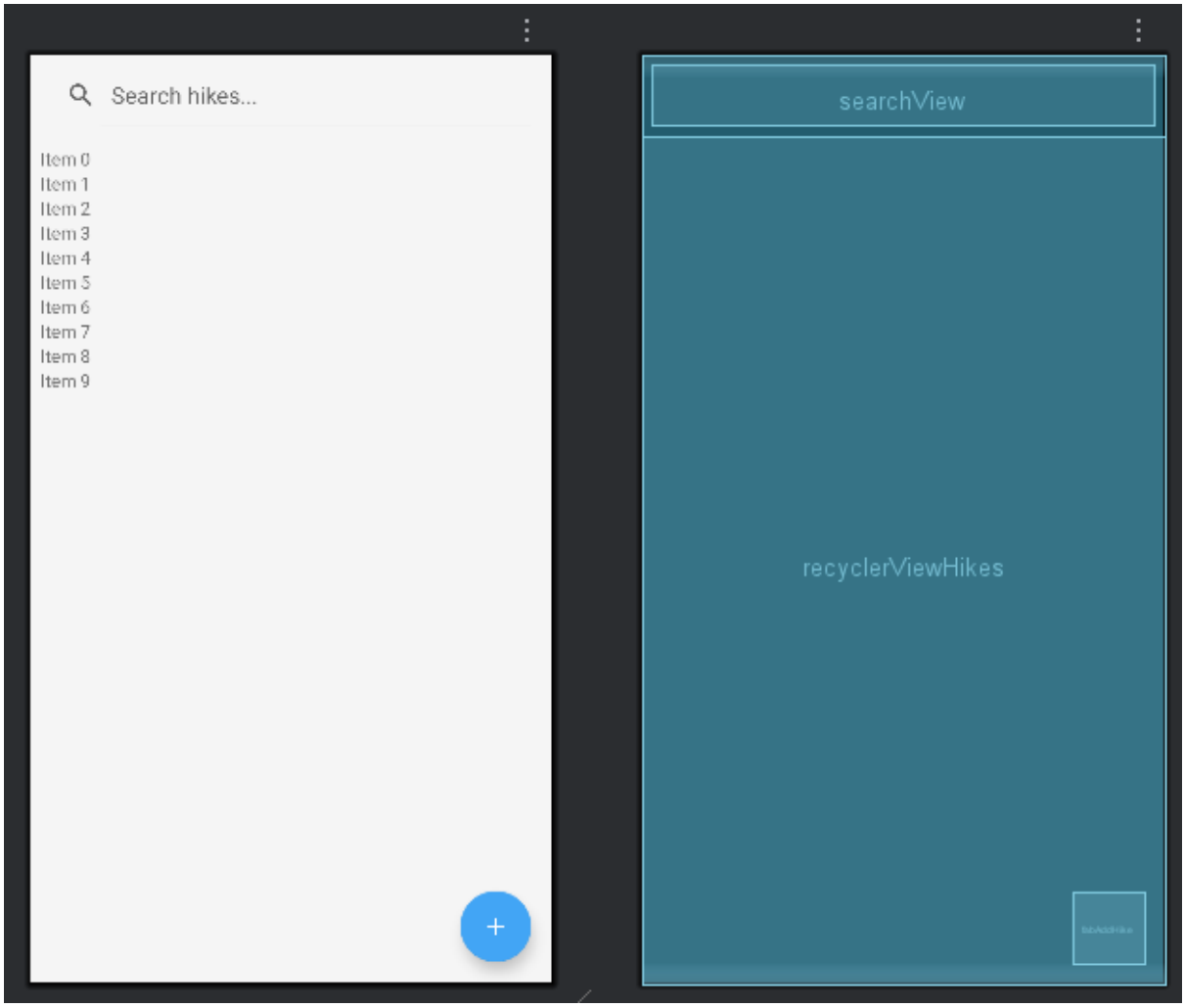

Figure 5 View list item screen

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context=".HikeListActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <SearchView
            android:id="@+id/searchView"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:queryHint="Search hikes..."
            android:iconifiedByDefault="false"
            android:layout_margin="8dp" />
```

```
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerViewHikes"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="8dp"
        android:clipToPadding="false"/>
</LinearLayout>

<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fabAddHike"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="16dp"
    android:src="@drawable/ic_add"
    app:backgroundTint="@color/colorAccent"
    app:tint="@color/white"
    android:contentDescription="Add New Hike"/>

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

This layout defines the main interface of HikeListActivity. The screen is wrapped in a CoordinatorLayout, providing a flexible structure for modern UI behavior. Inside it, a vertical LinearLayout organizes the main components. At the top, a SearchView allows users to search for hikes using keywords. It is expanded by default and includes margins for a clean and accessible look. Below it, a RecyclerView displays the list of hikes in a smooth, scrollable format, with padding to ensure each item appears clear and spacious.

In the bottom-right corner, a FloatingActionButton with a plus icon is positioned for adding new hikes. Its floating style makes it easy to notice and quick to access. Overall, the interface is clean, intuitive, and optimized for managing and browsing hiking entries.

5. activity_hike_detail.xml

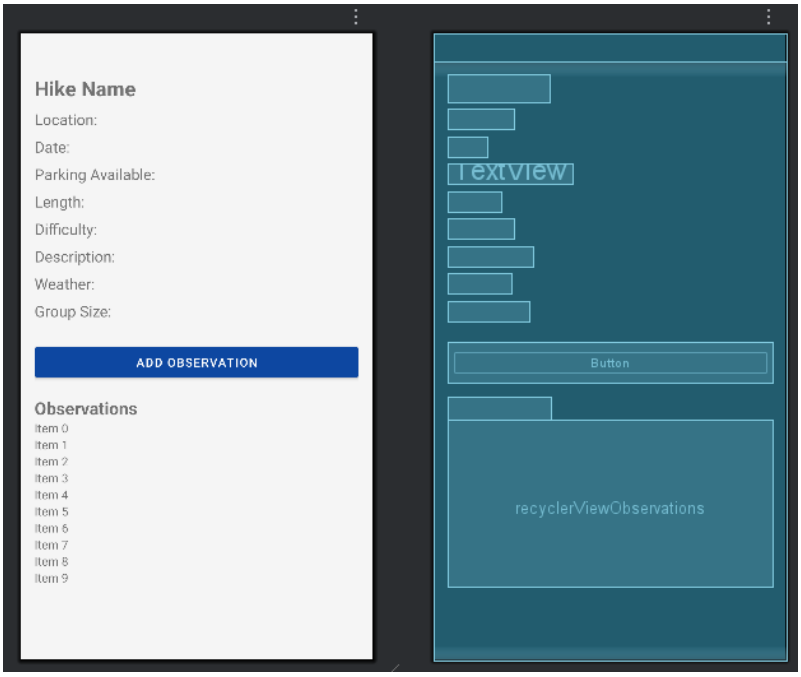

Figure 6 View information detail screen

```xml
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fillViewport="true">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="16dp"
        android:layout_marginTop="32dp">

        <!-- Hike Details -->
        <TextView
            android:id="@+id/detail_hike_name"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hike Name"
            android:textSize="24sp"
            android:textStyle="bold" />
        <TextView
            android:id="@+id/detail_hike_location"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Location: "
            android:textSize="18sp"
            android:layout_marginTop="8dp"/>
        <TextView
            android:id="@+id/detail_hike_date"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Date: "
            android:textSize="18sp"
            android:layout_marginTop="8dp"/>
        <TextView
            android:id="@+id/detail_hike_parking"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Parking Available: "
            android:textSize="18sp"
            android:layout_marginTop="8dp"/>
        <TextView
            android:id="@+id/detail_hike_length"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Length: "
            android:textSize="18sp"
            android:layout_marginTop="8dp"/>
        <TextView
            android:id="@+id/detail_hike_difficulty"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Difficulty: "
            android:textSize="18sp"
```

```
        android:layout_marginTop="8dp"/>
    <TextView
        android:id="@+id/detail_hike_description"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Description: "
        android:textSize="18sp"
        android:layout_marginTop="8dp"/>
    <TextView
        android:id="@+id/detail_hike_weather"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Weather: "
        android:textSize="18sp"
        android:layout_marginTop="8dp"/>
    <TextView
        android:id="@+id/detail_hike_group_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Group Size: "
        android:textSize="18sp"
        android:layout_marginTop="8dp"/>

    <!-- Add Observation Button -->
    <Button
        android:id="@+id/btnAddObservation"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Add Observation"
        android:layout_marginTop="24dp"/>

    <!-- Observations List -->
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Observations"
        android:textSize="20sp"
        android:textStyle="bold"
        android:layout_marginTop="16dp"/>

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerViewObservations"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:nestedScrollingEnabled="false"/>

  </LinearLayout>
</ScrollView>
```

This layout represents the Hike Detail screen. The entire interface is placed inside a ScrollView, allowing the content to scroll when it becomes long. Inside it, a vertical LinearLayout holds all the detailed information about the hike.

The top section includes several TextViews displaying key details such as the hike name, location, date, parking availability, length, difficulty, description, weather, and group size. Each

piece of information is presented clearly with large text sizes and proper spacing, making it easy for users to read.

Below the details, there is an “Add Observation” button that lets users add new notes or observations related to the hike. At the bottom, a bold “Observations” label and a RecyclerView display the list of added observations.

The overall layout is clean, well-organized, and designed to provide a complete, easy-to-read overview of the hike’s details.

6. observation_list_item.xml

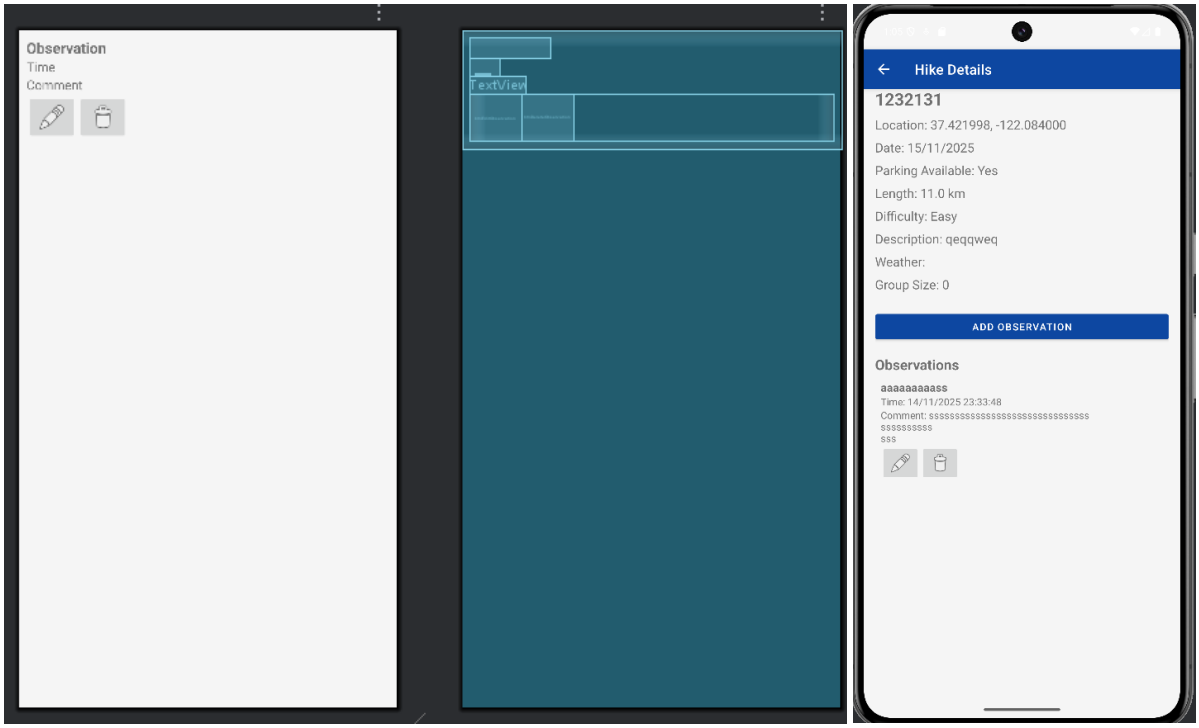

Figure 7 List observation layout screen

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="8dp">

    <TextView
        android:id="@+id/item_observation_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Observation"
        android:textSize="16sp"
        android:textStyle="bold" />

    <TextView
        android:id="@+id/item_observation_time"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Time" />
```

```
    <TextView
        android:id="@+id/item_observation_comment"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Comment" />

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

        <ImageButton
            android:id="@+id/btnEditObservation"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@android:drawable/ic_menu_edit" />

        <ImageButton
            android:id="@+id/btnDeleteObservation"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@android:drawable/ic_menu_delete" />
    </LinearLayout>

