APPENDIX

CHAPTER 1: CREATING YOUR FIRST APP

ACTIVITY 1.01: PRODUCING AN APP TO CREATE RGB COLORS

Solution:

1. Create a new project called **`Colors`**. Start by creating a new **`Empty Activity`** project (**`File`** | **`New`** | **`New Project`** | **`Empty Activity`**). Name your application **`Colors`** and leave everything else with its default values and click **`Finish`**.
2. You need to add all the resource values you need that are not added by default here. The **`strings.xml`** file is needed to display all the text displayed in the app:

```
<resources>
    <string name="app_name">Colors</string>

    <string name="color_creator_title">Create an RGB Color</string>
    <!--Escape special characters by placing
          a backslash before them-->
    <string name="color_creator_description">
      Add two hexadecimal characters between 0-9, A-F
      or a-f without the \'#\' for each channel </string>
    <string name="red_channel">Red Channel</string>
    <string name="green_channel">Green Channel</string>
    <string name="blue_channel">Blue Channel</string>
    <string name="color_creator_button_text">Create RGB Color
    </string>
    <string name="color_created_display_panel">
      Created color display panel</string>
    <string name="invalid_characters_found">Invalid Characters
      Found</string>

</resources>
```

3. This **`dimens.xml`** file specifies dimension units in **`dp`** (density-independent pixels) used in the layout. This file is not present when initially creating a project, but it can be added by creating the file dimens.xml in the same 'values' folder.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="color_creator_layout_margin">8dp</dimen>
    <dimen name="color_creator_display_panel">40dp</dimen>
</resources>
```

4. The **`themes.xml`** file is updated to include the style of the screen title.

```
    <style name="color_creator_title"
      parent="TextAppearance.MaterialComponents.Headline5">
        <item name="android:textStyle">bold</item>
    </style>
```

5. You have now set up the resources that will be used to customize the layout and UI of the app. Now, add a title constrained to the top of the layout. Here, you have to go into **`activity_main.xml`** and update the following so the title is constrained to the top using **`app:layout_constraintTop_toTopOf="parent"`**:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    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:layout_margin="@dimen/color_creator_layout_margin"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="match_parent"
        android:id="@+id/color_creator_title"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/color_creator_layout_margin"
        style="@style/color_creator_title"
```

```
            android:gravity="center"
            android:text="@string/color_creator_title"
            app:layout_constraintTop_toTopOf="parent"
            app:layout_constraintStart_toStartOf="parent"/>

    </androidx.constraintlayout.widget.ConstraintLayout>
```

6. You have now created the title that will be displayed on screen. Add a brief description to the user on how to complete the form. You constrain the description to below the title, and then add style, dimensions, and text:

```
    <?xml version="1.0" encoding="utf-8"?>
    <androidx.constraintlayout.widget.ConstraintLayout
        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:layout_margin="@dimen/color_creator_layout_margin"
        tools:context=".MainActivity">

        <TextView
            android:layout_width="match_parent"
            android:id="@+id/color_creator_title"
            android:layout_height="wrap_content"
            android:layout_margin="@dimen/color_creator_layout_margin"
            style="@style/color_creator_title"
            android:gravity="center"
            android:text="@string/color_creator_title"
            app:layout_constraintTop_toTopOf="parent"
            app:layout_constraintStart_toStartOf="parent"/>

        <TextView
            android:layout_width="match_parent"
            android:id="@+id/color_creator_description"
            android:layout_height="wrap_content"
            android:layout_margin="@dimen/color_creator_layout_margin"
            style="@style/TextAppearance.MaterialComponents.Body1"
            android:text="@string/color_creator_description"
```

```
            app:layout_constraintTop_toBottomOf="@id/color_creator_title"
            app:layout_constraintStart_toStartOf="parent"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

You will notice that styles, dimensions, and text have been added using the preceding resources:

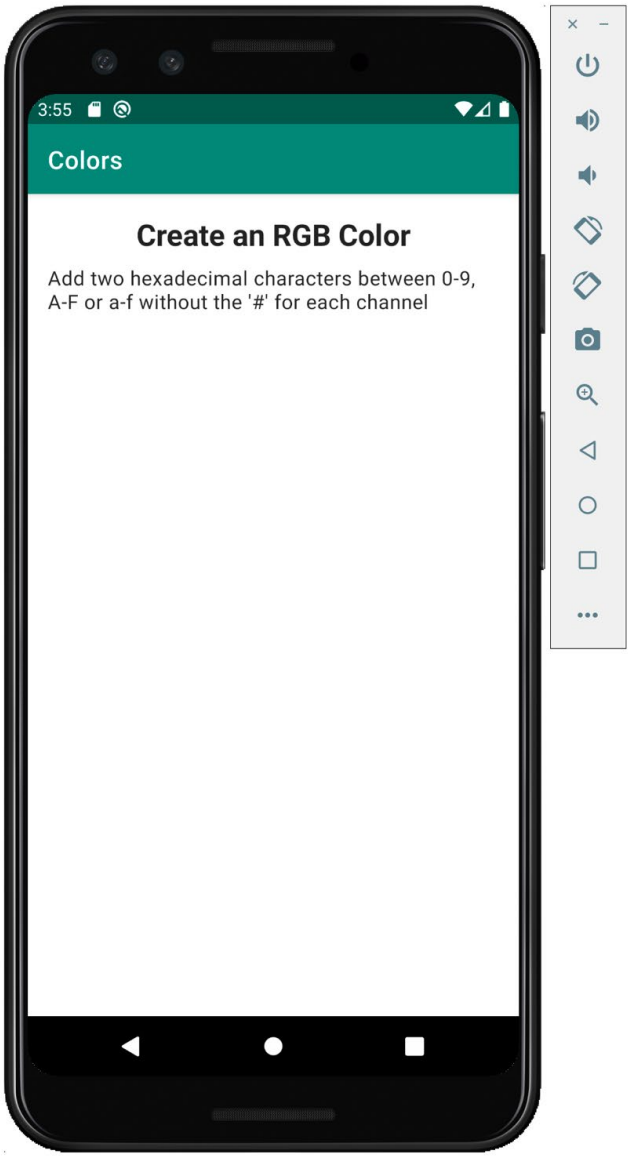

Figure 1.26: Layout with title and description

7. The layout is taking shape now that you've added the description. Continue by adding three material **`TextInputLayout`** fields wrapping three **`TextInputEditText`** fields that appear under **`Title`**. These should be constrained so that each view is on top of the other (rather than to the side). Name the **`TextInputEditText`** fields **`Red Channel`**, **`Green Channel`**, and **`Blue Channel`**, respectively and add some restriction to each field to only be able to enter two hexadecimal characters. These views are similar to what you have worked with before in the exercises, the only difference being that they have the **`digits`** and **`maxLength`** attributes:

```
        <com.google.android.material.textfield.TextInputLayout
            android:id="@+id/red_channel_wrapper"
            style="@style/Widget.MaterialComponents.TextInputLayout
              .OutlinedBox"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_margin="@dimen/color_creator_layout_margin"
            android:hint="@string/red_channel"
            app:layout_constraintTop_toBottomOf
              ="@id/color_creator_description"
            app:layout_constraintStart_toStartOf="parent">

            <com.google.android.material.textfield.TextInputEditText
                android:id="@+id/red_channel"
                android:inputType="textCapCharacters"
                android:digits="ABCDEFabcdef0123456789"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:maxLength="2" />

        </com.google.android.material.textfield.TextInputLayout>

        <com.google.android.material.textfield.TextInputLayout
            android:id="@+id/green_channel_wrapper"
            style="@style/Widget.MaterialComponents.TextInputLayout
              .OutlinedBox"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_margin="@dimen/color_creator_layout_margin"
            android:hint="@string/green_channel"
```

```
        app:layout_constraintTop_toBottomOf="@id/red_channel_wrapper"
        app:layout_constraintStart_toStartOf="parent">

        <com.google.android.material.textfield.TextInputEditText
            android:id="@+id/green_channel"
            android:inputType="textCapCharacters"
            android:digits="ABCDEFabcdef0123456789"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:maxLength="2"/>

    </com.google.android.material.textfield.TextInputLayout>

    <com.google.android.material.textfield.TextInputLayout
        android:id="@+id/blue_channel_wrapper"
        style="@style/Widget.MaterialComponents.TextInputLayout
          .OutlinedBox"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/color_creator_layout_margin"
        android:hint="@string/blue_channel"
        app:layout_constraintTop_toBottomOf=
          "@id/green_channel_wrapper"
        app:layout_constraintStart_toStartOf="parent">

        <com.google.android.material.textfield.TextInputEditText
            android:id="@+id/blue_channel"
            android:inputType="textCapCharacters"
            android:digits="ABCDEFabcdef0123456789"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:maxLength="2" />

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

Once you add this code, the output will be as follows:

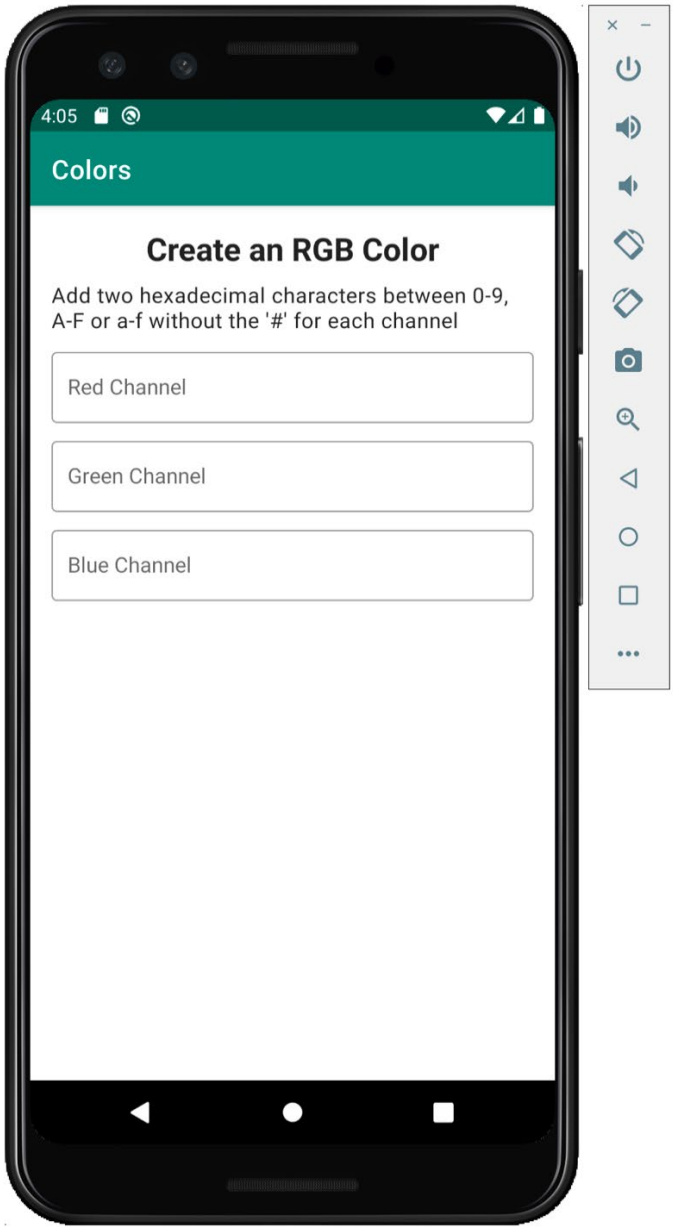

Figure 1.27: Color channel EditText fields added

8. You've now added all the input fields shown in *Figure 1.28*, but now you need to create a button to process these inputs. Add a button that takes the inputs from the three color fields. It's the **`id`** that is important here as it's used to trigger retrieval of the values from the color fields:

```
<com.google.android.material.button.MaterialButton
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="@dimen/color_creator_layout_margin"
    android:id="@+id/color_creator_button"
    android:gravity="center"
    android:text="@string/color_creator_button_text"
    app:layout_constraintTop_toBottomOf="@id/blue_channel_wrapper"
    app:layout_constraintStart_toStartOf="parent"/>
```

9. Add a **`View`**, which will display the produced color in the layout. We need to constrain this view appropriately and make it sufficiently large such that the newly created color will be seen clearly using the dimensions specified previously:

```
<TextView
    android:id="@+id/color_creator_display"
    android:layout_width="match_parent"
    style="@style/TextAppearance.MaterialComponents.Body1"
    android:text="@string/color_created_display_panel"
    android:gravity="center"
    android:layout_height="@dimen/color_creator_display_panel"
    android:layout_margin="@dimen/color_creator_layout_margin"
    app:layout_constraintTop_toBottomOf="@id/color_creator_button"
    app:layout_constraintStart_toStartOf="parent" />
```