</LinearLayout>
```

This XML defines the layout for a single observation item, typically used in a RecyclerView. The main vertical LinearLayout wraps the entire item with 8dp padding, keeping the content organized and readable. At the top, a TextView displays the observation title in bold with a 16sp font to make it stand out. Below it, another TextView shows the time the observation was added, followed by a TextView for the comment or details of the observation. At the bottom, a horizontal LinearLayout contains two ImageButtons: one for editing the observation using a standard edit icon, and one for deleting it using a trash icon. Overall, the layout presents information clearly at the top with action buttons at the bottom, creating an intuitive and user-friendly interface for managing observations.

7. activity_add_observation.xml

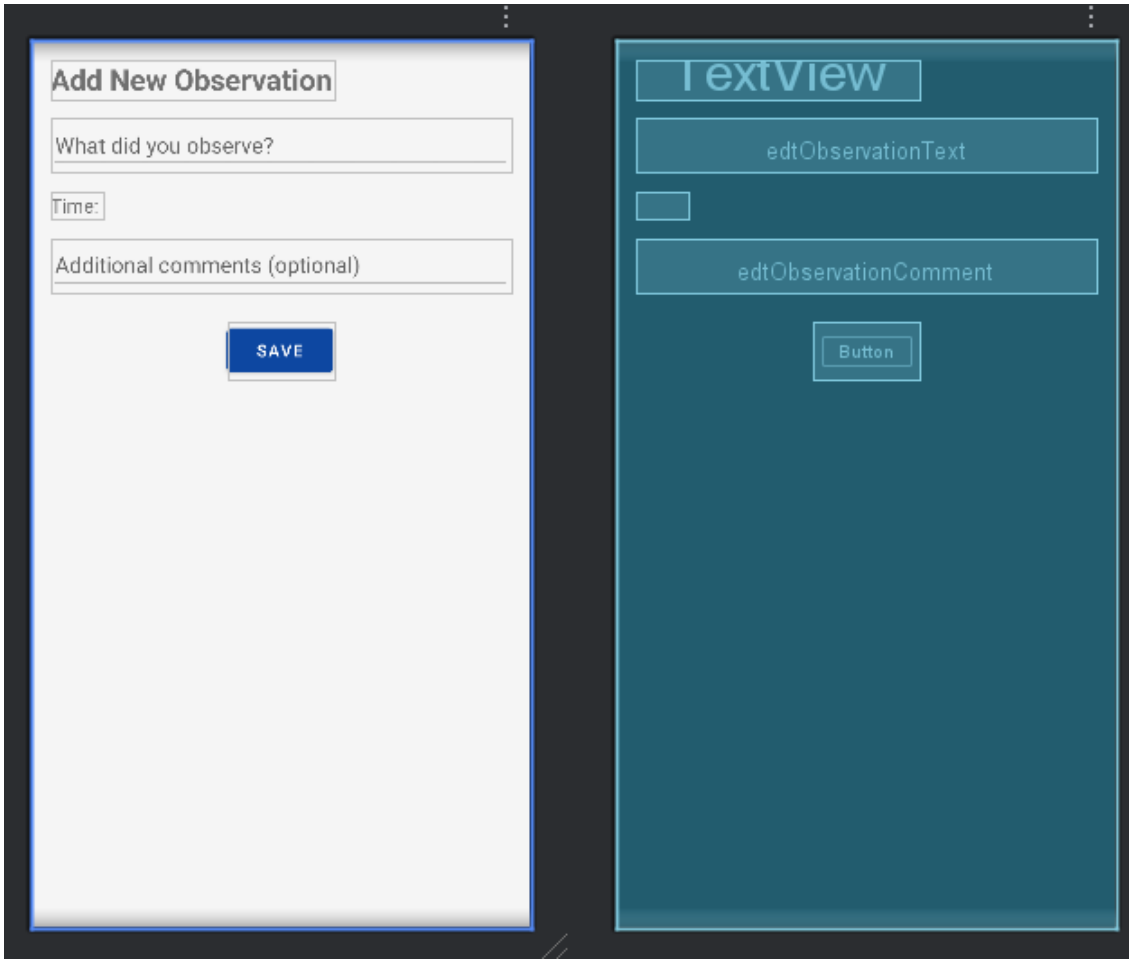

Figure 8 Add observation screen

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Add New Observation"
        android:textSize="24sp"
        android:textStyle="bold"/>

    <EditText
        android:id="@+id/edtObservationText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="What did you observe?"
        android:layout_marginTop="16dp"/>

    <TextView
        android:id="@+id/txtObservationTime"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
    android:text="Time: "
    android:textSize="16sp"
    android:layout_marginTop="16dp"/>

  <EditText
    android:id="@+id/edtObservationComment"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Additional comments (optional)"
    android:layout_marginTop="16dp"/>

  <Button
    android:id="@+id/btnSaveObservation"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Save"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="24dp"/>

</LinearLayout>
```

This layout defines the Add New Observation screen. It is contained within a vertical LinearLayout with 16dp padding, keeping the components organized and easy to interact with. At the top, a TextView displays the title “Add New Observation” in bold with a 24sp font, making it immediately noticeable. Next is an EditText (edtObservationText) where users can enter the observation content, with a hint “What did you observe?”. Below it, a TextView (txtObservationTime) shows the current time, indicating when the observation is recorded. Another EditText (edtObservationComment) allows users to add optional additional comments, with proper spacing from the other fields.

At the bottom, a Save button (btnSaveObservation) is centered horizontally, allowing users to save the new observation. Overall, the layout is clean, simple, and user-friendly, making it easy to input and save observation data efficiently.

8. activity_edit_observation.xml

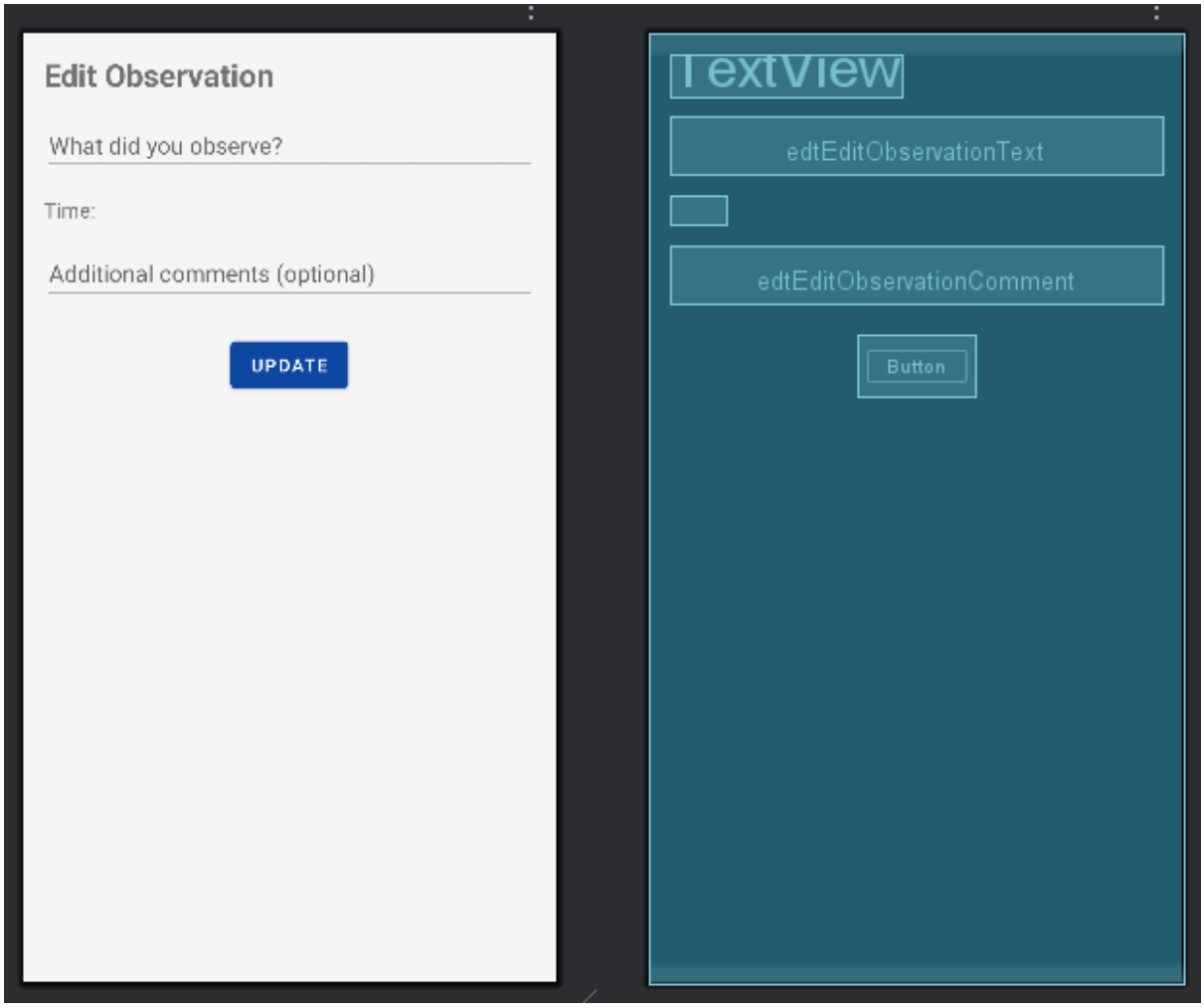

Figure 9 Edit observation screen

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Edit Observation"
        android:textSize="24sp"
        android:textStyle="bold"/>

    <EditText
        android:id="@+id/edtEditObservationText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="What did you observe?"
        android:layout_marginTop="16dp"/>

    <TextView
        android:id="@+id/txtEditObservationTime"
        android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
    android:text="Time: "
    android:textSize="16sp"
    android:layout_marginTop="16dp"/>

  <EditText
    android:id="@+id/edtEditObservationComment"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Additional comments (optional)"
    android:layout_marginTop="16dp"/>

  <Button
    android:id="@+id/btnUpdateObservation"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Update"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="24dp"/>

</LinearLayout>
```

Similar to the interface of the add section, in this edit section, when users want to edit, they can still see the information fields they previously entered to change as they want. When editing successfully, the new information will be updated in the observation list outside.

B. FlutterCode

1.main.dart

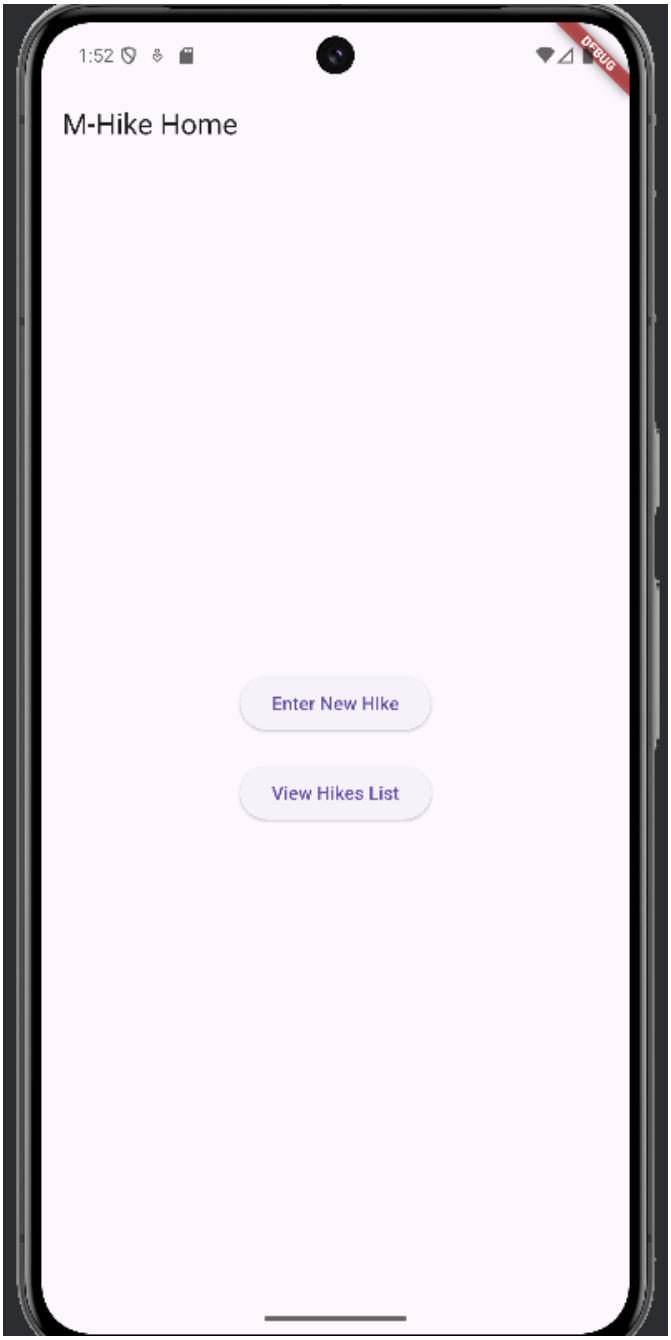

Figure 10 Menu screen flutter

```dart
import 'package:flutter/material.dart';
import 'hike_entry_page.dart';
import 'hike_list_page.dart';

void main() {
  runApp(const MHikeApp());
}

class MHikeApp extends StatelessWidget {
  const MHikeApp({super.key});

  @override
```

```
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'M-Hike',
      theme: ThemeData(
        primarySwatch: Colors.green,
        useMaterial3: true,
      ),
      home: const HomePage(),
    );
  }
}

class HomePage extends StatelessWidget {
  const HomePage({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('M-Hike Home')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            ElevatedButton(
              onPressed: () {
                Navigator.push(
                  context,
                  MaterialPageRoute(builder: (_) => const HikeEntryPage()),
                );
              },
              child: const Text('Enter New Hike'),
            ),
            const SizedBox(height: 20),
            ElevatedButton(
              onPressed: () {
                Navigator.push(
                  context,
                  MaterialPageRoute(builder: (_) => const HikeListPage()),
                );
              },
              child: const Text('View Hikes List'),
            ),
          ],
        ),
      ),
    );
  }
}
```

The main.dart file acts as the primary entry point for the "M-Hike" Flutter application. Execution begins in the main function, which launches the root MHikeApp widget that configures essential app settings, including the application title, a primary green color theme (Colors.green), and the activation of Material 3 design standards. The initial screen displayed to the user is the HomePage, which presents a simple interface featuring an "M-Hike Home" title bar and two central buttons: "Enter New Hike," which navigates the user to the data

entry screen (HikeEntryPage), and "View Hikes List," which directs them to the screen displaying saved hikes (HikeListPage), providing immediate access to the app's core features.

2. m_hike_database.dart

```dart
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';

class MHikeDatabase {
 static Database? _db;

 static Future<Database> get database async {
  if (_db != null) return _db!;
  _db = await _initDB();
  return _db!;
 }

 static Future<Database> _initDB() async {
  final path = join(await getDatabasesPath(), 'mhike.db');
  return await openDatabase(path, version: 1, onCreate: (db, version) async {
   await db.execute('''
    CREATE TABLE hikes(
     id INTEGER PRIMARY KEY AUTOINCREMENT,
     name TEXT,
     location TEXT,
     date TEXT,
     parking TEXT,
     length TEXT,
     difficulty TEXT,
     description TEXT,
     custom1 TEXT,
     custom2 TEXT
    )
   ''');
  });
 }

 static Future<void> saveHike(Map<String, dynamic> hike) async {
  final db = await database;
  await db.insert('hikes', hike);
 }

 static Future<List<Map<String, dynamic>>> getHikes() async {
  final db = await database;
  return await db.query('hikes', orderBy: 'date DESC');
 }

 static Future<void> deleteHike(int id) async {
  final db = await database;
  await db.delete('hikes', where: 'id = ?', whereArgs: [id]);
 }

 static Future<void> deleteAll() async {
  final db = await database;
  await db.delete('hikes');
 }
```

```
  static Future<void> updateHike(int id, Map<String, dynamic> hike) async {
    final db = await database;
    await db.update('hikes', hike, where: 'id = ?', whereArgs: [id]);
  }
}
```

The m_hike_database.dart file acts as the central SQLite database manager for the "M-Hike" Flutter application. It employs a Singleton pattern to ensure that only one database connection (_db) is created and reused throughout the app's lifecycle. Upon initial execution via the _initDB function, it creates a table named hikes with columns designed to store trip details such as name, location, date, length, and difficulty. This class provides a complete set of static methods to perform CRUD (Create, Read, Update, Delete) operations, including saveHike to insert new entries, getHikes to retrieve the list sorted by the most recent date, updateHike to modify existing records, deleteHike to remove specific trips, and deleteAll to wipe the data, thereby ensuring information persists locally on the device even after the app is closed.

3. hike_list_page.dart

Figure 11 View list hike screen

```
import 'package:flutter/material.dart';
import 'hike_detail_page.dart';
```

```dart
import 'm_hike_database.dart';

class HikeListPage extends StatefulWidget {
 const HikeListPage({super.key});

 @override
 State<HikeListPage> createState() => _HikeListPageState();
}

class _HikeListPageState extends State<HikeListPage> {
 List<Map<String, dynamic>> hikes = [];

 @override
 void initState() {
  super.initState();
  _loadHikes();
 }

 Future<void> _loadHikes() async {
  hikes = await MHikeDatabase.getHikes();
  if (mounted) setState(() {});
 }

 Future<void> _deleteHike(int id) async {
  await MHikeDatabase.deleteHike(id);
  await _loadHikes();
 }

 Future<void> _deleteAll() async {
  bool confirm = await showDialog(
   context: context,
   builder: (context) => AlertDialog(
    title: const Text('Delete All Hikes?'),
    content: const Text('This will delete all hikes permanently.'),
    actions: [
     TextButton(onPressed: () => Navigator.pop(context, false), child: const Text('Cancel')),
     ElevatedButton(onPressed: () => Navigator.pop(context, true), child: const Text('Delete All')),
    ],
   ),
  );

  if (confirm) {
   await MHikeDatabase.deleteAll();
   await _loadHikes();
  }
 }

 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
    title: const Text('Hikes List'),
    actions: [IconButton(onPressed: _deleteAll, icon: const Icon(Icons.delete_forever))],
   ),
   body: hikes.isEmpty
     ? const Center(child: Text('No hikes available'))
     : ListView.builder(
```

```
      itemCount: hikes.length,
      itemBuilder: (context, index) {
        final hike = hikes[index];
        return Card(
          margin: const EdgeInsets.symmetric(horizontal: 12, vertical: 6),
          child: ListTile(
            title: Text(hike['name'], style: const TextStyle(fontWeight: FontWeight.bold)),
            subtitle: Text(hike['location']),
            onTap: () async {
              bool? updated = await Navigator.push(
                context,
                MaterialPageRoute(builder: (_) => HikeDetailPage(hike: hike)),
              );
              if (updated == true) await _loadHikes();
            },
            trailing: IconButton(
              icon: const Icon(Icons.delete, color: Colors.red),
              onPressed: () async {
                bool confirm = await showDialog(
                  context: context,
                  builder: (context) => AlertDialog(
                    title: const Text('Delete Hike?'),
                    content: Text('Are you sure you want to delete "${hike['name']}"?'),
                    actions: [
                      TextButton(onPressed: () => Navigator.pop(context, false), child: const Text('Cancel')),
                      ElevatedButton(onPressed: () => Navigator.pop(context, true), child: const Text('Delete')),
                    ],
                  ),
                );
                if (confirm) await _deleteHike(hike['id']);
              },
            ),
          ),
        );
      },
    ),
  );
 }
}
```

The hike_list_page.dart file is responsible for displaying a scrollable list (ListView) of all hikes stored in the application. Upon initialization, the _loadHikes function queries data from the MHikeDatabase to update the UI; if the database is empty, a "No hikes available" message is shown. Each item in the list displays the hike's name and location, allowing users to tap on it to view details (HikeDetailPage) or tap the red trash icon to delete that specific entry after confirming via a dialog. Additionally, the app bar features a "Delete All" button for clearing all data at once, and the navigation logic ensures the list automatically reloads whenever the user returns from the detail page or performs a deletion.

4. hike_entry_page.dart

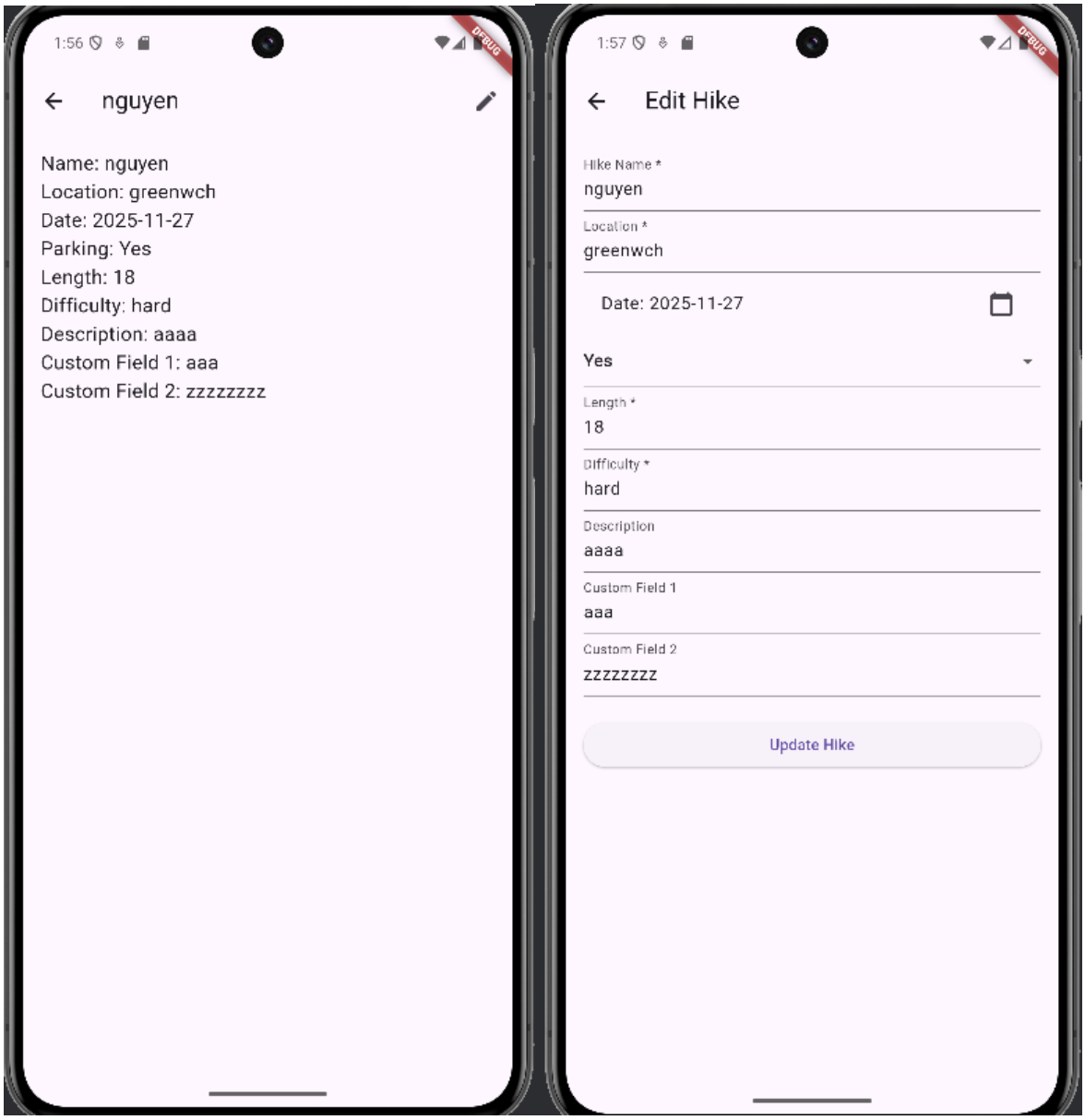

Figure 12 Detail and edit screen

```dart
import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import 'm_hike_database.dart';

class HikeEntryPage extends StatefulWidget {
  final Map<String, dynamic>? hikeToEdit;
  const HikeEntryPage({super.key, this.hikeToEdit});

  @override
  State<HikeEntryPage> createState() => _HikeEntryPageState();
}

class _HikeEntryPageState extends State<HikeEntryPage> {
  final _formKey = GlobalKey<FormState>();

  final TextEditingController nameController = TextEditingController();
  final TextEditingController locationController = TextEditingController();
  final TextEditingController lengthController = TextEditingController();
```

```dart
final TextEditingController difficultyController = TextEditingController();
final TextEditingController descriptionController = TextEditingController();
final TextEditingController custom1Controller = TextEditingController();
final TextEditingController custom2Controller = TextEditingController();

String? parking;
DateTime hikeDate = DateTime.now();

@override
void initState() {
  super.initState();
  if (widget.hikeToEdit != null) {
    final hike = widget.hikeToEdit!;
    nameController.text = hike['name'];
    locationController.text = hike['location'];
    lengthController.text = hike['length'];
    difficultyController.text = hike['difficulty'];
    descriptionController.text = hike['description'] ?? '';
    custom1Controller.text = hike['custom1'] ?? '';
    custom2Controller.text = hike['custom2'] ?? '';
    parking = hike['parking'];
    hikeDate = DateTime.parse(hike['date']);
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text(widget.hikeToEdit != null ? 'Edit Hike' : 'Enter Hike Details')),
    body: Padding(
      padding: const EdgeInsets.all(16.0),
      child: Form(
        key: _formKey,
        child: ListView(
          children: [
            TextFormField(
              controller: nameController,
              decoration: const InputDecoration(labelText: 'Hike Name *'),
              validator: (value) => value!.isEmpty ? 'Required' : null,
            ),
            TextFormField(
              controller: locationController,
              decoration: const InputDecoration(labelText: 'Location *'),
              validator: (value) => value!.isEmpty ? 'Required' : null,
            ),
            ListTile(
              title: Text('Date: ${DateFormat('yyyy-MM-dd').format(hikeDate)}'),
              trailing: const Icon(Icons.calendar_today),
              onTap: _pickDate,
            ),
            DropdownButtonFormField<String>(
              value: parking,
              hint: const Text('Parking Available *'),
              items: const ['Yes', 'No'].map((e) => DropdownMenuItem(value: e, child: Text(e))).toList(),
              onChanged: (val) => setState(() => parking = val),
              validator: (value) => value == null ? 'Required' : null,
            ),
```

```
          TextFormField(
            controller: lengthController,
            decoration: const InputDecoration(labelText: 'Length *'),
            validator: (value) => value!.isEmpty ? 'Required' : null,
          ),
          TextFormField(
            controller: difficultyController,
            decoration: const InputDecoration(labelText: 'Difficulty *'),
            validator: (value) => value!.isEmpty ? 'Required' : null,
          ),
          TextFormField(
            controller: descriptionController,
            decoration: const InputDecoration(labelText: 'Description'),
          ),
          TextFormField(
            controller: custom1Controller,
            decoration: const InputDecoration(labelText: 'Custom Field 1'),
          ),
          TextFormField(
            controller: custom2Controller,
            decoration: const InputDecoration(labelText: 'Custom Field 2'),
          ),
          const SizedBox(height: 20),
          ElevatedButton(
            onPressed: _submitHike,
            child: Text(widget.hikeToEdit != null ? 'Update Hike' : 'Submit'),
          )
        ],
      ),
    ),
  ),
 );
}

Future<void> _pickDate() async {
 DateTime? date = await showDatePicker(
  context: context,
  initialDate: hikeDate,
  firstDate: DateTime(2000),
  lastDate: DateTime(2100),
 );
 if (date != null) setState(() => hikeDate = date);
}

Future<void> _submitHike() async {
 if (_formKey.currentState!.validate() && parking != null) {
  bool confirmed = await showDialog(
   context: context,
   builder: (context) => AlertDialog(
    title: const Text('Confirm Hike'),
    content: Text(
     'Name: ${nameController.text}\n'
       'Location: ${locationController.text}\n'
       'Date: ${DateFormat('yyyy-MM-dd').format(hikeDate)}\n'
       'Parking: $parking\n'
       'Length: ${lengthController.text}\n'
       'Difficulty: ${difficultyController.text}\n'
```

```
            'Description: ${descriptionController.text}\n'
            'Custom1: ${custom1Controller.text}\n'
            'Custom2: ${custom2Controller.text}',
        ),
        actions: [
          TextButton(onPressed: () => Navigator.pop(context, false), child: const Text('Edit')),
          ElevatedButton(onPressed: () => Navigator.pop(context, true), child: const Text('Confirm')),
        ],
      ),
    );

    if (confirmed) {
      Map<String, dynamic> hikeData = {
        'name': nameController.text,
        'location': locationController.text,
        'date': hikeDate.toIso8601String(),
        'parking': parking!,
        'length': lengthController.text,
        'difficulty': difficultyController.text,
        'description': descriptionController.text,
        'custom1': custom1Controller.text,
        'custom2': custom2Controller.text,
      };

      if (widget.hikeToEdit != null) {
        await MHikeDatabase.updateHike(widget.hikeToEdit!['id'], hikeData);
        if (!mounted) return;
        ScaffoldMessenger.of(context).showSnackBar(const SnackBar(content: Text('Hike updated!')));
      } else {
        await MHikeDatabase.saveHike(hikeData);
        if (!mounted) return;
        ScaffoldMessenger.of(context).showSnackBar(const SnackBar(content: Text('Hike saved!')));
      }