10. Finally, display the RGB color created from the three channels in the layout. This is where we need to set the click listener on the button and retrieve the three color values. Then, we need to concatenate these values in the correct order in order to create a new color and set that to the background of the color display panel:

```
package com.example.colors

import android.graphics.Color
import android.os.Bundle
import android.widget.Button
```

```
import android.widget.TextView
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import com.google.android.material.textfield.TextInputEditText

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

        findViewById<Button>(R.id.color_creator_button)
          ?.setOnClickListener {
            var redChannelText = findViewById<TextInputEditText>
              (R.id.red_channel)?.text.toString()
            var greenChannelText = findViewById<TextInputEditText>
              (R.id.green_channel)?.text.toString()
            var blueChannelText = findViewById<TextInputEditText>
              (R.id.blue_channel)?.text.toString()

            //Check that all fields are filled in
            //and show error message if not.
            if (redChannelText.isEmpty() or
              greenChannelText.isEmpty()
              or blueChannelText.isEmpty()) {
            Toast.makeText(this, "All Values are required",
              Toast.LENGTH_LONG).show()
            } else {
                    //check that 2 hexadecimal characters
                    //have been entered and if not
                    //add the same hexadecimal character again.
                if (redChannelText.length == 1) redChannelText =
                  redChannelText.plus(redChannelText)
                if (greenChannelText.length == 1) greenChannelText =
                  greenChannelText.plus(greenChannelText)
                if (blueChannelText.length == 1) blueChannelText =
                  blueChannelText.plus(blueChannelText)

                val colorToDisplay = redChannelText
                  .plus(greenChannelText).plus(blueChannelText)

                val colorAsInt = Color.parseColor
                  ("#".plus(colorToDisplay))
                findViewById<TextView>(R.id.color_creator_display)
                  ?.setBackgroundColor(colorAsInt)
```

```
                    }
                }
            }
        }
```

Your solution should appear something like the app displayed here. The similarity to the exact layout that follows is not important. This is just to give you an indication of the layout you are aiming for:

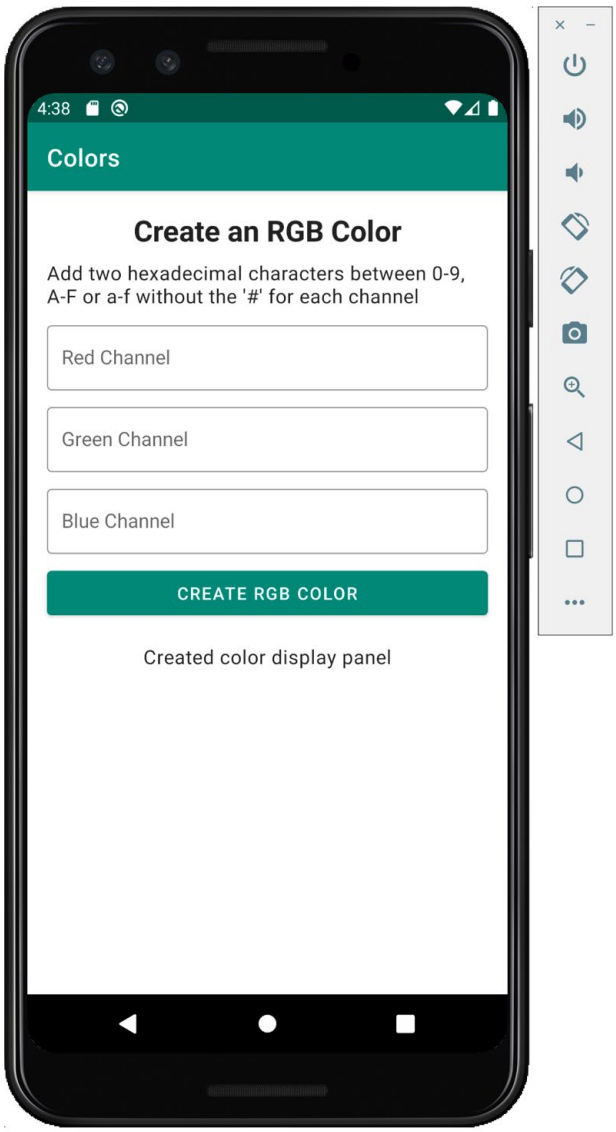

Figure 1.28: Text fields that create RGB colors

The following screenshot shows the display of the app after the color has been entered and is shown within the app:

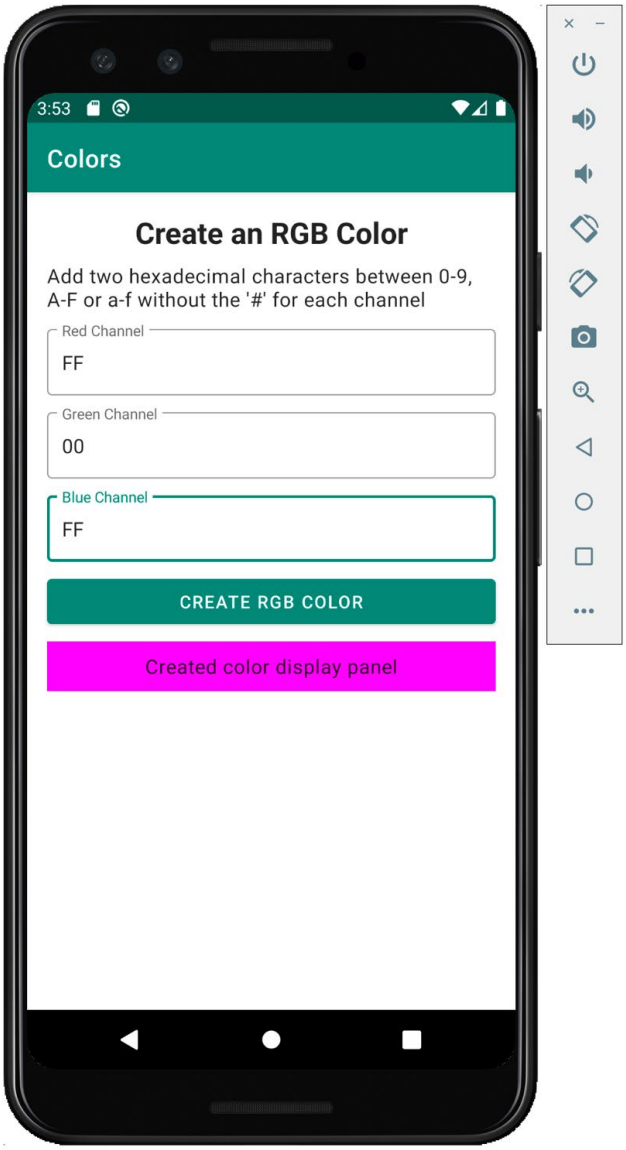

Figure 1.29: Output when the color is displayed

With this activity, we have produced an app to create RGB colors. Depending on the values that you enter in the text fields shown in *Figure 1.30*, the output will vary.

The source code for this activity is here: http://packt.live/3o2Ji1W

CHAPTER 2: BUILDING USER SCREEN FLOWS

ACTIVITY 2.01: CREATING A LOGIN FORM

Solution

1. Create a new project called **`Login Activity`**. Start by creating a new empty Activity project (**`File`** | **`New`** | **`New Project`** | **`Empty Activity`**). Name your application **`Login Activity`** and leave everything else with their default values, and then click **`Finish`**.

2. You need to add all the resource values you need that are not added by default. First, let's make changes to the **`strings.xml`** file. This file is needed to display all the text displayed in the app:

```
<resources>
    <string name="app_name">Login Activity</string>
    <string name="header_text">Please enter your username
      and password below to login:</string>
    <string name="welcome_text">Hello %s you are now
      logged in, we hope you enjoy using the app!</string>
    <string name="login_form_entry_error">Please fill in
      both username and password fields!</string>
    <string name="login_error">Either your user name or
      password is not recognised! Please try again.</string>
    <string name="username_label">Enter your username:
      </string>
    <string name="password_label">Enter your password:
      </string>
    <string name="submit_button_text">LOGIN</string>
</resources>
```

3. Now, open the **`themes.xml`** file and add the following styles:

```
    <style name="header"
      parent="TextAppearance.AppCompat.Title">
        <item name="android:gravity">center</item>
        <item name="android:layout_marginStart">24dp</item>
        <item name="android:layout_marginLeft">24dp</item>
        <item name="android:layout_marginEnd">24dp</item>
        <item name="android:layout_marginRight">24dp</item>
        <item name="android:textSize">20sp</item>
    </style>

    <style name="edit_text_login"
      parent="TextAppearance.AppCompat.Body1">
        <item name="android:layout_marginTop">16dp</item>
```

```
        <item name="android:layout_gravity">center</item>
        <item name="android:textSize">20sp</item>
        <item name="android:inputType">text</item>
    </style>

    <style name="button"
      parent="TextAppearance.AppCompat.Button">
        <item name="android:layout_margin">16dp</item>
        <item name="android:gravity">center</item>
        <item name="android:textSize">20sp</item>
    </style>

    <style name="page">
        <item name="android:layout_margin">8dp</item>
        <item name="android:padding">8dp</item>
    </style>
```

4. This sets up the strings and styles that the app will use. Now, you need to create the layout that the app will use. Open up **`activity_main.xml`** and add the **`header`**, **`user_name`**, **`password`**, and **`submit_button`** Views. This follows the format you used previously with **`ConstraintLayout`** where you added a **`TextView`** field called **`header`**, which was constrained to the top of its parent using **`app:layout_constraintTop_toTopOf="parent"`**. The username is then constrained to the bottom of the header using **`app:layout_constraintTop_toBottomOf="@id/header"`**. The rest of the form with the password and submit button follow the same pattern, leaving you with the following layout:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    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"
    style="@style/page"
    tools:context=".MainActivity">

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

```
            android:layout_height="wrap_content"
            android:text="@string/header_text"
            app:layout_constraintTop_toTopOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"/>

        <EditText
            android:id="@+id/user_name"
            style="@style/edit_text_login"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:hint="@string/username_label"
            app:layout_constraintTop_toBottomOf="@id/header"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintEnd_toEndOf="parent"/>

        <EditText
            android:id="@+id/password"
            style="@style/edit_text_login"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:hint="@string/password_label"
            android:inputType="textPassword"
            app:layout_constraintTop_toBottomOf="@id/user_name"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintEnd_toEndOf="parent"/>

        <Button
            android:id="@+id/submit_button"
            style="@style/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/submit_button_text"
            app:layout_constraintTop_toBottomOf="@id/password"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"/>
    </androidx.constraintlayout.widget.ConstraintLayout>
```

> **NOTE**

> You can find the preceding layout in the GitHub repository at http://packt.live/3qxWL3s.

5. Now, the key part of this solution to the Activity is to use a **`singleTop`** Android Activity. Go to **`app`** | **`src`** | **`main`** | **`AndroidManifest.xml`** and add **`android:launchMode="singleTop"`** to **`MainActivity`**:

```
<activity android:name="com.example.loginactivity.MainActivity"
android:launchMode="singleTop">
```

6. Next, open up **`MainActivity`** and replace the code with the following:

```
package com.example.loginactivity

import android.content.Context
import android.content.Intent
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.Gravity
import android.view.inputmethod.InputMethodManager
import android.widget.Button
import android.widget.EditText
import android.widget.TextView
import android.widget.Toast
import androidx.core.view.isVisible

const val USER_NAME_KEY = "USER_NAME_KEY"
const val PASSWORD_KEY = "PASSWORD_KEY"

const val IS_LOGGED_IN = "IS_LOGGED_IN"
const val LOGGED_IN_USERNAME = "LOGGED_IN_USERNAME"

//This is done as an example for simplicity and user/pwd
  credentials should never be stored in an app
const val USER_NAME_CORRECT_VALUE = "someusername"
const val PASSWORD_CORRECT_VALUE = "somepassword"

class MainActivity : AppCompatActivity() {

    private var isLoggedIn = false
    private var loggedInUser = ""
```

```
private val submitButton: Button
    get() = findViewById(R.id.submit_button)

private val userName: EditText
    get() = findViewById(R.id.user_name)

private val password: EditText
    get() = findViewById(R.id.password)

private val header: TextView
    get() = findViewById(R.id.header)

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

    submitButton.setOnClickListener {

        val userNameForm = userName.text.toString().trim()
        val passwordForm = password.text.toString().trim()

        hideKeyboard()

        if (userNameForm.isNotEmpty() &&
          passwordForm.isNotEmpty()) {

            //Set the name of the activity to launch
            Intent(this, MainActivity::class.java).also {
              loginIntent ->
                //Add the data
                loginIntent.putExtra(USER_NAME_KEY,
                  userNameForm)
                loginIntent.putExtra(PASSWORD_KEY,
                  passwordForm)
                //Launch
                startActivity(loginIntent)
            }

        } else {
            val toast = Toast.makeText(this,
              getString(R.string.login_form_entry_error),
                Toast.LENGTH_LONG)
```

```
                toast.setGravity(Gravity.CENTER, 0, 0)
                toast.show()
            }

        }
    }

    private fun hideKeyboard() {
        if (currentFocus != null) {
            val imm = getSystemService
              (Context.INPUT_METHOD_SERVICE) as
                InputMethodManager

imm.hideSoftInputFromWindow(currentFocus?.windowToken, 0)
        }
    }
}
```

This should be familiar to you from previous exercises. You add interaction with the button by adding a **`ClickListener`** interface, checking that the username and password fields are not empty and then creating an intent. In this case, as this is a **`singleTop`** Activity, the intent takes the same name as the current **`MainActivity`** activity as you are going to launch it again; but on this occasion, it will be a different intent that has the username and password details, which need to be verified. If the user has not entered their username and password correctly, then pop up a toast message.