      Navigator.pop(context, true);
    }
  }
}
```

The hike_entry_page.dart file serves as a versatile input screen that allows users to either add a new hike or edit an existing one. Upon initialization, it checks for provided data to decide whether to pre-fill fields (edit mode) or start fresh (add mode). The interface utilizes a Form with mandatory validation for critical fields like name, location, and difficulty, along with a date picker and a dropdown menu for parking availability. A key feature is the confirmation dialog displayed upon submission, which forces users to review their entries before the data is committed to the SQLite database via MHikeDatabase, ensuring accuracy before returning to the list view.

5. hike_detail_page.dart

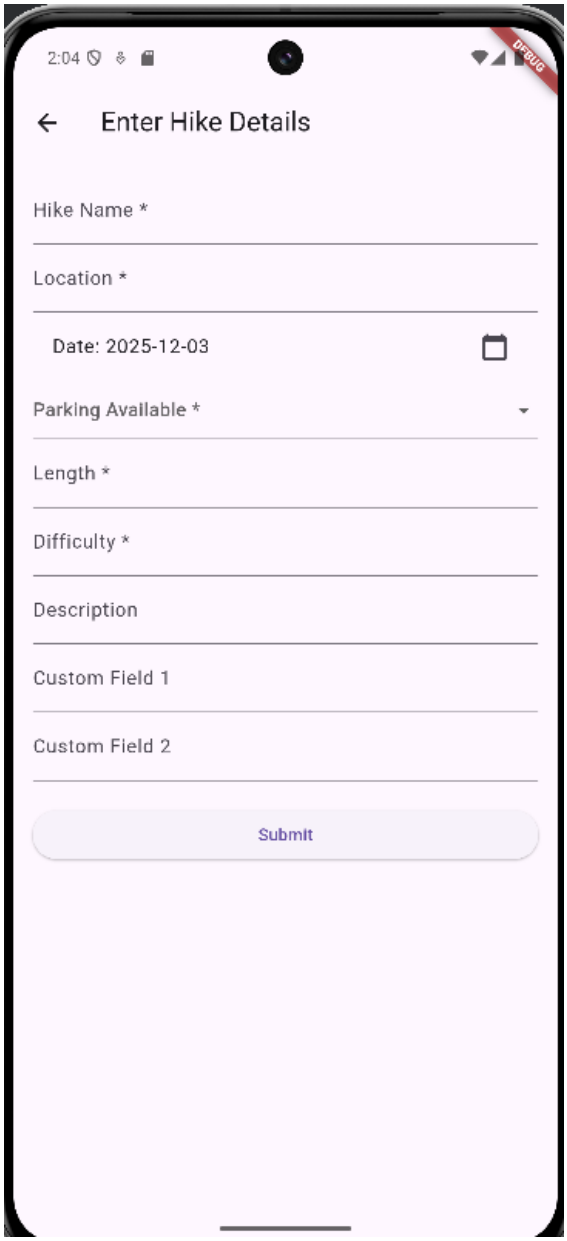

Figure 13 Add hike screen

```dart
import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import 'hike_entry_page.dart';

class HikeDetailPage extends StatelessWidget {
 final Map<String, dynamic> hike;
 const HikeDetailPage({super.key, required this.hike});

 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
    title: Text(hike['name']),
    actions: [
     IconButton(
      icon: const Icon(Icons.edit),
      onPressed: () async {
```

```
            bool? updated = await Navigator.push(
              context,
              MaterialPageRoute(builder: (_) => HikeEntryPage(hikeToEdit: hike)),
            );
            if (updated == true && context.mounted) {
              Navigator.pop(context, true);
            }
          },
        ),
      ],
    ),
    body: Padding(
      padding: const EdgeInsets.all(16.0),
      child: ListView(
        children: [
          Text("Name: ${hike['name']}", style: const TextStyle(fontSize: 18)),
          Text("Location: ${hike['location']}", style: const TextStyle(fontSize: 18)),
          Text("Date: ${DateFormat('yyyy-MM-dd').format(DateTime.parse(hike['date']))}", style: const
TextStyle(fontSize: 18)),
          Text("Parking: ${hike['parking']}", style: const TextStyle(fontSize: 18)),
          Text("Length: ${hike['length']}", style: const TextStyle(fontSize: 18)),
          Text("Difficulty: ${hike['difficulty']}", style: const TextStyle(fontSize: 18)),
          Text("Description: ${hike['description']}", style: const TextStyle(fontSize: 18)),
          Text("Custom Field 1: ${hike['custom1']}", style: const TextStyle(fontSize: 18)),
          Text("Custom Field 2: ${hike['custom2']}", style: const TextStyle(fontSize: 18)),
        ],
      ),
    ),
  );
 }
}
```

The hike_detail_page.dart file serves as the detailed view for a specific hike. It accepts hike data passed from the list view and presents all attributes—such as name, location, formatted date, parking status, and descriptions—in a scrollable vertical list. A key feature is the "Edit" button in the AppBar; tapping it navigates the user to the HikeEntryPage in edit mode, pre-filled with the current data. If the user successfully updates the hike, this detail screen closes and signals the parent list view to refresh and display the latest changes.

3. REFLECTION

The development of the M-Hike application was a foundational journey into native Android development, centered around core principles of data management and user interface design. The process involved creating a multi-screen application capable of performing essential CRUD (Create, Read, Update, Delete) operations, backed by a local SQLite database.

What Went Well

- The project established a clear and logical application flow.
- Separation of concerns was well-executed, with distinct Activities for each primary function:

    - Main menu
    - Form for adding/editing hikes
    - List view for all hikes
    - Detail view managing related observations
- Use of **RecyclerView** with custom adapters (HikeAdapter, ObservationAdapter) efficiently displayed dynamic lists.
- Encapsulation of all database logic within a single **DatabaseHelper** class ensured a clean separation between data and UI layers, making the codebase easier to navigate and debug.
- Implementation of **Material Design** components, such as MaterialCardView and TextInputLayout, contributed to a polished and modern user interface, providing a good user experience.

Lessons Learned

- Proper configuration of **AndroidManifest.xml** is crucial. An early issue where the "edit observation" screen failed to launch was due to forgetting to declare EditObservationActivity. Even perfectly written Java code will fail if core configuration is incomplete.
- Understanding the **Android Activity lifecycle** is important. Correct use of methods like onResume to refresh lists after returning from edit/add screens ensured the UI reflected the current state of the database.
- Implementing features like **location access** highlighted the need for handling **runtime permissions**, respecting user privacy in modern Android development.

What Could Have Been Improved and How

While functional, the application's architecture could be significantly enhanced. Currently, Activities handle UI logic, data loading, and user input, which places a heavy burden on them. Improvements include:

1. **Introduce a ViewModel (MVVM Architecture)**
    - Move all data-related logic and state out of Activities.
    - Activities would only observe data changes (via LiveData or Flow) and update the UI.
    - Solves issues with data loss during configuration changes (e.g., screen rotation).
2. **Migrate to Room**
    - Manual implementation of SQLiteOpenHelper is error-prone and requires boilerplate code.
    - Room provides compile-time verification of SQL queries, simplifies database setup, and integrates seamlessly with ViewModel and LiveData.
    - Makes the data access layer more robust and concise.
3. **Use Coroutines for Background Threads**
    - Currently, database operations are performed on the main UI thread, which can freeze the app for larger datasets.
    - Moving database calls to a background thread via **Kotlin Coroutines** simplifies asynchronous programming and keeps the app responsive.

4. EVALUATION

The M-Hike application serves as a digital logbook for outdoor enthusiasts, allowing them to record, manage, and review their hiking excursions. Developed as a native Android application, it provides core functionality for creating, reading, updating, and deleting hike details and associated observations, all stored locally on the user's device.
This evaluation critically analyzes the application based on several key software development aspects, including **human-computer interaction (HCI), security, adaptability to screen sizes,** and **necessary changes for production deployment.** Each point is justified with specific examples from the application's source code and design.

1. Human-Computer Interaction (HCI)
Human-computer interaction focuses on the design and use of computer technology, specifically the interface between users and computers. A successful application must be **intuitive, efficient, and satisfying** to use. The M-Hike app demonstrates a solid understanding of fundamental HCI principles.
Strengths and Justification

- **Simplicity and clarity:** The main screen (activity_main.xml) presents two clear actions: **"ADD A NEW HIKE"** and **"VIEW ALL HIKES"**, reducing cognitive load.
- **Predictable navigation:** Linear flow from hike list (HikeListActivity) → detail view (HikeDetailActivity) → add/edit related data (AddObservationActivity, EditObservationActivity). The **Up button** in the ActionBar ensures users can return to previous screens.
- **Feedback mechanisms:**
    - Toast messages confirm successful actions, e.g., "Hike saved successfully!".
    - TextInputLayout provides inline validation errors, e.g., "Name is required."
    - AlertDialogs prevent accidental deletion, following Jakob Nielsen’s heuristic of **error prevention**.
- **Consistent UI design:** Uses **Material Design** components:
    - MaterialCardView for organized lists (hike_list_item.xml)
    - MaterialButton for consistent interactive elements
    - DatePickerDialog for date selection and AutoCompleteTextView for difficulty level selection

2. Security
Security is essential to protect user data and maintain trust. As an offline application, M-Hike has minimal security, but existing vulnerabilities exist.
Current State and Justification

- **Unencrypted local storage:** The SQLite database managed by DatabaseHelper is stored in plain text. On rooted or compromised devices, data (including location) can be easily extracted.
- **Client-side validation only:** Checks for empty strings, but this does not prevent malicious input if connected to a backend.
- **Positive practice:** Correct handling of **runtime permissions** for location access (FusedLocationProviderClient with ACCESS_FINE_LOCATION), respecting user consent and privacy.

3. Ability of the App to Run on a Range of Screen Sizes

Android apps must work consistently across various devices, screen sizes, and orientations.

Strengths

- **ScrollView** in activity_add_hike.xml ensures users can access all fields on small screens.
- Layouts use match_parent and wrap_content for flexible sizing.
- Consistent use of **dp** for layout and **sp** for text ensures proportional scaling across screen densities.

Limitations and Improvements

- Layouts are mostly single-pane, portrait-oriented:
    - Landscape mode: buttons take up too much vertical space
    - Tablets: single-column list/detail views appear stretched
- **Recommended improvements:**
    - Create **alternative layouts** (layout-land, layout-sw600dp)
    - Use **ConstraintLayout** for a more flexible and performant view hierarchy
    - Implement **master-detail layouts** for larger screens

4. Changes Needed for Live Use

Transitioning M-Hike to a production-ready app requires substantial architectural and feature enhancements:

1. **Backend and cloud synchronization:**
    - Users expect persistent, secure data across devices
    - Implement a backend (Node.js, Python, or Firebase) with cloud database (Firestore, PostgreSQL)
    - Refactor app for authentication, network requests, and data sync between SQLite and cloud
2. **Security improvements:**
    - Robust authentication (OAuth 2.0, Firebase Auth)
    - HTTPS for all communication
    - Encrypt local database (SQLCipher)
3. **Architecture overhaul:**
    - Adopt **MVVM**: Activities observe data via ViewModel, improving scalability and handling configuration changes
    - Migrate from **SQLiteOpenHelper** to **Room** for safer, concise database management

- Move database/network operations to background threads using **Kotlin Coroutines**

4. **Error handling and analytics:**
    - Integrate **Firebase Crashlytics** for bug monitoring
    - Integrate **Firebase Analytics** for user engagement insights

5. Other Considerations (Testing and Accessibility)

- **Testing:** Currently lacks automated testing
    - Implement **Unit Tests** (JUnit) for ViewModel/business logic
    - Implement **Instrumentation Tests** (Espresso) for UI behavior and user flows
- **Accessibility (a11y):**
    - Ensure touch targets ≥48dp
    - Provide **content descriptions** for images/buttons
    - Verify **color contrast ratios** meet WCAG standards

5. CODE DESCRIPTION

1.MainActivity.java

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        EdgeToEdge.enable(this);
        setContentView(R.layout.activity_main);

        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main), (v, insets) -> {
            Insets systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars());
            v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom);
            return insets;
        });

        Button btnGoToAddHike = findViewById(R.id.btnGoToAddHike);
        Button btnGoToViewAllHikes = findViewById(R.id.btnGoToViewAllHikes);

        btnGoToAddHike.setOnClickListener(v -> {
            Intent intent = new Intent(MainActivity.this, AddHikeActivity.class);
            startActivity(intent);
        });

        btnGoToViewAllHikes.setOnClickListener(v -> {
            Intent intent = new Intent(MainActivity.this, HikeListActivity.class);
            startActivity(intent);
        });
    }
}
```

This code sets up the main screen of the application. In the onCreate method, it displays the user interface defined in the activity_main.xml file. It then finds two buttons and assigns click listeners to them. When the "Go To Add Hike" button is pressed, the app navigates to

the AddHikeActivity screen. When the "Go To View All Hikes" button is pressed, the app opens the HikeListActivity screen.
This part of the code enables "edge-to-edge" mode, allowing the app's UI to draw under the system bars for a more modern and immersive look.
1. **EdgeToEdge.enable(this);**: This line enables the edge-to-edge feature.
2. **ViewCompat.setOnApplyWindowInsetsListener(...)**: This code listens for the dimensions of the system bars (like the status bar at the top and the navigation bar at the bottom), which are called "insets".
3. **v.setPadding(...):** Based on the size of those insets, it adds padding to the main layout. This prevents your content from being hidden behind the system bars, ensuring the UI is both beautiful and functional.

2.AddHikeActivity.java

```
public class AddHikeActivity extends AppCompatActivity {

    public static final String EXTRA_HIKE_ID = "com.example.myapplication.EXTRA_HIKE_ID";
    private static final int LOCATION_PERMISSION_REQUEST_CODE = 1;

    private TextInputEditText edtName, edtLocation, edtDate, edtLength, edtDescription;
    private TextInputLayout nameLayout, locationLayout, dateLayout, lengthLayout;
    private AutoCompleteTextView spinnerDifficulty;
    private RadioGroup radioParking;
    private Button btnSubmit, btnGetLocation;
    private TextView txtFormTitle;
    private DatabaseHelper databaseHelper;
    private int hikeId = -1;
    private FusedLocationProviderClient fusedLocationClient;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_add_hike);

        databaseHelper = new DatabaseHelper(this);
        fusedLocationClient = LocationServices.getFusedLocationProviderClient(this);

        setupViews();
        setupDifficultyDropdown();
        setupListeners();

        if (getIntent().hasExtra(EXTRA_HIKE_ID)) {
            hikeId = getIntent().getIntExtra(EXTRA_HIKE_ID, -1);
            if (hikeId != -1) {
                setTitle(R.string.form_title_edit_hike);
                txtFormTitle.setText(R.string.form_title_edit_hike);
                loadHikeData();
            } else {
                setTitle(R.string.form_title_add_hike);
                txtFormTitle.setText(R.string.form_title_add_hike);
            }
        } else {
            setTitle(R.string.form_title_add_hike);
            txtFormTitle.setText(R.string.form_title_add_hike);
```

```
        }
    }

    private void loadHikeData() {
        Hike hike = databaseHelper.getHikeById(hikeId);
        if (hike != null) {
            edtName.setText(hike.getName());
            edtLocation.setText(hike.getLocation());
            edtDate.setText(hike.getDate());
            edtLength.setText(String.valueOf(hike.getLength()));
            if (hike.getParkingAvailable() == 1) {
                radioParking.check(R.id.rdoYes);
            } else {
                radioParking.check(R.id.rdoNo);
            }
            spinnerDifficulty.setText(hike.getDifficulty(), false);
            edtDescription.setText(hike.getDescription());
        }
    }

    private void setupViews() {
        txtFormTitle = findViewById(R.id.txtFormTitle);
        nameLayout = findViewById(R.id.name_layout);
        locationLayout = findViewById(R.id.location_layout);
        dateLayout = findViewById(R.id.date_layout);
        lengthLayout = findViewById(R.id.length_layout);

        edtName = findViewById(R.id.edtName);
        edtLocation = findViewById(R.id.edtLocation);
        edtDate = findViewById(R.id.edtDate);
        edtLength = findViewById(R.id.edtLength);
        radioParking = findViewById(R.id.radioParking);
        spinnerDifficulty = findViewById(R.id.spinnerDifficulty);
        edtDescription = findViewById(R.id.edtDescription);
        btnSubmit = findViewById(R.id.btnSubmit);
        btnGetLocation = findViewById(R.id.btnGetLocation);
    }

    private void setupDifficultyDropdown() {
        String[] difficultyLevels = getResources().getStringArray(R.array.difficulty_levels);
        ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
android.R.layout.simple_dropdown_item_1line, difficultyLevels);
        spinnerDifficulty.setAdapter(adapter);
    }

    private void setupListeners() {
        edtDate.setOnClickListener(v -> showDatePickerDialog());
        btnSubmit.setOnClickListener(v -> {
            if (validateInputs()) {
                showConfirmationDialog();
            }
        });
        btnGetLocation.setOnClickListener(v -> getCurrentLocation());
    }

    private void getCurrentLocation() {
        if (ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) !=
```

```
PackageManager.PERMISSION_GRANTED && ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_COARSE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
LOCATION_PERMISSION_REQUEST_CODE);
        return;
    }
    fusedLocationClient.getLastLocation()
            .addOnSuccessListener(this, location -> {
              if (location != null) {
                edtLocation.setText(String.format(Locale.getDefault(), "%f, %f", location.getLatitude(),
location.getLongitude()));
              } else {
                Toast.makeText(this, "Unable to get location.", Toast.LENGTH_SHORT).show();
              }
            });
  }

  @Override
  public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[]
grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    if (requestCode == LOCATION_PERMISSION_REQUEST_CODE) {
      if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
        getCurrentLocation();
      } else {
        Toast.makeText(this, "Location permission denied.", Toast.LENGTH_SHORT).show();
      }
    }
  }

  private boolean validateInputs() {
    nameLayout.setError(null);
    locationLayout.setError(null);
    dateLayout.setError(null);
    lengthLayout.setError(null);

    String name = edtName.getText().toString().trim();
    String location = edtLocation.getText().toString().trim();
    String date = edtDate.getText().toString().trim();
    String length = edtLength.getText().toString().trim();

    if (TextUtils.isEmpty(name)) {
      nameLayout.setError("Name is required.");
      return false;
    }

    if (TextUtils.isEmpty(location)) {
      locationLayout.setError("Location is required.");
      return false;
    }

    if (TextUtils.isEmpty(date)) {
      dateLayout.setError("Date is required.");
      return false;
    }

    if (radioParking.getCheckedRadioButtonId() == -1) {
```

```
            Toast.makeText(this, "Please select a parking option.", Toast.LENGTH_SHORT).show();
            return false;
        }

        if (TextUtils.isEmpty(length)) {
            lengthLayout.setError("Length is required.");
            return false;
        }

        return true;
    }

    private void showConfirmationDialog() {
        String name = edtName.getText().toString().trim();
        String location = edtLocation.getText().toString().trim();
        String date = edtDate.getText().toString().trim();
        String lengthStr = edtLength.getText().toString().trim();
        String difficulty = spinnerDifficulty.getText().toString();
        String description = edtDescription.getText().toString().trim();

        boolean parkingAvailable = radioParking.getCheckedRadioButtonId() == R.id.rdoYes;
        String parking = parkingAvailable ? "Yes" : "No";

        String confirmationMessage = "Please confirm the details:\n\n" +
                "Name: " + name + "\n" +
                "Location: " + location + "\n" +
                "Date: " + date + "\n" +
                "Parking: " + parking + "\n" +
                "Length: " + lengthStr + " km\n" +
                "Difficulty: " + difficulty + "\n" +
                "Description: " + (!description.isEmpty() ? description : "N/A");

        new AlertDialog.Builder(this)
                .setTitle("Confirm Hike Details")
                .setMessage(confirmationMessage)
                .setPositiveButton("Confirm", (dialog, which) -> {
                    saveHikeAndNavigate();
                })
                .setNegativeButton("Go Back", (dialog, which) -> {
                    dialog.dismiss();
                })
                .create()
                .show();
    }

    private void saveHikeAndNavigate() {
        String name = edtName.getText().toString().trim();
        String location = edtLocation.getText().toString().trim();
        String date = edtDate.getText().toString().trim();
        double length = Double.parseDouble(edtLength.getText().toString().trim());
        int parking = radioParking.getCheckedRadioButtonId() == R.id.rdoYes ? 1 : 0;
        String difficulty = spinnerDifficulty.getText().toString();
        String description = edtDescription.getText().toString().trim();
        String weather = ""; // Not in layout
        int groupSize = 0; // Not in layout

        if (hikeId == -1) {
```

```
            databaseHelper.addHike(name, location, date, parking, length, difficulty, description, weather,
groupSize);
            Toast.makeText(this, "Hike saved successfully!", Toast.LENGTH_SHORT).show();
        } else {
            databaseHelper.updateHike(hikeId, name, location, date, parking, length, difficulty, description,
weather, groupSize);
            Toast.makeText(this, "Hike updated successfully!", Toast.LENGTH_SHORT).show();
        }

        Intent intent = new Intent(AddHikeActivity.this, HikeListActivity.class);
        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP | Intent.FLAG_ACTIVITY_NEW_TASK);
        startActivity(intent);
        finish();
    }

    private void showDatePickerDialog() {
        final Calendar calendar = Calendar.getInstance();
        int year = calendar.get(Calendar.YEAR);
        int month = calendar.get(Calendar.MONTH);
        int day = calendar.get(Calendar.DAY_OF_MONTH);

        new DatePickerDialog(this,
                (view, year1, monthOfYear, dayOfMonth) -> {
                    Calendar newDate = Calendar.getInstance();
                    newDate.set(year1, monthOfYear, dayOfMonth);
                    SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy", Locale.getDefault());
                    edtDate.setText(sdf.format(newDate.getTime()));
                }, year, month, day).show();
    }
}
```

This code manages the "Add/Edit Hike" screen. It functions as a form where users can input details about a hike, such as its name, location, date, and difficulty. The screen has two modes: "Add" for creating a new hike, and "Edit" for modifying an existing one, which is determined by whether a hike ID is passed to it from the previous screen. After the user fills out the form, the code validates the inputs, shows a confirmation dialog, and then saves or updates the hike information in a local database.

The most complex part of this code is the location-fetching functionality. It allows the user to automatically fill in the hike's location using the device's GPS.

1. Initializing the FusedLocationProviderClient

- In the **onCreate()** method, you create the client:
  fusedLocationClient = LocationServices.getFusedLocationProviderClient(this);
- This is the modern Google Play Services API for retrieving the device's location.
- It provides better accuracy and battery optimization compared to older Android location APIs.

2. Permission Check and Request

- When the user taps the Get Location button, the **getCurrentLocation**() method is executed.
- **ActivityCompat.checkSelfPermission(...)** checks if the app already has the **ACCESS_FINE_LOCATION** permission.
- If the app does not have permission:

ActivityCompat.requestPermissions(...)

→ The system shows a permission dialog.

- **LOCATION_PERMISSION_REQUEST_CODE** is used to identify the permission request when handling the result.

3. Handling the Permission Request Result

- The system automatically calls **onRequestPermissionsResult()** after the user selects Allow or Deny.
- Inside this method:
  - You verify the correct **requestCode**.
  - You check whether the permission was actually granted.
- If the permission was granted, **getCurrentLocation**() is called again to continue retrieving the location.
- If the permission was denied, a **Toast** message notifies the user.

4. Fetching the Device Location

- Once the app has permission, it calls: **fusedLocationClient.getLastLocation()**
- This is an asynchronous operation that retrieves the device's last known location.
- .**addOnSuccessListener**(...) runs only when the location is successfully returned.
- If **location != null**:
  - Extract latitude and longitude.
  - Display them in the UI.
- If **location == null**:
  → This may happen if GPS is off, or the device has never recorded a location → show a message to the user.

3. HikeListActivity.java

```
public class HikeListActivity extends AppCompatActivity {

    private RecyclerView recyclerViewHikes;
    private HikeAdapter hikeAdapter;
    private List<Hike> hikeList;
    private DatabaseHelper databaseHelper;
    private SearchView searchView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hike_list);

        if (getSupportActionBar() != null) {
            getSupportActionBar().setDisplayHomeAsUpEnabled(true);
            getSupportActionBar().setTitle("All Hikes");
        }

        databaseHelper = new DatabaseHelper(this);

        recyclerViewHikes = findViewById(R.id.recyclerViewHikes);
        recyclerViewHikes.setLayoutManager(new LinearLayoutManager(this));
```

```
    hikeList = new ArrayList<>();
    hikeAdapter = new HikeAdapter(this, hikeList); // Pass context here
    recyclerViewHikes.setAdapter(hikeAdapter);

    searchView = findViewById(R.id.searchView);
    searchView.setOnQueryTextListener(new SearchView.OnQueryTextListener() {
        @Override
        public boolean onQueryTextSubmit(String query) {
            searchHikes(query);
            return true;
        }

        @Override
        public boolean onQueryTextChange(String newText) {
            searchHikes(newText);
            return true;
        }
    });

    FloatingActionButton fabAddHike = findViewById(R.id.fabAddHike);
    fabAddHike.setOnClickListener(v -> {
        Intent intent = new Intent(HikeListActivity.this, AddHikeActivity.class);
        startActivity(intent);
    });
}

@Override
protected void onResume() {
    super.onResume();
    loadHikesFromDatabase();
}

private void loadHikesFromDatabase() {
    // If search view is not empty, maintain the search results
    if (searchView != null && !searchView.getQuery().toString().isEmpty()) {
        searchHikes(searchView.getQuery().toString());
    } else {
        List<Hike> updatedHikeList = databaseHelper.getAllHikes();
        hikeAdapter.updateData(updatedHikeList);
    }
}

private void searchHikes(String query) {
    List<Hike> searchResults;
    if (query.isEmpty()) {
        searchResults = databaseHelper.getAllHikes();
    } else {
        searchResults = databaseHelper.searchHikes(query);
    }
    hikeAdapter.updateData(searchResults);
}

@Override
public boolean onSupportNavigateUp() {
    finish();
    return true;
```

```
    }
}
```

This code manages the "Hike List" screen. Its main function is to display all the hikes that have been saved in the database. The screen uses a RecyclerView to display the list efficiently. It also includes a SearchView that allows the user to filter the list by name, and a Floating Action Button to open the "Add Hike" screen. The list automatically refreshes each time the user returns to this screen, ensuring the data is always up-to-date.
The core and most notable part of this screen is the combination of the RecyclerView, SearchView, and DatabaseHelper to create a searchable, self-updating list.

1. RecyclerView Setup:

- A **RecyclerView** is an advanced UI component for displaying long lists. It works efficiently by recycling the item views that have scrolled off-screen.
- **hikeAdapter = new HikeAdapter(...)**: The HikeAdapter is a custom class (not present in this file) that acts as a bridge. It receives the data (hikeList) and "knows" how to create the layout for each row in the list and fill it with data.
- **recyclerViewHikes.setAdapter(hikeAdapter)**: This attaches the adapter to the RecyclerView to begin displaying the data.