7. Add code to validate the username and password as follows:

```
private fun setLoggedIn(loggedInUserName: String) {
    loggedInUser = loggedInUserName
    val welcomeMessage = getString(R.string.welcome_text,
      loggedInUserName)
    userName.isVisible = false
    password.isVisible = false
    submitButton.isVisible = false
    header.text = welcomeMessage
}

private fun hasEnteredCorrectCredentials(
    userNameForm: String,
    passwordForm: String
): Boolean {
```

```
        return userNameForm.contentEquals(USER_NAME_CORRECT_VALUE)
          && passwordForm.contentEquals(
            PASSWORD_CORRECT_VALUE
        )
    }
```

The **`hasEnteredCorrectCredentials`** method checks that the username and password match the values stored as constants in the Activity:

```
const val USER_NAME_CORRECT_VALUE = "someusername"
const val PASSWORD_CORRECT_VALUE = "somepassword"
```

The **`setLoggedIn`** method displays a welcome message with the user's name in the header and sets all the other Views to **`gone`** so that they no longer appear in the layout.

8. Next, we use the **`onNewIntent`** function to process the intent that has just been sent:

```
override fun onNewIntent(newIntent: Intent?) {
    super.onNewIntent(newIntent)

    //Set the new Intent to the one to process
    intent = newIntent

    //Get the intent which started this activity
    intent?.let { loginIntent ->

        val userNameForm = loginIntent.getStringExtra
          (USER_NAME_KEY) ?: ""
        val passwordForm = loginIntent.getStringExtra
          (PASSWORD_KEY) ?: ""

        val loggedInCorrectly =hasEnteredCorrectCredentials
          (userNameForm.trim(), passwordForm.trim())

        if (loggedInCorrectly) {
            setLoggedIn(userNameForm)
            isLoggedIn = true
        } else {
            val toast = Toast.makeText(this,
              getString(R.string.login_error),
              Toast.LENGTH_LONG)
            toast.setGravity(Gravity.CENTER, 0, 0)
            toast.show()
```

```
            }
        }
    }
```

The callback for the **`singleTop`** mode Activities is **`override fun onNewIntent(intent: Intent?)`**, and it's here where you use **`setIntent(intent)`** to set the intent to process. The rest of the code retrieves the username and password passed in the intent's extras and then validates these values against the correct username and password, displaying a welcome message with the user's name if they are successful or popping up an error message if the values don't match,

9. Then, finally, you store the username and password and the logged-in status so that if the Activity is recreated, the user will still be logged in and the welcome message with their name will still be displayed:

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)

    outState.putBoolean(IS_LOGGED_IN, isLoggedIn)
    outState.putString(LOGGED_IN_USERNAME, loggedInUser)
}

override fun onRestoreInstanceState(savedInstanceState:
  Bundle) {
    super.onRestoreInstanceState(savedInstanceState)

    isLoggedIn = savedInstanceState.getBoolean(IS_LOGGED_IN,
      false)
    loggedInUser = savedInstanceState.getString
      (LOGGED_IN_USERNAME, "")

    if (isLoggedIn && loggedInUser.isNotEmpty()) {
        setLoggedIn(loggedInUser)
    }
}
```

10. When the details have been entered and we load the screen for the first time, it will look as in *Figure 2.24*:

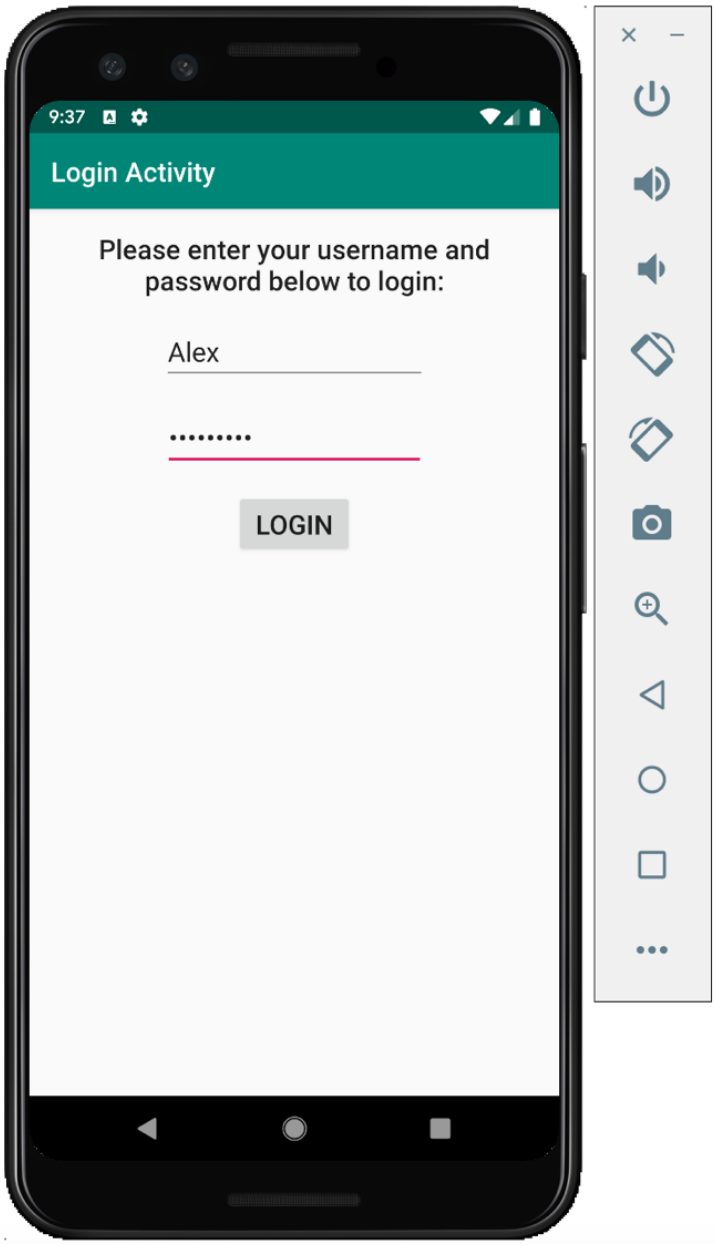

Figure 2.24: The display after the user enters their details

11. On entering an incorrect username or password, a toast error message will be displayed, as shown in *Figure 2.25*

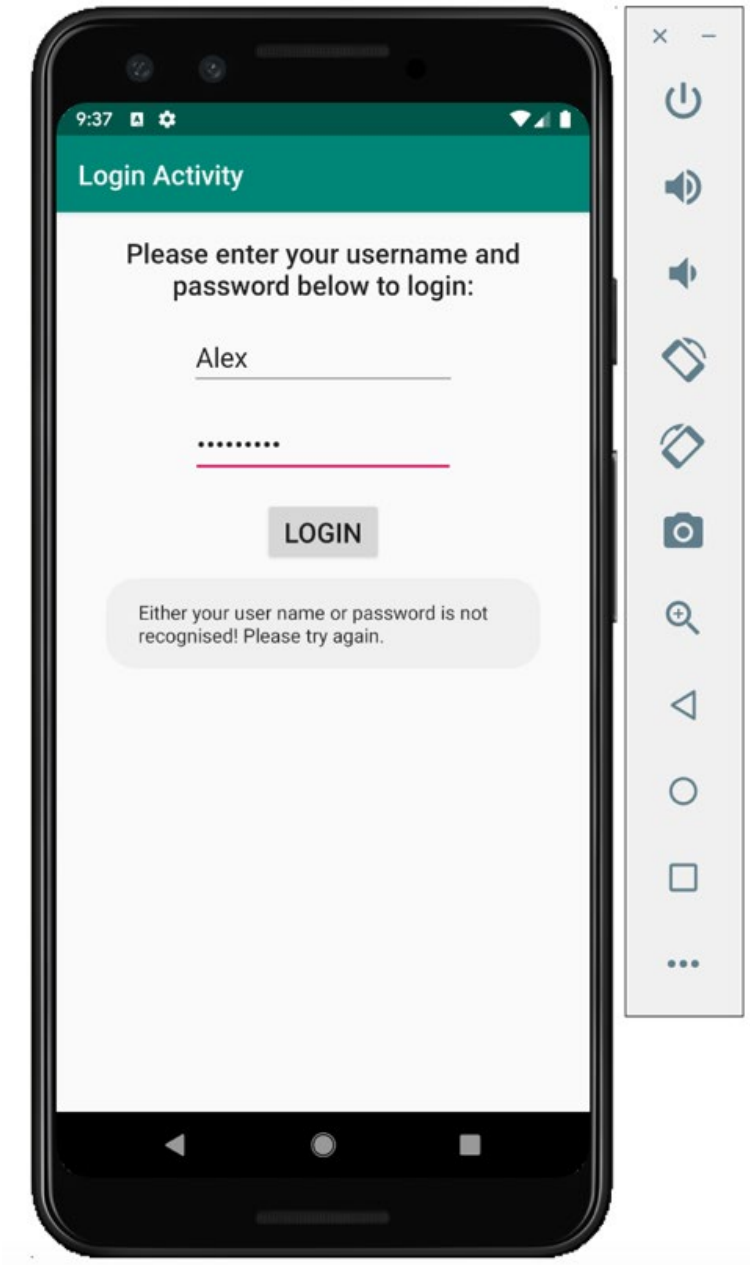

Figure 2.25: The display on entering an incorrect username or password

12. On a successful login, here's the display, shown in *Figure 2.26*:

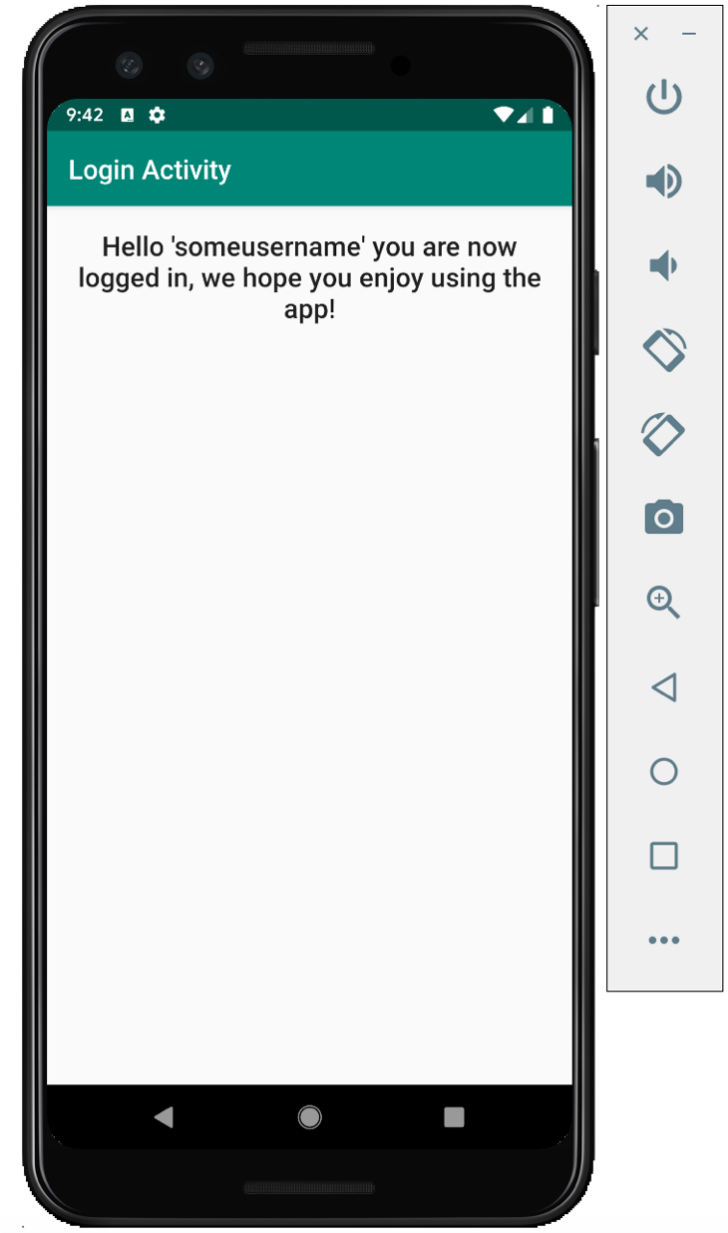

Figure 2.26: The display after the user successfully logs in

This activity has demonstrated how to create a login form, add interaction with the user, and validate the entered data to either log in successfully or handle errors and give feedback to the user.

The solution code for all of the approaches to solve this activity, including this one, can be found on GitHub at the following locations:

- **Using a singleTop Activity**: http://packt.live/3iw849D
- **Using a standard Activity**: http://packt.live/39S7HlW
- **Using startActivityForResult**: http://packt.live/3qEeTIZ

CHAPTER 3: DEVELOPING THE UI WITH FRAGMENTS

ACTIVITY 3.01: CREATING A QUIZ ON THE PLANETS

Solution:

1. First of all, create a new project with an empty activity and call it **`Planet Quiz`**.
2. Once you have done that, add the following dependency you need to manage adding/replacing fragments, **`FragmentContainerView`** into app/build.gradle within the dependences{ } block:

```
    implementation 'androidx.fragment:fragment-ktx:1.2.5'
```

3. Add the following strings needed in the project:

`strings.xml`

```
    <string name="app_name">Planet Quiz</string>
    <string name="largest_planet">What is the largest planet?</string>
    <string name="most_moons">Which planet has the most moons?</string>
    <string name="side_spinning">Which planet spins on its side?</string>

    <string name="mercury">MERCURY</string>
    <string name="venus">VENUS</string>
    <string name="earth">EARTH</string>
    <string name="mars">MARS</string>
    <string name="jupiter">JUPITER</string>
    <string name="saturn">SATURN</string>
    <string name="uranus">URANUS</string>
    <string name="neptune">NEPTUNE</string>

    <string name="jupiter_answer">%s Jupiter is the largest planet
      and is 2.5 times the mass of all the other planets put together.
    </string>
    <string name="saturn_answer">%s Saturn has the most moons and has 82
      moons.</string>
    <string name="uranus_answer">%s Uranus spins on its side
        with its axis at nearly a right angle to the sun.</string>
    <string name="correct">CORRECT!</string>
    <string name="wrong">WRONG!</string>
```

You will notice that the answer strings have **%s** in them. This is so the answer strings can be formatted with a string format argument to display whether the answer is **CORRECT!** or **WRONG!**.

4. Then update **themes.xml**:

themes.xml

```
<style name="HeaderText"
  parent="Base.TextAppearance.AppCompat.Large">
    <item name="android:padding">18dp</item>
    <item name="android:textSize">24sp</item>
    <item name="android:textStyle">bold</item>
    <item name="android:gravity">center</item>
</style>

    <style name="ButtonText">
        <item name="android:padding">14dp</item>
        <item name="android:textAllCaps">false</item>
        <item name="android:textSize">18sp</item>
    </style>

    <style name="AnswerText">
        <item name="android:padding">14dp</item>
        <item name="android:textSize">18sp</item>
        <item name="android:textStyle">bold</item>
    </style>
```

These are basic styles so feel free to change them. Buttons, by default, display in uppercase text so the **android:textAllCaps** item set to **false** allows us to not display the buttons in uppercase.

The approach that is used for this activity is to use dynamic fragments and a listener in a **QuestionsFragment** class to pass data to an **AnswersFragment** with the fragments being added to a container **ViewGroup** in the activity layout file.

5. The first stage of creating this is to create a new blank fragment with the toolbar **File** | **New** | **Fragment** | **Fragment (Blank)** option and call it **QuestionsFragment**.

6. Once the fragment has been created, open the **`fragment_questions.xml`** layout file and add this code:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
  android"

    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".QuestionsFragment">

    <TextView
        style="@style/HeaderText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:text="@string/app_name"/>

    <Button
        android:id="@+id/largest_planet"
        style="@style/ButtonText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:text="@string/largest_planet"/>

    <Button
        android:id="@+id/most_moons"
        style="@style/ButtonText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:text="@string/most_moons"/>

    <Button
        android:id="@+id/side_spinning"
        style="@style/ButtonText"
```

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:text="@string/side_spinning"/>
</LinearLayout>
```

You are adding a simple **`LinearLayout`** to display the questions one on top of the other. You assign IDs to the buttons so they can be retrieved in the fragment.

7. Next, go into the **`QuestionsFragment`** and update the code to the following:

```
package com.example.planetquiz

import android.content.Context
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

interface AnswersListener {
    fun onSelected(questionId: Int)
}

class QuestionsFragment : Fragment(), View.OnClickListener {

    private lateinit var answersListener: AnswersListener

    override fun onAttach(context: Context) {
        super.onAttach(context)
        if (context is AnswersListener) {
            answersListener = context
        } else {
            throw RuntimeException("Must implement AnswersListener")
        }
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
          savedInstanceState: Bundle?): View? {

        return inflater.inflate(R.layout.fragment_questions, container,
          false)
```

```
        }

        override fun onViewCreated(view: View,
          savedInstanceState: Bundle?) {
            super.onViewCreated(view, savedInstanceState)

            val planets = listOf<View>(
                view.findViewById(R.id.most_moons),
                view.findViewById(R.id.largest_planet),
                view.findViewById(R.id.side_spinning)
            )

            planets.forEach {
                it.setOnClickListener(this)
            }
        }

        override fun onClick(v: View?) {

            v?.let { question ->
                answersListener.onSelected(question.id)
            }
        }
    }
```

As you saw in *Exercise 3.04*, *Adding Fragments Dynamically to an Activity*, you define a listener (**`AnswersListener`**) that will be associated with the Android activity, and the fragment is included with the **`onAttach`** method. This is the way you will communicate back to the activity which question the user has clicked on. The rest of the class should be quite familiar. You set the layout and then retrieve the view by IDs before setting a **`ClickListener`** to pass the selected question back into the Android activity.

8. Next, create another blank fragment called **`AnswersFragment`**. The first thing to do is update the **`fragment_answers.xml`** layout file to include a view for the question header text, all of the planet buttons, and a view to show the answer. Update the file to the following:

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

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        tools:context=".AnswersFragment">

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

            <TextView
                android:id="@+id/header_text"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:gravity="center"
                style="@style/HeaderText"
                tools:text="Question Text" />

            <Button
                android:id="@+id/mercury"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:gravity="center"
                style="@style/ButtonText"
                android:text="@string/mercury" />

            <!-- Add the 7 other planets below this line here -->

            <TextView
                android:id="@+id/answer"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                style="@style/AnswerText"
                tools:text="Planet"/>

        </LinearLayout>
    </ScrollView>
```

What is shown here is only a snippet of the full file.

9. Add the other seven planets after **`mercury`**, making sure the IDs are all lowercase as they are referenced as lower case in the following code examples.

10. Next, open the **`AnswersFragment`** that has been created. You should then see the following initial fragment contents from the class header to the end of the file:

```
class AnswersFragment : Fragment() {
    // TODO: Rename and change types of parameters
    private var param1: String? = null
    private var param2: String? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        arguments?.let {
            param1 = it.getString(ARG_PARAM1)
            param2 = it.getString(ARG_PARAM2)
        }
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_answers,
          container, false)
    }

    companion object {
        /**
        * Use this factory method to create a new instance of
         * this fragment using the provided parameters.
         *
         * @param param1 Parameter 1.
         * @param param2 Parameter 2.
         * @return A new instance of fragment BlankFragment.
         */
        // TODO: Rename and change types and number of parameters
        @JvmStatic
        fun newInstance(param1: String, param2: String) =
            AnswersFragment().apply {
                arguments = Bundle().apply {
                    putString(ARG_PARAM1, param1)
                    putString(ARG_PARAM2, param2)
            }
```

```
            }
        }
    }
```

11. Add the following constants to the companion object :

```
    private const val QUESTION_ID = "QUESTION_ID"
    private const val NO_QUESTION_SET = 0
```

    These will be used to pass in the ID of the question from the **`MainActivity`** by setting the fragment arguments with the **`QUESTION_ID`** key and also to add a default value with **`NO_QUESTION_SET`**.

12. In the class header, add a view **`click listener`** so it appears like this:

```
    class AnswersFragment : Fragment(), View.OnClickListener
```

    It will be highlighted in red, and if you hover over the error, it will prompt you to implement the **`click listener`** function, **`onClick`**, which you should do.

13. For now, it will just display as follows:

```
    override fun onClick(v: View?) {
        TODO("not implemented yet")
    }
```

14. Add the view widget import to the imports list:

```
    import android.widget.TextView
```

15. Add the **`questionId`** property and view references below the class header, so it appears like this:

```
    class AnswersFragment : Fragment(), View.OnClickListener {

        var questionId: Int = NO_QUESTION_SET

        private val headerText: TextView?
            get() = view?.findViewById(R.id.header_text)

        private val answer: TextView?
            get() = view?.findViewById(R.id.answer)
```

    This will be the property used to both set the question header with what text to display on this screen as well as to evaluate which question is being answered when the user clicks an answer to a question to set the result.

16. Next, replace the **`newInstance`** function in the companion object with the following:

```
@JvmStatic
fun newInstance(questionId: Int) =
    AnswersFragment().apply {
        arguments = Bundle().apply {
            putInt(QUESTION_ID, questionId)
        }
    }
```

We can put java's static methods inside Kotlin **Companion objects**. Here, you are creating a factory method, **`newInstance`**, which the **`MainActivity`** will use to create the **`AnswersFragment`** and pass in the **`questionId`** with the **`QUESTION_ID`** key. This can then be retrieved in the **`AnswersFragment`**.

17. Next, override the **`onViewCreated`** function:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    val planets = listOf<View>(
        view.findViewById(R.id.mercury),
        view.findViewById(R.id.venus),
        view.findViewById(R.id.earth),
        view.findViewById(R.id.mars),
        view.findViewById(R.id.jupiter),
        view.findViewById(R.id.saturn),
        view.findViewById(R.id.uranus),
        view.findViewById(R.id.neptune)
    )

    planets.forEach {
        it.setOnClickListener(this)
    }

    questionId = arguments?.getInt(QUESTION_ID) ?: NO_QUESTION_SET

    //Set Header Text
    when (questionId) {
        R.id.largest_planet -> {
            headerText?.text = getString(R.string.largest_planet)
```

```
            }
            R.id.most_moons -> {
                headerText?.text = getString(R.string.most_moons)
            }
            R.id.side_spinning -> {
                headerText?.text = getString(R.string.side_spinning)
            }
        }
    }
```

18. Here, you are retrieving the IDs of the buttons representing planets in the layout with the **`val planets = listOf<View>(view.findViewById(R.id.mercury), view.findViewById(R.id.venus), ...)`** line. Most importantly, it's here where you set the **`questionId`**:

```
questionId = arguments?.getInt(QUESTION_ID) ?: NO_QUESTION_SET
```

    This is then used to show the question header text to display in the layout, but also will be used in the **`onClick`** method to show the answer the user has clicked based on the question.

19. Now update the **`onClick`** function with the following:

```
when (questionId) {
    R.id.largest_planet -> {
        if (v?.id == R.id.jupiter) {
            answer?.text = getString(R.string.jupiter_answer,
              getString(R.string.correct))
        } else {
            answer?.text = getString(R.string.jupiter_answer,
              getString(R.string.wrong))
        }
    }
    R.id.most_moons -> {
        if (v?.id == R.id.saturn) {
            answer?.text = getString(R.string.saturn_answer,
              getString(R.string.correct))
        } else {
            answer?.text = getString(R.string.saturn_answer,
              getString(R.string.wrong))
```

```
            }
        }
        R.id.side_spinning -> {
            if (v?.id == R.id.uranus) {
                answer?.text = getString(R.string.uranus_answer,
                  getString(R.string.correct))
            } else {
                answer?.text = getString(R.string.uranus_answer,
                  getString(R.string.wrong))
            }
        }
    }
```

You have already set the **questionId** by parsing the argument, which was set when the **AnswersFragment** was created in **onViewCreated**. Here, you evaluate the **questionId** in the **when** expression, which has already been set, and then you know which branch of the **when** expression to execute.

R.id.most_moons ->, for example, when clicked, then uses a further **if** condition to check whether the button the user clicked is the correct one by evaluating the ID of the button clicked, **if (v?.id == R.id.saturn)**, then display to the user that their answer is correct passing in the message text to display, formatting it with the **CORRECT!** string argument. Otherwise in the **else** statement format the message text to display with the **WRONG!** string argument.

20. Currently, the **QuestionsFragment** and **AnswersFragment** are not connected. To do this firstly, you need to open the **activity_main.xml** file and replace the **ConstraintLayout** with a **FragmentContainerView**:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
  xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

21. This will be used as the container for the fragments. Next, go into **`MainActivity`** and update it with the following:

```
import androidx.fragment.app.FragmentContainerView
class MainActivity : AppCompatActivity(), AnswersListener {