2. SearchView Integration:

- **searchView.setOnQueryTextListener(...)**: A listener is assigned to the search field.
- **onQueryTextChange(String newText)**: This method is called every time the user types or deletes a character. It immediately calls searchHikes(newText) to update the list in real-time.
- **searchHikes(String query)**: This method queries the database **(databaseHelper.searchHikes(query))** to get the list of matching results and then calls hikeAdapter.updateData(...) to request that the adapter refresh the UI with the new data.

3. Automatic Refresh in onResume():

- The **onResume**() method is part of the Activity lifecycle, called every time the screen becomes visible again (for instance, after adding/editing a hike and navigating back).
- By calling **loadHikesFromDatabase**() here, you ensure the list always shows the most current data without requiring the user to manually refresh.
- The logic inside **loadHikesFromDatabase** is smart: it checks if the user is currently searching for something. If so, it preserves the search results; otherwise, it reloads the entire list. This helps maintain the user's state.

4. Hike.java

```
public class Hike {

    private int id;
    private String name;
    private String location;
    private String date;
    private int parkingAvailable;
    private double length;
    private String difficulty;
```

```
    private String description;
    private String weather;
    private int groupSize;

    // Constructor
    public Hike(int id, String name, String location, String date, int parkingAvailable, double length, String
difficulty, String description, String weather, int groupSize) {
        this.id = id;
        this.name = name;
        this.location = location;
        this.date = date;
        this.parkingAvailable = parkingAvailable;
        this.length = length;
        this.difficulty = difficulty;
        this.description = description;
        this.weather = weather;
        this.groupSize = groupSize;
    }

    // Getters and Setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getLocation() {
        return location;
    }

    public void setLocation(String location) {
        this.location = location;
    }

    public String getDate() {
        return date;
    }

    public void setDate(String date) {
        this.date = date;
    }

    public int getParkingAvailable() {
        return parkingAvailable;
    }

    public void setParkingAvailable(int parkingAvailable) {
```

```
        this.parkingAvailable = parkingAvailable;
    }

    public double getLength() {
        return length;
    }

    public void setLength(double length) {
        this.length = length;
    }

    public String getDifficulty() {
        return difficulty;
    }

    public void setDifficulty(String difficulty) {
        this.difficulty = difficulty;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public String getWeather() {
        return weather;
    }

    public void setWeather(String weather) {
        this.weather = weather;
    }

    public int getGroupSize() {
        return groupSize;
    }

    public void setGroupSize(int groupSize) {
        this.groupSize = groupSize;
    }
}
```

1.Properties: These are the private variables at the top (like **name, location, date**). They are the basic pieces of information that define a hike.
2.Constructor: This is the **public Hike(...)** method. It's the way you create a new Hike object. When you call it, you must provide all the necessary information, and it will create a complete Hike object from that data.
3.Getters and Setters: These are the public methods like **getName()** and **setName().** Other parts of your app (like your Activities) will use these methods to safely read (get) or change (set) the information of a Hike object.Essentially, this class is an organized way to bundle all

the data for a single hike into one object, making it easier to pass that data between screens and save it to the database.

5. HikeDetailActivity.java

```
public class HikeDetailActivity extends AppCompatActivity implements
ObservationAdapter.OnObservationListener {

    public static final String EXTRA_HIKE_ID = "extra_hike_id";

    private DatabaseHelper databaseHelper;
    private Hike hike;
    private int hikeId;

    private TextView detailHikeName, detailHikeLocation, detailHikeDate, detailHikeParking,
detailHikeLength, detailHikeDifficulty, detailHikeDescription, detailHikeWeather, detailHikeGroupSize;
    private Button btnAddObservation;
    private RecyclerView recyclerViewObservations;
    private ObservationAdapter observationAdapter;
    private List<Observation> observationList;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hike_detail);

        if (getSupportActionBar() != null) {
            getSupportActionBar().setDisplayHomeAsUpEnabled(true);
            getSupportActionBar().setTitle("Hike Details");
        }

        databaseHelper = new DatabaseHelper(this);
        initViews();

        hikeId = getIntent().getIntExtra(EXTRA_HIKE_ID, -1);

        if (hikeId != -1) {
            hike = databaseHelper.getHikeById(hikeId);
            if (hike != null) {
                populateHikeDetails();
                setupObservationList();
                setupListeners();
            } else {
                Toast.makeText(this, "Hike not found!", Toast.LENGTH_SHORT).show();
                finish();
            }
        } else {
            Toast.makeText(this, "Invalid Hike ID!", Toast.LENGTH_SHORT).show();
            finish();
        }
    }

    @Override
    protected void onResume() {
        super.onResume();
        if (hikeId != -1) {
```

```
        loadObservations();
    }
}

private void initViews() {
    detailHikeName = findViewById(R.id.detail_hike_name);
    detailHikeLocation = findViewById(R.id.detail_hike_location);
    detailHikeDate = findViewById(R.id.detail_hike_date);
    detailHikeParking = findViewById(R.id.detail_hike_parking);
    detailHikeLength = findViewById(R.id.detail_hike_length);
    detailHikeDifficulty = findViewById(R.id.detail_hike_difficulty);
    detailHikeDescription = findViewById(R.id.detail_hike_description);
    detailHikeWeather = findViewById(R.id.detail_hike_weather);
    detailHikeGroupSize = findViewById(R.id.detail_hike_group_size);
    btnAddObservation = findViewById(R.id.btnAddObservation);
    recyclerViewObservations = findViewById(R.id.recyclerViewObservations);
}

private void populateHikeDetails() {
    detailHikeName.setText(hike.getName());
    detailHikeLocation.setText("Location: " + hike.getLocation());
    detailHikeDate.setText("Date: " + hike.getDate());
    detailHikeParking.setText("Parking Available: " + (hike.getParkingAvailable() == 1 ? "Yes" : "No"));
    detailHikeLength.setText("Length: " + hike.getLength() + " km");
    detailHikeDifficulty.setText("Difficulty: " + hike.getDifficulty());
    detailHikeDescription.setText("Description: " + hike.getDescription());
    detailHikeWeather.setText("Weather: " + hike.getWeather());
    detailHikeGroupSize.setText("Group Size: " + hike.getGroupSize());
}

private void setupObservationList() {
    recyclerViewObservations.setLayoutManager(new LinearLayoutManager(this));
    observationList = new ArrayList<>();
    observationAdapter = new ObservationAdapter(this, observationList, this);
    recyclerViewObservations.setAdapter(observationAdapter);
}

private void setupListeners() {
    btnAddObservation.setOnClickListener(v -> {
        Intent intent = new Intent(HikeDetailActivity.this, AddObservationActivity.class);
        intent.putExtra(AddObservationActivity.EXTRA_HIKE_ID, hikeId);
        startActivity(intent);
    });
}

private void loadObservations() {
    List<Observation> updatedList = databaseHelper.getObservationsForHike(hikeId);
    observationAdapter.updateData(updatedList);
}

@Override
public void onDeleteClick(Observation observation) {
    databaseHelper.deleteObservation(observation.getId());
    Toast.makeText(this, "Observation deleted", Toast.LENGTH_SHORT).show();
    loadObservations();
}
```

```
    @Override
    public void onEditClick(Observation observation) {
        Intent intent = new Intent(this, EditObservationActivity.class);
        intent.putExtra(EditObservationActivity.EXTRA_OBSERVATION_ID, observation.getId());
        startActivity(intent);
    }

    @Override
    public boolean onSupportNavigateUp() {
        finish();
        return true;
    }
}
```

This code file manages the "Hike Detail Screen". Its main function is to display all the information for a specific hike selected from the list screen. It receives the hike's ID, uses that ID to query data from the database, and then populates the information fields like name, location, date, etc.Additionally, this screen displays a list of "Observations" recorded for that hike. The user can add, edit, or delete these observations directly from the detail screen.

The most notable part of the code here is how this Activity manages and interacts with the child list (the list of observations) through an Adapter and an Interface.

1.Loading Data:

- In onCreate, the Activity gets the hikeId passed from the previous screen.
- Based on the hikeId, it retrieves the hike's information (Hike object) and the corresponding list of observations (Observation list) from the DatabaseHelper.

2.Setting up the Observation List:

- **setupObservationList()** initializes the **RecyclerView** and **ObservationAdapter**.
- Crucially, when creating the adapter, it passes this (the Activity itself) to serve as an event listener (**OnObservationListener**).

3.The Listener Pattern:

- **HikeDetailActivity** has the line implements **ObservationAdapter**.**OnObservationListener**. This means the Activity "promises" to handle click events that happen inside the ObservationAdapter.
- Inside the **ObservationAdapter** (a file not shown here), when a user clicks the "Edit" or "Delete" button on an observation item, the adapter doesn't handle it directly. Instead, it calls the **onEditClick**() or **onDeleteClick**() methods on the listener that was provided to it.

4. Since this Activity is that listener, the onEditClick and onDeleteClick methods at the end of the HikeDetailActivity file are the ones that get executed.

- **On Delete**: It calls **databaseHelper**.**deleteObservation**(...) and then reloads the list.
- **On Edit**: It opens the **EditObservationActivity**, passing the ID of the observation to be edited.

6. HikeAdapter.java

```
public class HikeAdapter extends RecyclerView.Adapter<HikeAdapter.HikeViewHolder> {

    private List<Hike> hikeList;
    private Context context;
    private DatabaseHelper databaseHelper;

    public HikeAdapter(Context context, List<Hike> hikeList) {
        this.context = context;
        this.hikeList = hikeList;
        this.databaseHelper = new DatabaseHelper(context);
    }

    @NonNull
    @Override
    public HikeViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.hike_list_item, parent, false);
        return new HikeViewHolder(view);
    }

    @Override
    public void onBindViewHolder(@NonNull HikeViewHolder holder, int position) {
        Hike hike = hikeList.get(position);
        holder.hikeName.setText(hike.getName());
        holder.hikeLocation.setText("Location: " + hike.getLocation());
        holder.hikeDate.setText("Date: " + hike.getDate());

        holder.itemView.setOnClickListener(v -> {
            Intent intent = new Intent(context, HikeDetailActivity.class);
            intent.putExtra(HikeDetailActivity.EXTRA_HIKE_ID, hike.getId());
            context.startActivity(intent);
        });

        holder.btnEdit.setOnClickListener(v -> {
            Intent intent = new Intent(context, AddHikeActivity.class);
            intent.putExtra(AddHikeActivity.EXTRA_HIKE_ID, hike.getId());
            context.startActivity(intent);
        });

        holder.itemView.setOnLongClickListener(v -> {
            PopupMenu popup = new PopupMenu(context, v);
            popup.getMenuInflater().inflate(R.menu.hike_list_item_menu, popup.getMenu());
            popup.setOnMenuItemClickListener(item -> {
                int itemId = item.getItemId();
                if (itemId == R.id.menu_delete) {
                    new AlertDialog.Builder(context)
                            .setTitle("Delete Hike")
                            .setMessage("Are you sure you want to delete this hike?")
                            .setPositiveButton(android.R.string.yes, (dialog, which) -> {
                                databaseHelper.deleteHike(hike.getId());
                                updateData(databaseHelper.getAllHikes());
                            })
                            .setNegativeButton(android.R.string.no, null)
                            .setIcon(android.R.drawable.ic_dialog_alert)
                            .show();
                    return true;
```

```
            } else if (itemId == R.id.menu_delete_all) {
                new AlertDialog.Builder(context)
                        .setTitle("Delete All Hikes")
                        .setMessage("Are you sure you want to delete all hikes?")
                        .setPositiveButton(android.R.string.yes, (dialog, which) -> {
                            databaseHelper.deleteAllHikes();
                            updateData(databaseHelper.getAllHikes());
                        })
                        .setNegativeButton(android.R.string.no, null)
                        .setIcon(android.R.drawable.ic_dialog_alert)
                        .show();
                return true;
            }
            return false;
        });
        popup.show();
        return true;
    });
}

@Override
public int getItemCount() {
    return hikeList.size();
}

public void updateData(List<Hike> newHikeList) {
    this.hikeList.clear();
    this.hikeList.addAll(newHikeList);
    notifyDataSetChanged();
}

static class HikeViewHolder extends RecyclerView.ViewHolder {
    TextView hikeName, hikeLocation, hikeDate;
    Button btnEdit;

    public HikeViewHolder(@NonNull View itemView) {
        super(itemView);
        hikeName = itemView.findViewById(R.id.item_hike_name);
        hikeLocation = itemView.findViewById(R.id.item_hike_location);
        hikeDate = itemView.findViewById(R.id.item_hike_date);
        btnEdit = itemView.findViewById(R.id.btn_edit_hike);
    }
}
}
```

This file is an Adapter for a RecyclerView, a crucial component in Android for displaying lists. The role of the HikeAdapter is to act as a bridge between the data source (a list of Hike objects) and the user interface (the RecyclerView that shows the list of hikes). It takes the data, creates the layout for each row in the list, fills that row with data, and handles user interactions on each row (like clicks, long-presses, or clicking the edit button).

1.HikeViewHolder (Inner Class):

- This class acts as a memory cache. It holds direct references to the UI components of a single row (e.g., **item_hike_name**, **btn_edit_hike**).

- Doing this significantly improves performance because the app doesn't have to find these views with **findViewById** every time the list is scrolled.

2.onCreateViewHolder():

- This method is called by the **RecyclerView** when it needs to create a new row to display.
- It "inflates" the layout from the **hike_list_item.xml** file to create the view for one row, then wraps it inside a HikeViewHolder object and returns it.