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

    if (savedInstanceState == null) {

        findViewById<FragmentContainerView>
          (R.id.fragment_container)?.let { frameLayout ->

            val questionsFragment = QuestionsFragment()

            supportFragmentManager.beginTransaction()
                .add(frameLayout.id, questionsFragment).commit()
        }
    }
}

    override fun onSelected(questionId: Int) {
        findViewById<FragmentContainerView>
          (R.id.fragment_container)?.let {frameLayout ->

            val answersFragment =
              AnswersFragment.newInstance(questionId)

            supportFragmentManager.beginTransaction()
                .replace(frameLayout.id, answersFragment)
                .addToBackStack(null)
                .commit()
        }
    }
}
```

An example of a correct answer to one of the planet questions is as follows:

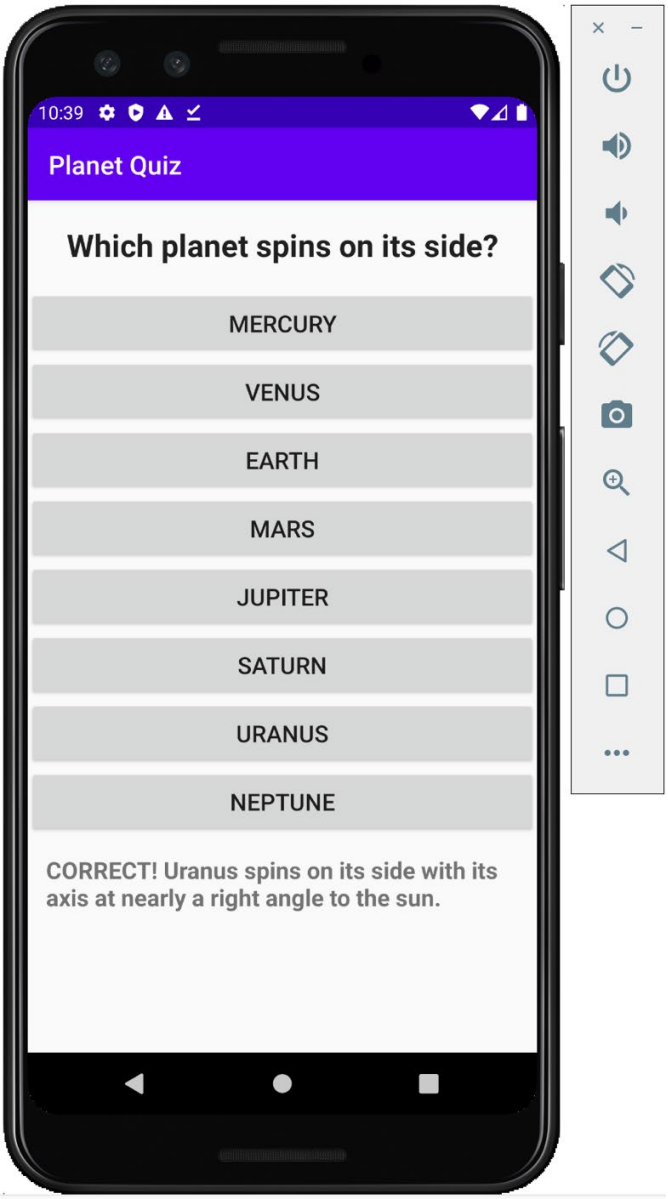

Figure 3.23: Planet Quiz answer screen with the correct answer

The **`MainActivity`** begins by retrieving the **`FrameLayout`** by its ID and then if the **`fragment_container`** is not null, which is checked by the **`?.let`** scope function, add the **`QuestionsFragment`** to the **`FrameLayout`** and *commit* the transaction.

The **`MainActivity`** implements the **`AnswersListener`** and in the **`onSelected`** function, retrieves the result of the user clicking on one of the question buttons by creating the **`AnswersFragment`** and passing in the **`questionId`** of the question button to the **`newInstance`** static method, which sets the **`questionId`** argument in the **`AnswersFragment`**. Then, the **`AnswersFragment`** replaces the **`QuestionsFragment`** with **`.replace(frameLayout.id, answersFragment)`**, the back stack is created with **`.addToBackStack(null)`**, and finally, the transaction is committed (**`.commit()`**), which executes the transaction that began with **`supportFragmentManager.beginTransaction()`**.

The source for this activity is at http://packt.live/35WMXrZ

CHAPTER 4: BUILDING APP NAVIGATION

ACTIVITY 4.01: BUILDING PRIMARY AND SECONDARY APP NAVIGATION

Solution:

1. Create a new app in Android Studio with an empty activity called **`Navigation Activity.`**
2. Add the following dependencies to **`app/build.gradle`**:

```
implementation 'androidx.navigation:navigation-fragment-
  ktx:2.3.2'
implementation 'androidx.navigation:navigation-ui-ktx:2.3.2'
```

3. Replace **`colors.xml`** with the following:

`colors.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#6200EE</color>
    <color name="colorPrimaryDark">#3700B3</color>
    <color name="colorAccent">#03DAC5</color>
</resources>
```

4. Append **`strings.xml`** in the **`res/values`** folder with the following values:

```
    <!-- Bottom Navigation -->
    <string name="home">Home</string>
    <string name="account">Account</string>
    <string name="mysports">My Sports</string>
    <string name="profile">Profile</string>


    <string name="home_fragment">Home Fragment</string>
    <string name="account_fragment">Account Fragment</string>
    <string name="mysports_fragment">
      My Sports Fragment</string>
    <string name="profile_fragment">Profile Fragment</string>
    <string name="football">Football</string>
    <string name="basketball">Basketball</string>
    <string name="hockey">Hockey</string>
    <string name="football_fragment">
      Football Fragment</string>
```

```
    <string name="basketball_fragment">
      Basketball Fragment</string>
    <string name="hockey_fragment">Hockey Fragment</string>
```

5. Replace **`themes.xml`** with the following:

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.NavigationActivity"
      parent="Theme.MaterialComponents.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">
          @color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>

    <style name="button_card" parent
      ="Widget.MaterialComponents.Button.OutlinedButton">
        <item name="strokeColor">@color/colorPrimary</item>
        <item name="strokeWidth">2dp</item>
    </style>

</resources>
```

6. Create the following blank fragments:

- **`HomeFragment`**
- **`MySportsFragment`**
- **`ProfileFragment`**
- **`AccountFragment`**
- **`HockeyFragment`**
- **`FootballFragment`**
- **`BasketballFragment`**

7. Add the following content for all the layout files except **`fragment_mysports.xml`**, changing only the **`TextView`** string corresponding to the layout name. There is an example of one of these fragment layout files below:

`fragment_profile.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
  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">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginBottom="8dp"
        android:text="@string/profile_fragment"
        android:textAlignment="center"
        android:layout_gravity="center_horizontal"
        android:textSize="20sp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

8. Update the **`fragment_mysports.xml`** layout file with the three material style buttons you used in the other exercises in this chapter to link to the three sports secondary destinations; that is, basketball, football, and hockey:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
  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">

    <com.google.android.material.button.MaterialButton
```

```
            android:id="@+id/football"
            style="@style/button_card"
            android:layout_width="140dp"
            android:layout_height="140dp"
            android:layout_marginTop="16dp"
            android:layout_marginStart="16dp"
            android:text="@string/football"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

        <com.google.android.material.button.MaterialButton
            android:id="@+id/basketball"
            style="@style/button_card"
            android:layout_width="140dp"
            android:layout_height="140dp"
            android:layout_marginTop="16dp"
            android:layout_marginStart="16dp"
            android:text="@string/basketball"
            app:layout_constraintStart_toEndOf="@id/football"
            app:layout_constraintTop_toTopOf="parent" />

        <com.google.android.material.button.MaterialButton
            android:id="@+id/hockey"
            style="@style/button_card"
            android:layout_width="140dp"
            android:layout_height="140dp"
            android:layout_marginTop="4dp"
            android:layout_marginStart="16dp"
            android:text="@string/hockey"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@id/football" />

    </androidx.constraintlayout.widget.ConstraintLayout>
```

9. Create a navigation folder and then a navigation graph file named **mobile_navigation** with the following content:

```
    <?xml version="1.0" encoding="utf-8"?>
    <navigation
      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:id="@+id/mobile_navigation"
        app:startDestination="@+id/nav_home">

        <fragment
            android:id="@+id/nav_home"
            android:name="com.example.navigationactivity
              .HomeFragment"
            android:label="@string/home"
            tools:layout="@layout/fragment_home"/>

        <fragment
            android:id="@+id/nav_account"
            android:name="com.example.navigationactivity
              .AccountFragment"
            android:label="@string/account"
            tools:layout="@layout/fragment_account" />

        <fragment
            android:id="@+id/nav_profile"
            android:name="com.example.navigationactivity
              .ProfileFragment"
            android:label="@string/profile"
            tools:layout="@layout/fragment_profile" />

        <fragment
            android:id="@+id/nav_football"
            android:name="com.example.navigationactivity
              .FootballFragment"
            android:label="@string/football"
            tools:layout="@layout/fragment_football" />

        <fragment
            android:id="@+id/nav_basketball"
            android:name="com.example.navigationactivity
              .BasketballFragment"
            android:label="@string/basketball"
            tools:layout="@layout/fragment_basketball" />

        <fragment
            android:id="@+id/nav_hockey"
            android:name="com.example.navigationactivity
              .HockeyFragment"
            android:label="@string/hockey"
```

```
            tools:layout="@layout/fragment_hockey" />

        <fragment
            android:id="@+id/nav_mysports"
            android:name="com.example.navigationactivity
              .MySportsFragment"
            android:label="@string/mysports"
            tools:layout="@layout/fragment_mysports" />

    </navigation>
```

10. Now that you have added all seven fragments that will be used in the app, create the three actions that will be used to navigate from the **`My Sports`** primary destination to the secondary destinations. These should be added to the **`nav_mysports`** fragment:

```
    <fragment
        android:id="@+id/nav_mysports"
        android:name="com.example.navigationactivity
          .MySportsFragment"
        android:label="@string/mysports"
        tools:layout="@layout/fragment_mysports" >

        <action
            android:id="@+id/nav_mysports_to_football"
            app:destination="@id/nav_football"
            app:popUpTo="@id/nav_mysports" />

        <action
            android:id="@+id/nav_mysports_to_basketball"
            app:destination="@id/nav_basketball"
            app:popUpTo="@id/nav_mysports" />

        <action
            android:id="@+id/nav_mysports_to_hockey"
            app:destination="@id/nav_hockey"
            app:popUpTo="@id/nav_mysports" />

    </fragment>
```

11. Go back into **`MySportsFragment`** and replace it with the following to set up the **`Navigation`** click listeners to these secondary destinations:

```
package com.example.navigationactivity

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Button
import androidx.fragment.app.Fragment
import androidx.navigation.Navigation

class MySportsFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {

        val view = inflater.inflate
          (R.layout.fragment_mysports, container, false)

        view.findViewById<Button>(R.id.basketball)
          ?.setOnClickListener(
            Navigation.createNavigateOnClickListener
              (R.id.nav_mysports_to_basketball, null)
        )

        view.findViewById<Button>(R.id.football)
          ?.setOnClickListener(
            Navigation.createNavigateOnClickListener
              (R.id.nav_mysports_to_football, null)
        )

        view.findViewById<Button>(R.id.hockey)
          ?.setOnClickListener(
```

```
                Navigation.createNavigateOnClickListener
                  (R.id.nav_mysports_to_hockey, null)
            )

            return view
        }
    }
```

12. Add the navigation icons you want to use by creating these icons using the Vector Clip Art available within Android Studio. Go to **`File`** | **`New`** | **`Vector Asset`**, select **`Clip Art`** and browse to the icons you wish to use in the **`Home`**, **`Account`**, **`Profile`** and **`My Sport`** sections. Remember to have the **`res`** folder selected before you use the **`File Toolbar`** option so that the option to create a vector asset appears. Alternatively use the icons available in the completed exercise here: http://packt.live/2NpO4Kr

13. Add the menu and icons you want to populate the bottom navigation with. The icons you have just created will be stored in the **`res/drawable`** folder. Do this by creating a menu called **`bottom_nav_menu`** and adding the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android
  ="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/nav_home"
        android:title="@string/home"
        android:icon="@drawable/home"/>

    <item
        android:id="@+id/nav_account"
        android:title="@string/account"
        android:icon="@drawable/account"/>

    <item
        android:id="@+id/nav_profile"
        android:title="@string/profile"
        android:icon="@drawable/profile"/>

    <item
        android:id="@+id/nav_mysports"
        android:title="@string/mysports"
```

```
        android:icon="@drawable/mysports"/>

</menu>
```

Now, it's time to connect everything together.

14. Update **`activity_main.xml`** with **`BottomNavigationView`** and **`NavHostFragment`**:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <com.google.android.material.bottomnavigation
      .BottomNavigationView
        android:id="@+id/nav_view"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="0dp"
        android:layout_marginEnd="0dp"
        android:background="?android:attr/windowBackground"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:menu="@menu/bottom_nav_menu"
        app:labelVisibilityMode="labeled"/>

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintBottom_toTopOf="@id/nav_view"
        android:name=
          "androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:defaultNavHost="true"
```

```
        app:navGraph="@navigation/mobile_navigation" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

15. Then go back into **`MainActivity`** and update it with the syntax from *Exercise 4.02*: *Adding Bottom Navigation to Your App*:

```
package com.example.navigationactivity

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.navigation.findNavController
import androidx.navigation.fragment.NavHostFragment
import androidx.navigation.ui.AppBarConfiguration
import androidx.navigation.ui.navigateUp
import androidx.navigation.ui.setupActionBarWithNavController
import androidx.navigation.ui.setupWithNavController
import com.google.android.material.bottomnavigation
  .BottomNavigationView

class MainActivity : AppCompatActivity() {

    private lateinit var appBarConfiguration:
      AppBarConfiguration

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

        val navHostFragment = supportFragmentManager
          .findFragmentById(R.id.nav_host_fragment) as
            NavHostFragment
        val navController = navHostFragment.navController

        appBarConfiguration = AppBarConfiguration(setOf(
            R.id.nav_home, R.id.nav_account, R.id.nav_profile,
              R.id.nav_mysports))
        setupActionBarWithNavController(navController,
          appBarConfiguration)
        findViewById<BottomNavigationView>(R.id.nav_view)
          ?.setupWithNavController(navController)
    }

    override fun onSupportNavigateUp(): Boolean {
```

```
            val navController
              = findNavController(R.id.nav_host_fragment)
            return navController.navigateUp(appBarConfiguration)
              || super.onSupportNavigateUp()
        }
    }
```

16. Now, run the app and navigate to the **`My Sports`** section:

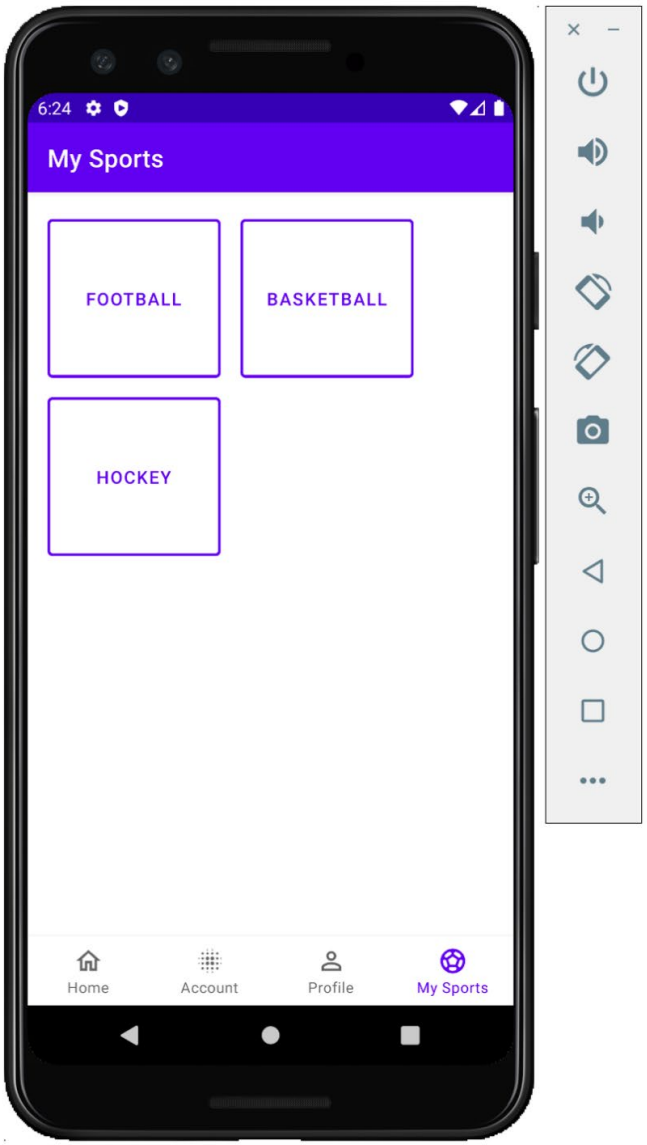

Figure 4.18: Bottom navigation with the My Sports menu displayed

17. Now, select one of the sports within this screen to use the action within the navigation graph to go to that sport's page:

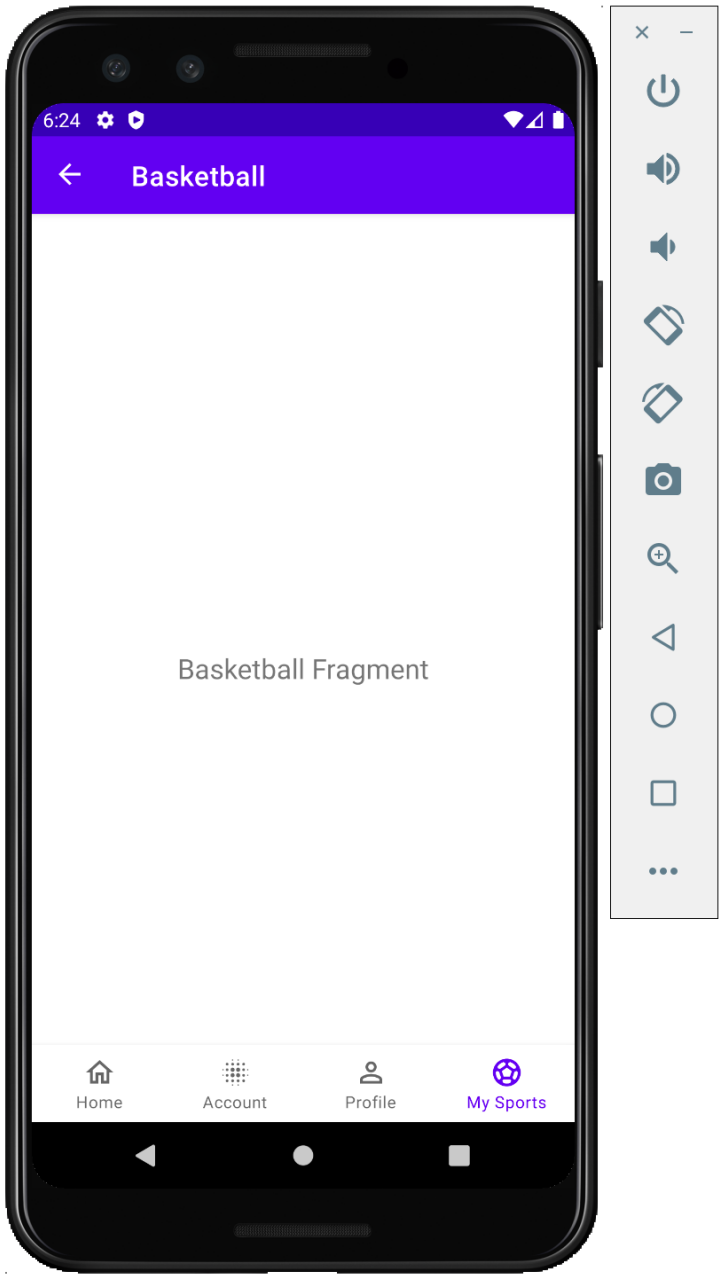

Figure 4.19: Bottom navigation showing a secondary sports destination

As you can see, the **`My Sports`** item is still selected in the primary bottom navigation, which lets the user know which section they are in. The content of the sport is displayed in the body of the page, the title of the destination is set, and the back navigation is handled by signaling to the user, with the arrow on the toolbar, that they can click it to go back to the primary destination.

This solution fulfills all the original criteria for this activity and has been developed using the power of Jetpack navigation and the view widgets available to us, such as **`BottomNavigationView`** and **`NavHostFragment`**. They do the heavy lifting while the navigation graph and bottom menu add the required configuration to bring your app together.

CHAPTER 5: ESSENTIAL LIBRARIES: RETROFIT, MOSHI, AND GLIDE

ACTIVITY 5.01: DISPLAYING THE CURRENT WEATHER

Solution:

Perform the following steps to complete the activity.

1. Create a new empty activity app.
2. Add internet permissions to the app's **`AndroidManifest.xml`** file to allow us to make API and image requests:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.newyorkweather">

    <uses-permission android:name="android.permission.INTERNET" />

    <application ...>
        ...
    </application>

</manifest>
```

3. Add Retrofit, the Moshi converter, and Glide support to the app's **`build.gradle`** file:

```
Dependencies {
    ...
    implementation 'com.squareup.retrofit2:retrofit:
      (latest version here)'
    implementation 'com.squareup.retrofit2:converter-moshi:2.9.0'
    implementation 'com.github.bumptech.glide:glide:4.11.0'
    ...
}
```

   You can use later versions, if available.

4. Add title, status, and description **`TextView`** instances to your main layout. Also, add an **`ImageView`** for the weather icon. Your layout should look somewhat like this:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
  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:padding="16dp">

        <TextView
            android:id="@+id/main_title"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:textSize="18sp"
            app:layout_constraintEnd_toStartOf="@+id/main_weather_icon"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            tools:text="New York" />

        <TextView
            android:id="@+id/main_status"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            app:layout_constraintEnd_toStartOf="@+id/main_weather_icon"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/main_title"
            tools:text="Light Rain" />

        <TextView
            android:id="@+id/main_description"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            app:layout_constraintEnd_toStartOf="@+id/main_weather_icon"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/main_status"
            tools:text="Humidity: 69%\nPredictability: 75%" />

        <ImageView
            android:id="@+id/main_weather_icon"
            android:layout_width="48dp"
            android:layout_height="48dp"
            app:layout_constraintEnd_toEndOf="parent"
```

```
            app:layout_constraintTop_toTopOf="parent"
            tools:background="@color/colorAccent" />

    </androidx.constraintlayout.widget.ConstraintLayout>
```

The output is as follows:

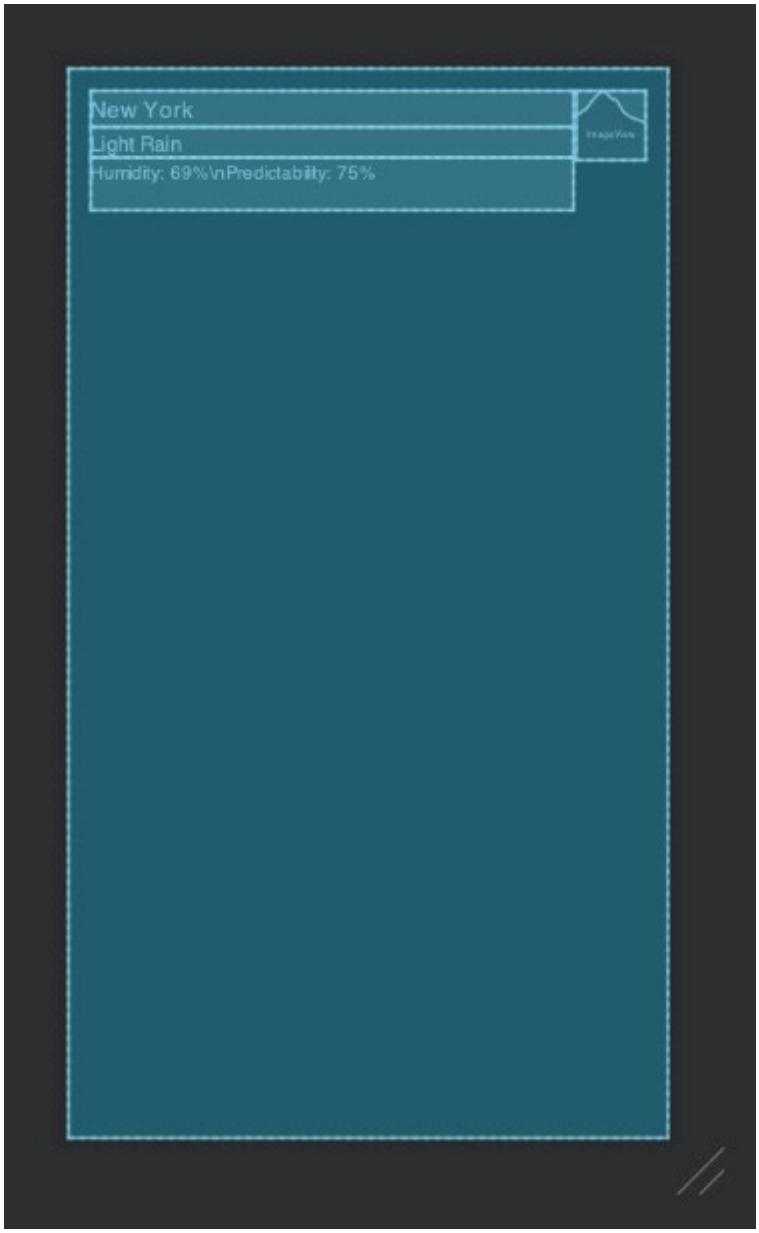

Figure 5.8: Main layout blueprint preview

5. Add models for the **`OpenWeatherMap`** weather API response. Given that we only have to define fields for data that is relevant to us, your models could look like this:

```
package com.example.newyorkweather.model

data class OpenWeatherMapResponseData(
    @field:Json(name = "name")
    val locationName: String,
    val weather: List<OpenWeatherMapWeatherData>
)

data class OpenWeatherMapWeatherData(
    @field:Json(name = "main")
    val status: String,
    val description: String,
    val icon: String
)
```

   Remember that you can use the **`@field:Json`** annotation to map API names to names that are meaningful to your app. In this example, **`name`** is mapped to **`locationName`** and **`main`** is mapped to **`status`**.

6. Add a service for the **`OpenWeatherMap`** weather API endpoint at **`https://api.openweathermap.org/data/2.5/weather`**, taking into account the location (**`q`**) and token (**`appid`**) query parameters, like so:

```
package com.example.newyorkweather.api

import retrofit2.Call
import retrofit2.http.GET
import retrofit2.http.Query

interface OpenWeatherMapService {
    @GET("weather")
    fun getWeather(
        @Query("q") location: String,
        @Query("appid") token: String
    ) : Call<OpenWeatherMapResponseData>
}
```

7. Create a Retrofit instance with a Moshi converter and the **`OpenWeatherMap`** service:

```
class MainActivity : AppCompatActivity() {
    private val retrofit by lazy {
        Retrofit.Builder()
            .baseUrl("https://api.openweathermap.org/data/2.5/")
            .addConverterFactory(MoshiConverterFactory.create())
            .build()
    }
    private val weatherApiService by lazy {
        retrofit.create(OpenWeatherMapService::class.java)
    }

    ...
}
```

8. Make a call to the API service in the **`onResume`** function of **`MainActivity`** with **`appid`** set to your token and the city set to **`New York`**:

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

    weatherApiService
        .getWeather("New York", "[YOUR TOKEN]")
        .enqueue(object : Callback<OpenWeatherMapResponseData> {
            override fun onFailure(call:
              Call<OpenWeatherMapResponseData>, t: Throwable) {
            }

            override fun onResponse(
                call: Call<OpenWeatherMapResponseData>,
                response: Response<OpenWeatherMapResponseData>
            ) {
            }
        })
}
```

While we hardcode the app token in this example, remember to never include your private tokens and other private information in code.