3.onBindViewHolder():

- This is the most important method. The **RecyclerView** calls it when it wants to display data at a specific position (row).
- It gets the corresponding Hike object from the data list.
- It populates the Hike's information (name, location, date) into the **TextViews** within the **ViewHolder**.
- It assigns listeners to each row:
    - **setOnClickListener**: When the user taps a row, it opens the HikeDetailActivity, passing along the ID of that hike.
    - **btnEdit.setOnClickListener**: When the user taps the "Edit" button, it opens the **AddHikeActivity** (in edit mode), also passing the hike's ID.
    - **setOnLongClickListener**: When the user presses and holds a row, a small **PopupMenu** appears with two options:
    - **Delete**: It shows an AlertDialog to confirm. If the user agrees, it calls **databaseHelper.deleteHike**() to remove the hike from the database and then calls **updateData**() to refresh the list.
    - **Delete All**: Similarly, it shows a confirmation dialog before deleting all hikes and refreshing the UI.

4.updateData():

This is a helper method. When there is new data (e.g., after a search or a deletion), this method clears the old list, adds all the new data, and then calls **notifyDataSetChanged**(). This call tells the RecyclerView that the data has changed and it needs to redraw the entire list.

7. Observation

```
public class Observation {

    private int id;
    private String observationText;
    private String timeOfObservation;
    private String additionalComments;
    private int hikeId;

    public Observation(int id, String observationText, String timeOfObservation, String additionalComments,
int hikeId) {
        this.id = id;
        this.observationText = observationText;
        this.timeOfObservation = timeOfObservation;
```

```
        this.additionalComments = additionalComments;
        this.hikeId = hikeId;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getObservationText() {
        return observationText;
    }

    public void setObservationText(String observationText) {
        this.observationText = observationText;
    }

    public String getTimeOfObservation() {
        return timeOfObservation;
    }

    public void setTimeOfObservation(String timeOfObservation) {
        this.timeOfObservation = timeOfObservation;
    }

    public String getAdditionalComments() {
        return additionalComments;
    }

    public void setAdditionalComments(String additionalComments) {
        this.additionalComments = additionalComments;
    }

    public int getHikeId() {
        return hikeId;
    }

    public void setHikeId(int hikeId) {
        this.hikeId = hikeId;
    }
}
```

Its main purpose is to define the structure for an "Observation" object in your app. It acts as a "blueprint" to store all the information related to a specific observation, such as the observation text, the time it was made, any additional comments, and the ID of the hike (hikeId) it belongs to.Similar to the Hike class, this class doesn't contain complex logic. It's simply an organized way to hold and transport data about observations within your application.

8. Observation adpter

```
public class ObservationAdapter extends
RecyclerView.Adapter<ObservationAdapter.ObservationViewHolder> {

    private List<Observation> observationList;
    private Context context;
    private OnObservationListener listener;

    public interface OnObservationListener {
        void onDeleteClick(Observation observation);
        void onEditClick(Observation observation);
    }

    public ObservationAdapter(Context context, List<Observation> observationList, OnObservationListener
listener) {
        this.context = context;
        this.observationList = observationList;
        this.listener = listener;
    }

    @NonNull
    @Override
    public ObservationViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(context).inflate(R.layout.observation_list_item, parent, false);
        return new ObservationViewHolder(view);
    }

    @Override
    public void onBindViewHolder(@NonNull ObservationViewHolder holder, int position) {
        Observation observation = observationList.get(position);
        holder.observationText.setText(observation.getObservationText());
        holder.observationTime.setText("Time: " + observation.getTimeOfObservation());

        String comment = observation.getAdditionalComments();
        if (comment != null && !comment.isEmpty()) {
            holder.observationComment.setText("Comment: " + comment);
            holder.observationComment.setVisibility(View.VISIBLE);
        } else {
            holder.observationComment.setVisibility(View.GONE);
        }

        holder.editButton.setOnClickListener(v -> {
            if (listener != null) {
                listener.onEditClick(observation);
            }
        });

        holder.deleteButton.setOnClickListener(v -> {
            new AlertDialog.Builder(context)
                    .setTitle("Delete Observation")
                    .setMessage("Are you sure you want to delete this observation?")
                    .setPositiveButton("Yes", (dialog, which) -> {
                        if (listener != null) {
                            listener.onDeleteClick(observation);
                        }
                    })
                    .setNegativeButton("No", null)
                    .show();
```

```
        });
    }

    @Override
    public int getItemCount() {
        return observationList.size();
    }

    public void updateData(List<Observation> newObservationList) {
        this.observationList.clear();
        this.observationList.addAll(newObservationList);
        notifyDataSetChanged();
    }

    static class ObservationViewHolder extends RecyclerView.ViewHolder {
        TextView observationText, observationTime, observationComment;
        ImageButton editButton, deleteButton;

        public ObservationViewHolder(@NonNull View itemView) {
            super(itemView);
            observationText = itemView.findViewById(R.id.item_observation_text);
            observationTime = itemView.findViewById(R.id.item_observation_time);
            observationComment = itemView.findViewById(R.id.item_observation_comment);
            editButton = itemView.findViewById(R.id.btnEditObservation);
            deleteButton = itemView.findViewById(R.id.btnDeleteObservation);
        }
    }
}
```

The ObservationAdapter class is a custom adapter used in Android to bind a list of data objects (specifically Observation objects) to a RecyclerView. It acts as a bridge between your data source (the list of observations) and the UI that displays each item in the list.

1. Class Members and Interface The adapter maintains a list of observations (**observationList**), a Context reference for inflating layouts and creating dialogs, and a listener interface (**OnObservationListener**). This interface defines two methods, onDeleteClick and onEditClick, which allow the adapter to communicate user actions (like clicking the edit or delete buttons) back to the parent Activity or Fragment without handling the business logic itself.
2. Constructor The constructor initializes the adapter with the necessary context, the data list, and the listener instance. This setup ensures the adapter has everything it needs to function immediately upon creation.
3. **onCreateViewHolder** Method This method is responsible for creating the visual structure of a single list item. It inflates the XML layout file (**R.layout.observation_list_item**) into a View object and passes it to a new instance of **ObservationViewHolder**. This process only happens enough times to fill the screen (plus a buffer), making it memory efficient.
4. **onBindViewHolder** Method This is where the actual data binding happens. For each item in the list, this method:

- Retrieves the specific Observation object based on the current position.

- Sets the text for the observation details and timestamp into their respective TextViews.
- Handles Logic: It checks if a comment exists. If additionalComments is not null or empty, it displays the comment TextView; otherwise, it sets the visibility to GONE to hide it and save space.
- Sets Click Listeners: It assigns actions to the "Edit" and "Delete" buttons. The Delete button, in particular, creates an AlertDialog to ask for user confirmation before triggering the onDeleteClick callback.

5. Helper Methods and Inner Class

- **getItemCount()** returns the total number of items in the list so the RecyclerView knows how many rows to draw.
- **updateData()** is a utility method that clears the existing list, adds new data, and calls **notifyDataSetChanged()** to refresh the UI dynamically (useful for search filters or data updates).
- The **ObservationViewHolder** inner class finds and holds references to the UI components (TextViews and Buttons) defined in the XML layout. This prevents the app from having to expensive lookup operations (findViewById) every time the list scrolls.

9. AddObservationActivity.java

```
public class AddObservationActivity extends AppCompatActivity {

    public static final String EXTRA_HIKE_ID = "extra_hike_id";

    private EditText edtObservationText, edtObservationComment;
    private TextView txtObservationTime;
    private Button btnSaveObservation;

    private DatabaseHelper databaseHelper;
    private int hikeId;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_add_observation);

        if (getSupportActionBar() != null) {
            getSupportActionBar().setDisplayHomeAsUpEnabled(true);
            getSupportActionBar().setTitle("Add Observation");
        }

        databaseHelper = new DatabaseHelper(this);
        hikeId = getIntent().getIntExtra(EXTRA_HIKE_ID, -1);

        initViews();
        setDefaultTime();

        btnSaveObservation.setOnClickListener(v -> saveObservation());
    }
```

```
    private void initViews() {
        edtObservationText = findViewById(R.id.edtObservationText);
        txtObservationTime = findViewById(R.id.txtObservationTime);
        edtObservationComment = findViewById(R.id.edtObservationComment);
        btnSaveObservation = findViewById(R.id.btnSaveObservation);
    }

    private void setDefaultTime() {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss", Locale.getDefault());
        String currentTime = sdf.format(new Date());
        txtObservationTime.setText("Time: " + currentTime);
    }

    private void saveObservation() {
        String observationText = edtObservationText.getText().toString().trim();
        String observationTime = txtObservationTime.getText().toString().replace("Time: ", "").trim();
        String comment = edtObservationComment.getText().toString().trim();

        if (TextUtils.isEmpty(observationText)) {
            edtObservationText.setError("Observation cannot be empty.");
            return;
        }

        if (hikeId == -1) {
            Toast.makeText(this, "Error: Hike ID is missing.", Toast.LENGTH_SHORT).show();
            return;
        }

        databaseHelper.addObservation(hikeId, observationText, observationTime, comment);

        Toast.makeText(this, "Observation saved!", Toast.LENGTH_SHORT).show();
        finish();    }

    @Override
    public boolean onSupportNavigateUp() {
        finish();
        return true;
    }
}
```

The **AddObservationActivity.java** file serves as the interface screen that allows users to create and save a new "Observation" linked to a specific hike. When launched, this Activity receives the hike's identifier (**hikeId**) from the previous screen to ensure the new data is correctly associated in the database. For user convenience, the system automatically retrieves the actual current time when the app is opened and pre-fills the time field using the **setDefaultTime** function, so the user only needs to input the observation content and any notes (if applicable). The most critical function lies in the "Save" button: when activated, it validates whether the user has entered information, then calls the **DatabaseHelper** to permanently store the data on the device and automatically closes this screen to return to the detail list.

10. EditObservationActivity.java

```
public class EditObservationActivity extends AppCompatActivity {

    public static final String EXTRA_OBSERVATION_ID = "extra_observation_id";

    private EditText edtEditObservationText, edtEditObservationComment;
    private TextView txtEditObservationTime;
    private Button btnUpdateObservation;

    private DatabaseHelper databaseHelper;
    private Observation currentObservation;
    private int observationId;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_edit_observation);

        if (getSupportActionBar() != null) {
            getSupportActionBar().setDisplayHomeAsUpEnabled(true);
            getSupportActionBar().setTitle("Edit Observation");
        }

        databaseHelper = new DatabaseHelper(this);
        observationId = getIntent().getIntExtra(EXTRA_OBSERVATION_ID, -1);

        initViews();

        if (observationId != -1) {
            currentObservation = databaseHelper.getObservationById(observationId);
            if (currentObservation != null) {
                populateData();
            } else {
                Toast.makeText(this, "Error: Observation not found.", Toast.LENGTH_SHORT).show();
                finish();
            }
        } else {
            Toast.makeText(this, "Error: Invalid Observation ID.", Toast.LENGTH_SHORT).show();
            finish();
        }

        btnUpdateObservation.setOnClickListener(v -> updateObservation());
    }

    private void initViews() {
        edtEditObservationText = findViewById(R.id.edtEditObservationText);
        txtEditObservationTime = findViewById(R.id.txtEditObservationTime);
        edtEditObservationComment = findViewById(R.id.edtEditObservationComment);
        btnUpdateObservation = findViewById(R.id.btnUpdateObservation);
    }

    private void populateData() {
        edtEditObservationText.setText(currentObservation.getObservationText());
        txtEditObservationTime.setText("Time: " + currentObservation.getTimeOfObservation());
        edtEditObservationComment.setText(currentObservation.getAdditionalComments());
    }

    private void updateObservation() {
```

```
        String newObservationText = edtEditObservationText.getText().toString().trim();
        String newComment = edtEditObservationComment.getText().toString().trim();

        if (TextUtils.isEmpty(newObservationText)) {
            edtEditObservationText.setError("Observation cannot be empty.");
            return;
        }

        databaseHelper.updateObservation(observationId, newObservationText, newComment);
        Toast.makeText(this, "Observation updated!", Toast.LENGTH_SHORT).show();
        finish();
    }

    @Override
    public boolean onSupportNavigateUp() {
        finish();
        return true;
    }
}
```

1. Purpose and Setup (**onCreate**) The **EditObservationActivity** is designed to handle the modification of an existing observation record.

- **Initialization**: Upon creation, the activity initializes the UI components and the **DatabaseHelper**. It also sets up the action bar with a back button and a title ("Edit Observation").
- **Retrieving Data**: The critical step here is retrieving the **EXTRA_OBSERVATION_ID** passed via the Intent.
- **Error Handling**: It validates this ID immediately. If the ID is invalid (-1) or if the corresponding observation cannot be found in the database **(databaseHelper.getObservationById**), the activity displays an error Toast ("Error: Observation not found") and closes itself (**finish()**) to prevent a crash or empty UI.

2. **Populating the UI (populateData)** Once a valid observation is found, the **populateData()** method is called. This method bridges the database object with the user interface:

- It sets the text of **edtEditObservationText** to the existing observation name.
- It sets the **edtEditObservationComment** to the existing notes.
- It displays the timestamp in **txtEditObservationTime**. Notably, the timestamp is displayed in a **TextView** (not an EditText), implying that the creation time is a historical record and cannot be edited by the user.

3. Handling Updates (**updateObservation**) This method contains the logic executed when the user clicks the "Update" button:

- **Input Retrieval**: It extracts the current strings from the observation text and comment input fields.
- **Validation**: It uses TextUtils.isEmpty to ensure the observation name is not blank. If it is, an error is shown on the input field, and the save process stops.
- **Database Operation**: If the input is valid, it calls **databaseHelper**.**updateObservation**() passing the original ID along with the new text and comment.

- **Completion**: Finally, it shows a success message ("Observation updated!") and calls **finish**() to close the activity and return the user to the previous screen.

4. **Navigation** (onSupportNavigateUp) This method overrides the default behavior of the back arrow in the toolbar (Action Bar). It ensures that when the user clicks the top-left arrow, the activity finishes (**finish()**) and properly navigates back to the parent activity, just like the hardware back button.

11. DatabaseHelper.

```
public class DatabaseHelper extends SQLiteOpenHelper {

    private static final String DATABASE_NAME = "hike_db";
    private static final int DATABASE_VERSION = 2;

    // Table Names
    public static final String TABLE_HIKES = "hikes";
    public static final String TABLE_OBSERVATIONS = "observations";


    public static final String KEY_HIKE_ID = "id";
    public static final String KEY_HIKE_NAME = "name";
    public static final String KEY_HIKE_LOCATION = "location";
    public static final String KEY_HIKE_DATE = "date";
    public static final String KEY_HIKE_PARKING = "parking_available";
    public static final String KEY_HIKE_LENGTH = "length";
    public static final String KEY_HIKE_DIFFICULTY = "difficulty";
    public static final String KEY_HIKE_DESCRIPTION = "description";
    public static final String KEY_HIKE_WEATHER = "weather";
    public static final String KEY_HIKE_GROUP_SIZE = "group_size";


    public static final String KEY_OBSERVATION_ID = "id";
    public static final String KEY_OBSERVATION_TEXT = "observation_text";
    public static final String KEY_OBSERVATION_TIME = "time_of_observation";
    public static final String KEY_OBSERVATION_COMMENT = "additional_comments";
    public static final String KEY_OBSERVATION_HIKE_ID = "hike_id";


    private static final String CREATE_TABLE_HIKES = "CREATE TABLE " + TABLE_HIKES + "(" +
            KEY_HIKE_ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
            KEY_HIKE_NAME + " TEXT NOT NULL," +
            KEY_HIKE_LOCATION + " TEXT NOT NULL," +
            KEY_HIKE_DATE + " TEXT NOT NULL," +
            KEY_HIKE_PARKING + " INTEGER NOT NULL," +
            KEY_HIKE_LENGTH + " REAL NOT NULL," +
            KEY_HIKE_DIFFICULTY + " TEXT NOT NULL," +
            KEY_HIKE_DESCRIPTION + " TEXT," +
            KEY_HIKE_WEATHER + " TEXT," +
            KEY_HIKE_GROUP_SIZE + " INTEGER)";


    private static final String CREATE_TABLE_OBSERVATIONS = "CREATE TABLE " + TABLE_OBSERVATIONS +
"(" +
            KEY_OBSERVATION_ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
            KEY_OBSERVATION_TEXT + " TEXT NOT NULL," +
```

```
        KEY_OBSERVATION_TIME + " TEXT NOT NULL," +
        KEY_OBSERVATION_COMMENT + " TEXT," +
        KEY_OBSERVATION_HIKE_ID + " INTEGER NOT NULL," +
        "FOREIGN KEY(" + KEY_OBSERVATION_HIKE_ID + ") REFERENCES " + TABLE_HIKES + "(" +
KEY_HIKE_ID + ") ON DELETE CASCADE)";

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

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_TABLE_HIKES);
        db.execSQL(CREATE_TABLE_OBSERVATIONS);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // A simple upgrade policy is to drop tables and recreate them.
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_OBSERVATIONS);
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_HIKES);
        onCreate(db);
    }

    public void addHike(String name, String location, String date, int parking, double length, String difficulty,
String description, String weather, int groupSize) {
        SQLiteDatabase db = getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(KEY_HIKE_NAME, name);
        values.put(KEY_HIKE_LOCATION, location);
        values.put(KEY_HIKE_DATE, date);
        values.put(KEY_HIKE_PARKING, parking);
        values.put(KEY_HIKE_LENGTH, length);
        values.put(KEY_HIKE_DIFFICULTY, difficulty);
        values.put(KEY_HIKE_DESCRIPTION, description);
        values.put(KEY_HIKE_WEATHER, weather);
        values.put(KEY_HIKE_GROUP_SIZE, groupSize);
        db.insert(TABLE_HIKES, null, values);
        db.close();
    }

    public int updateHike(int id, String name, String location, String date, int parking, double length, String
difficulty, String description, String weather, int groupSize) {
        SQLiteDatabase db = getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(KEY_HIKE_NAME, name);
        values.put(KEY_HIKE_LOCATION, location);
        values.put(KEY_HIKE_DATE, date);
        values.put(KEY_HIKE_PARKING, parking);
        values.put(KEY_HIKE_LENGTH, length);
        values.put(KEY_HIKE_DIFFICULTY, difficulty);
        values.put(KEY_HIKE_DESCRIPTION, description);
        values.put(KEY_HIKE_WEATHER, weather);
        values.put(KEY_HIKE_GROUP_SIZE, groupSize);

        return db.update(TABLE_HIKES, values, KEY_HIKE_ID + " = ?", new String[]{String.valueOf(id)});
    }
```

```
    public List<Hike> getAllHikes() {
        List<Hike> hikeList = new ArrayList<>();
        SQLiteDatabase db = getReadableDatabase();
        Cursor cursor = db.rawQuery("SELECT * FROM " + TABLE_HIKES, null);
        if (cursor.moveToFirst()) {
            do {
                Hike hike = new Hike(
                        cursor.getInt(cursor.getColumnIndexOrThrow(KEY_HIKE_ID)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_NAME)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_LOCATION)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_DATE)),
                        cursor.getInt(cursor.getColumnIndexOrThrow(KEY_HIKE_PARKING)),
                        cursor.getDouble(cursor.getColumnIndexOrThrow(KEY_HIKE_LENGTH)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_DIFFICULTY)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_DESCRIPTION)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_WEATHER)),
                        cursor.getInt(cursor.getColumnIndexOrThrow(KEY_HIKE_GROUP_SIZE))
                );
                hikeList.add(hike);
            } while (cursor.moveToNext());
        }
        cursor.close();
        db.close();
        return hikeList;
    }

    public List<Hike> searchHikes(String query) {
        List<Hike> hikeList = new ArrayList<>();
        SQLiteDatabase db = getReadableDatabase();

        String selection = KEY_HIKE_NAME + " LIKE ? OR " +
                           KEY_HIKE_LOCATION + " LIKE ? OR " +
                           KEY_HIKE_DATE + " LIKE ? OR " +
                           "CAST(" + KEY_HIKE_LENGTH + " AS TEXT) LIKE ?";
        String[] selectionArgs = new String[]{"%" + query + "%", "%" + query + "%", "%" + query + "%", "%" +
query + "%"};

        Cursor cursor = db.query(TABLE_HIKES, null, selection, selectionArgs, null, null, null);

        if (cursor.moveToFirst()) {
            do {
                Hike hike = new Hike(
                        cursor.getInt(cursor.getColumnIndexOrThrow(KEY_HIKE_ID)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_NAME)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_LOCATION)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_DATE)),
                        cursor.getInt(cursor.getColumnIndexOrThrow(KEY_HIKE_PARKING)),
                        cursor.getDouble(cursor.getColumnIndexOrThrow(KEY_HIKE_LENGTH)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_DIFFICULTY)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_DESCRIPTION)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_WEATHER)),
                        cursor.getInt(cursor.getColumnIndexOrThrow(KEY_HIKE_GROUP_SIZE))
                );
                hikeList.add(hike);
            } while (cursor.moveToNext());
        }
```

```
        cursor.close();
        db.close();
        return hikeList;
    }

    public Hike getHikeById(int hikeId) {
        SQLiteDatabase db = getReadableDatabase();
        Cursor cursor = db.query(TABLE_HIKES, null, KEY_HIKE_ID + "=?", new String[]{String.valueOf(hikeId)},
null, null, null, null);
        Hike hike = null;
        if (cursor != null && cursor.moveToFirst()) {
            hike = new Hike(
                    cursor.getInt(cursor.getColumnIndexOrThrow(KEY_HIKE_ID)),
                    cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_NAME)),
                    // ... (populate other fields)
                    cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_LOCATION)),
                    cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_DATE)),
                    cursor.getInt(cursor.getColumnIndexOrThrow(KEY_HIKE_PARKING)),
                    cursor.getDouble(cursor.getColumnIndexOrThrow(KEY_HIKE_LENGTH)),
                    cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_DIFFICULTY)),
                    cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_DESCRIPTION)),
                    cursor.getString(cursor.getColumnIndexOrThrow(KEY_HIKE_WEATHER)),
                    cursor.getInt(cursor.getColumnIndexOrThrow(KEY_HIKE_GROUP_SIZE))
            );
            cursor.close();
        }
        db.close();
        return hike;
    }

    public void deleteHike(int hikeId) {
        SQLiteDatabase db = getWritableDatabase();
        db.delete(TABLE_HIKES, KEY_HIKE_ID + " = ?", new String[]{String.valueOf(hikeId)});
        db.close();
    }

    public void deleteAllHikes() {
        SQLiteDatabase db = getWritableDatabase();
        db.delete(TABLE_HIKES, null, null);
        db.close();
    }

    public void addObservation(int hikeId, String observation, String time, String comment) {
        SQLiteDatabase db = getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(KEY_OBSERVATION_HIKE_ID, hikeId);
        values.put(KEY_OBSERVATION_TEXT, observation);
        values.put(KEY_OBSERVATION_TIME, time);
        values.put(KEY_OBSERVATION_COMMENT, comment);
        db.insert(TABLE_OBSERVATIONS, null, values);
        db.close();
    }

    public List<Observation> getObservationsForHike(int hikeId) {
        List<Observation> observationList = new ArrayList<>();
        SQLiteDatabase db = getReadableDatabase();
        Cursor cursor = db.query(TABLE_OBSERVATIONS, null, KEY_OBSERVATION_HIKE_ID + "=?", new
```

```
String[]{String.valueOf(hikeId)}, null, null, null, null);
        if (cursor.moveToFirst()) {
            do {
                observationList.add(new Observation(
                        cursor.getInt(cursor.getColumnIndexOrThrow(KEY_OBSERVATION_ID)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_OBSERVATION_TEXT)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_OBSERVATION_TIME)),
                        cursor.getString(cursor.getColumnIndexOrThrow(KEY_OBSERVATION_COMMENT)),
                        cursor.getInt(cursor.getColumnIndexOrThrow(KEY_OBSERVATION_HIKE_ID))
                ));
            } while (cursor.moveToNext());
        }
        cursor.close();
        db.close();
        return observationList;
    }

    public Observation getObservationById(int observationId) {
        SQLiteDatabase db = getReadableDatabase();
        Cursor cursor = db.query(TABLE_OBSERVATIONS, null, KEY_OBSERVATION_ID + "=?", new
String[]{String.valueOf(observationId)}, null, null, null, null);
        Observation observation = null;
        if (cursor != null && cursor.moveToFirst()) {
            observation = new Observation(
                    cursor.getInt(cursor.getColumnIndexOrThrow(KEY_OBSERVATION_ID)),
                    cursor.getString(cursor.getColumnIndexOrThrow(KEY_OBSERVATION_TEXT)),
                    cursor.getString(cursor.getColumnIndexOrThrow(KEY_OBSERVATION_TIME)),
                    cursor.getString(cursor.getColumnIndexOrThrow(KEY_OBSERVATION_COMMENT)),
                    cursor.getInt(cursor.getColumnIndexOrThrow(KEY_OBSERVATION_HIKE_ID))
            );
            cursor.close();
        }
        db.close();
        return observation;
    }

    public int updateObservation(int observationId, String newObservationText, String newComment) {
        SQLiteDatabase db = getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(KEY_OBSERVATION_TEXT, newObservationText);
        values.put(KEY_OBSERVATION_COMMENT, newComment);
        int rowsAffected = db.update(TABLE_OBSERVATIONS, values, KEY_OBSERVATION_ID + " = ?", new
String[]{String.valueOf(observationId)});
        db.close();
        return rowsAffected;
    }

    public void deleteObservation(int observationId) {
        SQLiteDatabase db = getWritableDatabase();
        db.delete(TABLE_OBSERVATIONS, KEY_OBSERVATION_ID + " = ?", new
String[]{String.valueOf(observationId)});
        db.close();
    }
}
```

1. Table and Column Definitions (Constants) At the beginning of the file, you will see many static final variables. These define the names of the tables and columns in the database to prevent typos when writing SQL queries later.

- **TABLE_HIKES**: The table storing hike information.
- Columns: id, name, location, date, parking, length, difficulty, etc.
- **TABLE_OBSERVATIONS**: The table storing observations.
- Columns: **id, observation_text, time, comment, hike_id.**

2. Table Creation (onCreate) This is where the initial data structure is defined.

- **CREATE_TABLE_HIKES**: Creates the table to store hikes.
- **CREATE_TABLE_OBSERVATIONS**: Creates the table to store observations.

Important Note: There is a **FOREIGN KEY(hike_id) REFERENCES hikes(id) ON DELETE CASCADE** clause. This means the Observation table is linked to the Hike table. **ON DELETE CASCADE** ensures that if you delete a Hike, all Observations belonging to that hike are automatically deleted as well (to clean up data).

3. Managing HIKE Data

- **addHike(...):** Accepts all details of a hike (name, location, date, etc.) and uses ContentValues to insert a new row into the hikes table.
- **getAllHikes():** Retrieves all hikes from the database to display on the main list screen. It uses a Cursor to iterate through each result row and convert it into a Hike object.
- **searchHikes(String query):** Searches for hikes. It uses the LIKE clause to find hikes where the name, location, or date matches the keyword (query) entered by the user.
- **updateHike(...):** Updates the information of an existing hike based on its id.
- **deleteHike(int hikeId):** Deletes a specific hike.
- **deleteAllHikes():** Wipes all hike data completely (often used for a "Reset" feature or during debugging).

4. Managing OBSERVATION Data

- **addObservation(...):** Adds a new observation. It requires the hikeId to know which hike this observation belongs to.
- **getObservationsForHike(int hikeId):** Retrieves a list of all observations belonging to a specific hike (using the **WHERE hike_id = ? clause**).
- **updateObservation(...):** Modifies the content or comments of an existing observation.
- **deleteObservation(...):** Deletes a specific observation.

6. Declaration of AI Use

I have used AI while undertaking my assignment in the following ways:
To develop research questions on the topic – YES

To create an outline of the topic – NO
To explain concepts – YES
To support my use of language – NO