9. Handle the happy path: process the response, get the first result, and construct a weather URL for it based on the API response:

```
package com.example.newyorkweather
...
class MainActivity : AppCompatActivity() {
    private val titleView: TextView
        by lazy { findViewById(R.id.main_title) }
    private val statusView: TextView
        by lazy { findViewById(R.id.main_status) }
    private val descriptionView: TextView
        by lazy { findViewById(R.id.main_description) }
    private val weatherIconView: ImageView
        by lazy { findViewById(R.id.main_weather_icon) }
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        weatherApiService
            .getWeather("New York", "[YOUR TOKEN]")
            .enqueue(object : Callback<OpenWeatherMapResponseData> {
                ...
                override fun onResponse(
                    call: Call<OpenWeatherMapResponseData>,
                    response: Response<OpenWeatherMapResponseData>
                ) = handleResponse(response)
            })
    }

    private fun handleResponse(response:
      Response<OpenWeatherMapResponseData>) =
        if (response.isSuccessful) {
            response.body()?.let { validResponse ->
                handleValidResponse(validResponse)
            } ?: Unit
        } else {
        }
```

```
    private fun handleValidResponse(response:
      OpenWeatherMapResponseData) {
        titleView.text = response.locationName
        response.weather.firstOrNull()?.let { weather ->
            statusView.text = weather.status
            descriptionView.text = weather.description
            Glide.with(this)
                .load("https://openweathermap.org/img/wn/
                  ${weather.icon}@2x.png")
                .centerInside()
                .into(weatherIconView)
        }
    }
}
```

There are a few points to note about the preceding code. Firstly, we looked up the views of the layout and kept references to them. We used **`lazy`** to do so on demand rather than immediately. We extracted the handling of the response to a private function to improve the code readability. Doing so also reduces nesting in our code, which in turn reduces the cognitive effort required to understand it. We then implemented a function to handle just the happy path scenario—when the response is successful and contains valid data. After grabbing the location name, we get the first weather response, if any, and grab the status, description, and icon from it. We took a shortcut by using Glide directly for the sake of keeping this example short. However, you should follow the instructions in this chapter and extract this into a **`GlideImageLoader`** class implementing an **`ImageLoader`** interface. Finally, we obtain the icon's URL template from https://openweathermap.org/weather-conditions. We replaced the provided HTTP URL with an HTTPS one for security reasons. Now, when you run the app, you should get an output similar to the following:

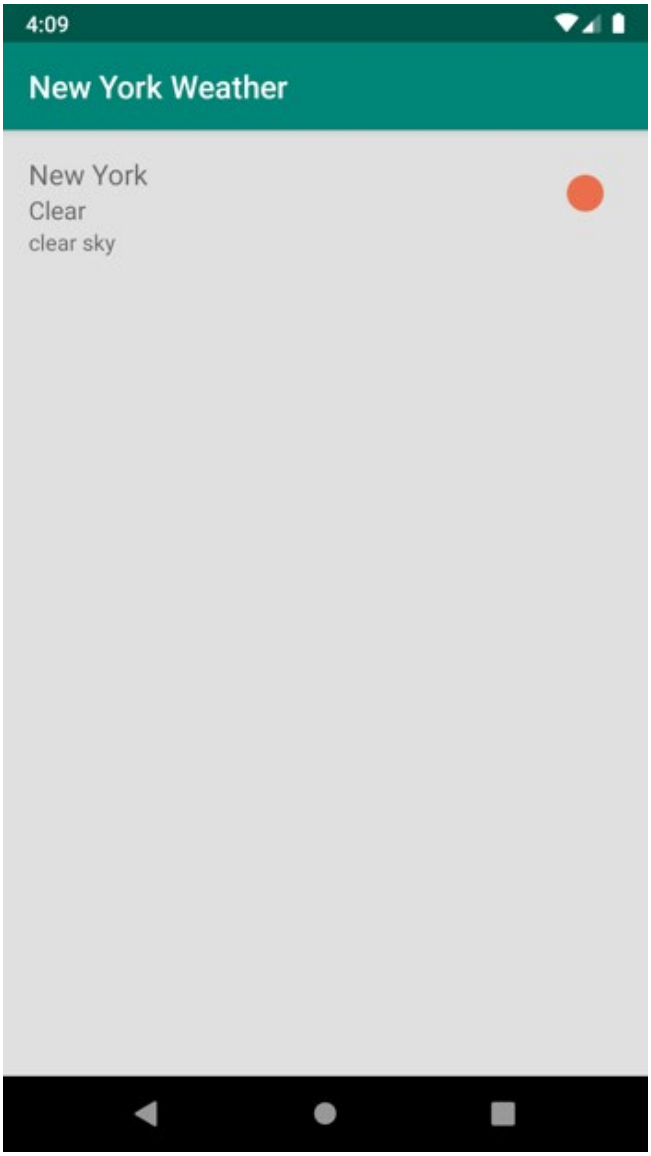

Figure 5.9: Showing the weather and a relevant icon

10. Handle the different possible failure outcomes of making the request:

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        weatherApiService
```

```
            .getWeather("New York", "[YOUR TOKEN]")
            .enqueue(object : Callback<OpenWeatherMapResponseData> {
                override fun onFailure(call:
                  Call<OpenWeatherMapResponseData>, t: Throwable) {
                    showError("Response failed: ${t.message}")
                }
            })
    }

    private fun handleResponse(response:
      Response<OpenWeatherMapResponseData>) =
        if (response.isSuccessful) {
            ...
        } else {
            showError("Response was unsuccessful:
              ${response.errorBody()}")
        }

    private fun showError(message: String) =
        Toast.makeText(this, message, Toast.LENGTH_SHORT)
            .show()
}
```

Remember to replace **`[YOUR TOKEN]`** with the token obtained earlier from https://home.openweathermap.org/users/sign_up.

We can handle the two possible error scenarios—when the call fails altogether due to an exception and when it fails due to an API failure—optionally with a failure message. Both scenarios result in a toast being presented to the user with some information about what went wrong.

CHAPTER 6: RECYCLERVIEW

ACTIVITY 6.01: MANAGING A LIST OF ITEMS

Solution:

1. Create a new app by navigating to **`File`** | **`New`** | **`New Project`**, selecting **`Empty Activity`**, clicking **`Next`**, and filling in the app name, and then save the location and click **`Finish`**.
2. Update the **`activity_main.xml`** layout file by removing **`TextView`** and adding **`RecyclerView`**, two **`EditText`** fields, and two buttons. Your layout should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
...

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/main_recipes_list"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        app:layout_constraintBottom_toTopOf="@id/main_recipe_title"
        app:layout_constraintTop_toTopOf="parent" />

    <EditText
        android:hint="Recipe Title"
        android:id="@+id/main_recipe_title"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toTopOf
          ="@+id/main_recipe_description" />

...

    <Button
        android:id="@+id/main_add_savory_button"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Add Savory"
```

```
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toStartOf
              ="@id/main_add_sweet_button"
            app:layout_constraintStart_toStartOf="parent" />

    ...

    </androidx.constraintlayout.widget.ConstraintLayout>
```

The complete code for this step can be found at http://packt.live/3pgdeZK.

3. Define your model. One approach would be to define a common interface, then implement it for both the title and the recipe models. Create an empty Kotlin file named **`ListItem.kt`** and place it under the **`com.example.recipebook.model`** package. Add the following code to it:

```
interface ListItem

data class TitleUiModel(
    val title: String
) : ListItem

data class RecipeUiModel(
    val title: String,
    val description: String,
    val flavor: Flavor
) : ListItem

enum class Flavor {
    SAVORY,
    SWEET
}
```

4. Add layouts for the flavor title and the recipe title. The flavor title, named **`item_title.xml`**, could look like this:

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

```
    <TextView
        android:id="@+id/title_label"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="16sp"
        android:textStyle="bold"
        tools:text="Sweet" />

</FrameLayout>
```

The recipe title, named **`item_recipe.xml`**, could look like this:

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

    <TextView
        android:id="@+id/recipe_title"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginLeft="16dp"
        android:textSize="12sp"
        tools:text="Tex-Mex Eggs" />

</FrameLayout>
```

5. Create a file named **`ViewHolder.kt`**. Add an abstract view holder as a superclass for the title and recipe view holders, and extend it to implement the title and recipe view holders:

```
class TitleViewHolder(containerView: View) :
  BaseViewHolder(containerView) {
    private val titleView: TextView
        by lazy { containerView.findViewById(R.id.title_label) }

    override fun bindData(listItem: ListItem) {
        titleView.text = (listItem as TitleUiModel).title
    }
```

```
}

class RecipeViewHolder(
    containerView: View
) : BaseViewHolder(containerView) {
    private val titleView: TextView
        by lazy { containerView.findViewById(R.id.recipe_title) }

    override fun bindData(listItem: ListItem) {
        titleView.text = (listItem as RecipeUiModel).title
    }
}
```

6. Add your adapter by implementing **`RecyclerView.Adapter`**:

```
private const val VIEW_TYPE_TITLE = 0
private const val VIEW_TYPE_RECIPE = 1

class RecipesAdapter(
    private val layoutInflater: LayoutInflater) :
      RecyclerView.Adapter<BaseViewHolder>() {
    private val savoryTitle = TitleUiModel("Savory")
    private val sweetTitle = TitleUiModel("Sweet")
    private val listItems = mutableListOf<ListItem>(savoryTitle,
      sweetTitle)

    override fun getItemViewType(position: Int) =
      when (listItems[position]) {
        is TitleUiModel -> VIEW_TYPE_TITLE
        is RecipeUiModel -> VIEW_TYPE_RECIPE
        else -> throw IllegalStateException("Unexpected data type at
          $position")
    }

    override fun onCreateViewHolder(parent: ViewGroup,
      viewType: Int): BaseViewHolder =
        when (viewType) {
            VIEW_TYPE_TITLE -> TitleViewHolder(
                layoutInflater.inflate(R.layout.item_title,
```

```
                    parent, false)
            )
            VIEW_TYPE_RECIPE -> RecipeViewHolder(
                layoutInflater.inflate(R.layout.item_recipe, parent,
                  false)
            )
            else -> throw IllegalStateException("Unexpected view type
              $viewType")
        }

    override fun getItemCount() = listItems.size

    override fun onBindViewHolder(holder: BaseViewHolder,
      position: Int) = holder.bindData(listItems[position])
}
```

7. Update **`RecipeViewHolder`** to handle clicks:

```
class RecipeViewHolder(
    containerView: View,
    private val onClickListener: OnClickListener
) : BaseViewHolder(containerView) {
    private val titleView: TextView
            by lazy { containerView.findViewById(R.id.recipe_title) }

    override fun bindData(listItem: ListItem) {
        titleView.text = (listItem as RecipeUiModel).title
        titleView.setOnClickListener {
            onClickListener.onClick(listItem)
        }
    }

    interface OnClickListener {
        fun onClick(recipe: RecipeUiModel)
    }
}
```

8. Add an **`OnClickListener`** interface to the adapter and include it in the constructor:

```
class RecipesAdapter(
    private val layoutInflater: LayoutInflater,
    private val onClickListener: OnClickListener
) : RecyclerView.Adapter<BaseViewHolder>() {
    ...
    interface OnClickListener {
        fun onItemClick(recipe: RecipeUiModel)
    }
}
```

Revise the **`RecipeViewHolder`** construction:

```
RecipeViewHolder(
    layoutInflater.inflate(R.layout.item_recipe, parent, false),
    object : RecipeViewHolder.OnClickListener {
        override fun onClick(recipe: RecipeUiModel) {
            onClickListener.onItemClick(recipe)
        }
    }
)
```

9. Update the adapter to support adding new recipes. Make sure savory recipes appear under the **`Savory`** title, and sweet under the **`Sweet`** title:

```
private val savoryTitle = TitleUiModel("Savory")
private val sweetTitle = TitleUiModel("Sweet")
private val listItems =
  mutableListOf<ListItem>(savoryTitle, sweetTitle)

fun addRecipe(recipe: RecipeUiModel) {
    val insertionIndex = listItems.indexOf(when (recipe.flavor) {
        Flavor.SAVORY -> savoryTitle
        Flavor.SWEET -> sweetTitle
    }) + 1
    listItems.add(insertionIndex, recipe)
    notifyItemInserted(insertionIndex)
}
```

10. Also, update the adapter by adding swipe behavior:

```
class RecipesAdapter(...) : RecyclerView.Adapter<BaseViewHolder>() {
    val swipeToDeleteCallback = SwipeToDeleteCallback()
    ...
    inner class SwipeToDeleteCallback :
        ItemTouchHelper.SimpleCallback(0, ItemTouchHelper.LEFT
          or ItemTouchHelper.RIGHT) {
        override fun onMove(
            recyclerView: RecyclerView,
            viewHolder: RecyclerView.ViewHolder,
            target: RecyclerView.ViewHolder
        ): Boolean = false

        override fun getMovementFlags(
            recyclerView: RecyclerView,
            viewHolder: RecyclerView.ViewHolder
        ) = if (viewHolder is RecipeViewHolder) {
            makeMovementFlags(
                ItemTouchHelper.ACTION_STATE_IDLE,
                ItemTouchHelper.LEFT or ItemTouchHelper.RIGHT
            ) or makeMovementFlags(
                ItemTouchHelper.ACTION_STATE_SWIPE,
                ItemTouchHelper.LEFT or ItemTouchHelper.RIGHT
            )
        } else {
            0
        }

        override fun onSwiped(viewHolder: RecyclerView.ViewHolder,
          direction: Int) {
            val position = viewHolder.adapterPosition
            listItems.removeAt(position)
            notifyItemRemoved(position)
        }
    }
}
```

11. Lastly, update your activity to handle the recipe-adding buttons, adapter clicks, and swipes:

```
class MainActivity : AppCompatActivity() {
    private val recipesList: RecyclerView
            by lazy { findViewById(R.id.main_recipes_list) }
    private val addSavoryButton: View
            by lazy { findViewById(R.id.main_add_savory_button) }
    private val addSweetButton: View
            by lazy { findViewById(R.id.main_add_sweet_button) }
    private val titleView: TextView
            by lazy { findViewById(R.id.main_recipe_title) }
    private val descriptionView: TextView
            by lazy { findViewById(R.id.main_recipe_description) }

    private val recipesAdapter by lazy {
        RecipesAdapter(
            layoutInflater,
            object : RecipesAdapter.OnClickListener {
                override fun onItemClick(recipe: RecipeUiModel) {
                    val builder = AlertDialog
                      .Builder(this@MainActivity)
                    builder.setMessage(recipe.description)
                        .setPositiveButton("OK", null)
                        .create()
                        .show()
                }
            }
        )
    }

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

        recipesList.apply {
            adapter = recipesAdapter
            layoutManager = LinearLayoutManager
              (this@MainActivity, VERTICAL, false)
```

```
            val itemTouchHelper =
                ItemTouchHelper(recipesAdapter.swipeToDeleteCallback)
            itemTouchHelper.attachToRecyclerView(this)
        }

        addSavoryButton.setOnClickListener {
            addRecipeAndClearForm(Flavor.SAVORY)
        }

        addSweetButton.setOnClickListener {
            addRecipeAndClearForm(Flavor.SWEET)
        }
    }

    private fun addRecipeAndClearForm(flavor: Flavor) {
        val title = titleView.text.toString().trim()
        val description = descriptionView.text.toString().trim()
        if (title.isEmpty() || description.isEmpty()) return

        recipesAdapter.addRecipe(
            RecipeUiModel(title, description, flavor)
        )
        titleView.text = ""
        descriptionView.text = ""
    }
}
```

Note how we also added validation to make sure users can't add empty recipes. This can be handled in different ways:

- You could disable the button until a valid date is provided. This probably provides a preferable user experience, but requires more code.
- You could prevent adding invalid items. This is the shortest solution, which is why we opted for it here. However, it doesn't provide a great user experience because the user has to guess why the button doesn't work.

- You could present the user with an error message via a dialog for a view. This is somewhat better than the option we opted for and is still easier to implement than disabling the button:

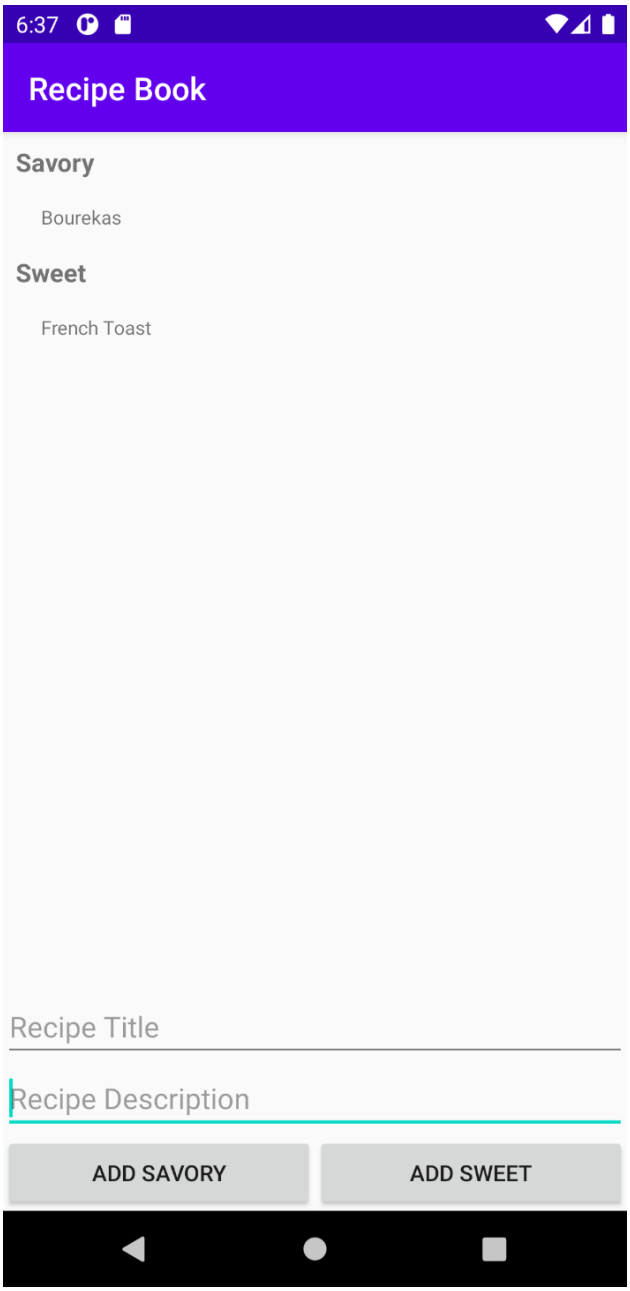

Figure 6.19: The Recipe Book app with some recipes

CHAPTER 7: ANDROID PERMISSIONS AND GOOGLE MAPS

ACTIVITY 7.01: CREATING AN APP TO FIND THE LOCATION OF A PARKED CAR

Solution:

1. Create a Google Maps Activity app named **`Find My Car`**.
2. Obtain an API key for the app by following the instructions in the link provided in your **`google_maps_api.xml`** file located under **`app/res/values`**. Update your **`google_maps_api.xml`** file by replacing **`YOUR_KEY_HERE`** with the key you obtained in the following string:

```
<string name="google_maps_key" templateMergeStrategy="preserve"
  translatable="false">YOUR_KEY_HERE</string>
```

3. Update its main layout to show a button at the bottom with an **`I'm parked here`** label:

```
<androidx.constraintlayout.widget.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:map="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment
        android:id="@+id/map"
        android:name="com.google.android.gms.maps.SupportMapFragment"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        map:layout_constraintBottom_toTopOf=
          "@+id/maps_mark_location_button"
        map:layout_constraintTop_toTopOf="parent"
        tools:context=".MapsActivity" />

    <Button
        android:id="@+id/maps_mark_location_button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="I'm parked here"
        map:layout_constraintBottom_toBottomOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

The output will be as follows:

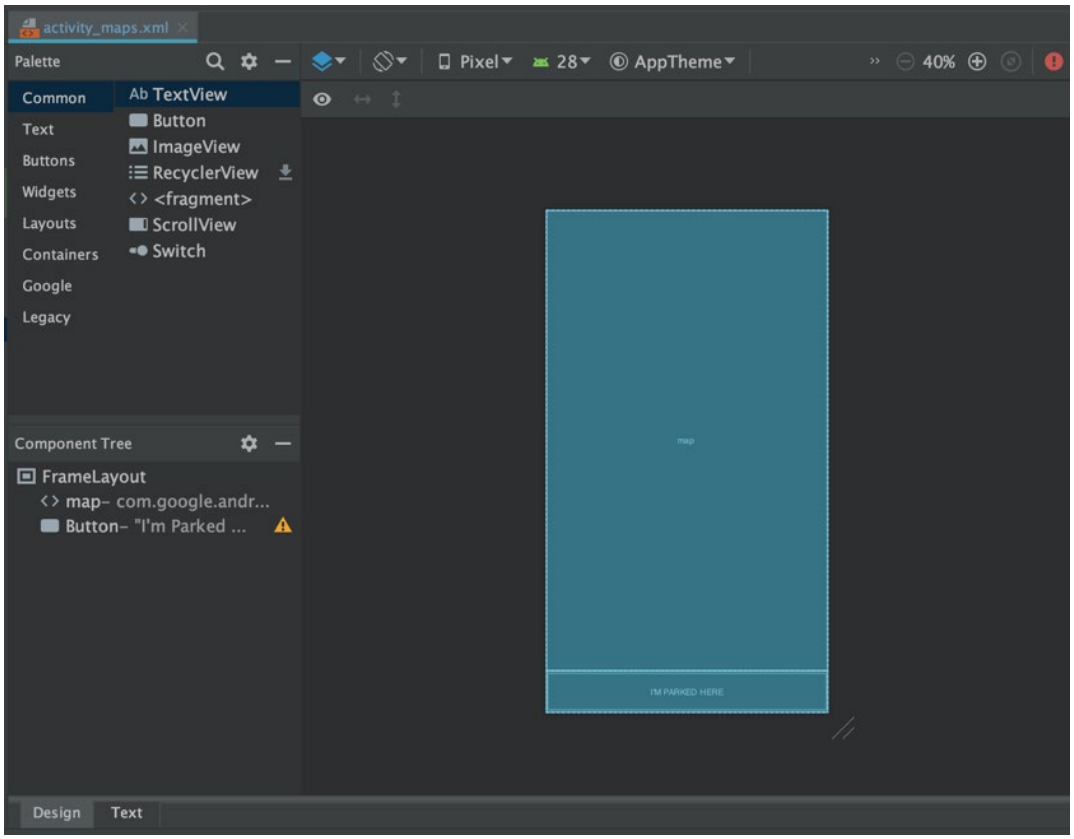

Figure 7.16: The main layout

4. Include the Google Play location service in your app's **`build.gradle`** file:

```
dependencies {
    implementation "com.google.android.gms:play-services-
location:17.1.0"
    ...
}
```

5. Request the user's permission to access their location. Present the rationale if the SDK tells your app it is required:

```
private const val PERMISSION_CODE_REQUEST_LOCATION = 1

class MapsActivity : AppCompatActivity(), OnMapReadyCallback {
    ...
```

```
override fun onResume() {
    super.onResume()

    val hasLocationPermissions = getHasLocationPermission()
}

private fun getHasLocationPermission() = if (
    ContextCompat.checkSelfPermission(
        this, Manifest.permission.ACCESS_FINE_LOCATION
    ) == PackageManager.PERMISSION_GRANTED
) {
    true
} else {
        if (ActivityCompat.shouldShowRequestPermissionRationale(
            this, Manifest.permission.ACCESS_FINE_LOCATION
        )
    ) {
        showPermissionRationale { requestLocationPermission() }
    } else {
        requestLocationPermission()
    }
    false
}

private fun showPermissionRationale(positiveAction: () -> Unit) {
    AlertDialog.Builder(this)
        .setTitle("Location permission")
        .setMessage("We need your permission to find your
          current location")
        .setPositiveButton(
            "OK"
        ) { _, _ -> positiveAction() }
        .create()
        .show()
}

private fun requestLocationPermission() {
    ActivityCompat.requestPermissions(
        this,
        arrayOf(Manifest.permission.ACCESS_FINE_LOCATION),
        PERMISSION_CODE_REQUEST_LOCATION
```

```
            )
        }
        ...
    }
```

6. Obtain the user's location and place a pin on the map at that location:

```
class MapsActivity : AppCompatActivity(), OnMapReadyCallback {
    private val fusedLocationProviderClient by lazy {
        LocationServices.getFusedLocationProviderClient(this)
    }

    ...

    override fun onResume() {
        ...
        if (hasLocationPermissions) {
            getLastLocation()
        }
    }

    ...

    @SuppressLint("MissingPermission")
    private fun getLastLocation() {
        fusedLocationProviderClient.lastLocation
            .addOnSuccessListener { location: Location? ->
                location?.let {
                    val userLocation = LatLng(location.latitude,
                      location.longitude)
                    updateMapLocation(userLocation)
                    addMarkerAtLocation(userLocation, "You")
                }
            }
    }

    private fun updateMapLocation(location: LatLng) {
        mMap.moveCamera(CameraUpdateFactory
          .newLatLngZoom(location, 7f))
    }

    private fun addMarkerAtLocation(location: LatLng, title: String)
```

```
{
        mMap.addMarker(MarkerOptions().title(title)
          .position(location))
    }

    override fun onRequestPermissionsResult(
        requestCode: Int,
        permissions: Array<out String>,
        grantResults: IntArray
    ) {
        super.onRequestPermissionsResult(requestCode, permissions,
          grantResults)

        when (requestCode) {
            PERMISSION_CODE_REQUEST_LOCATION -> getLastLocation()
        }
    }
}
```

7. Add a car icon to your project from the Android Studio Clip Art library:

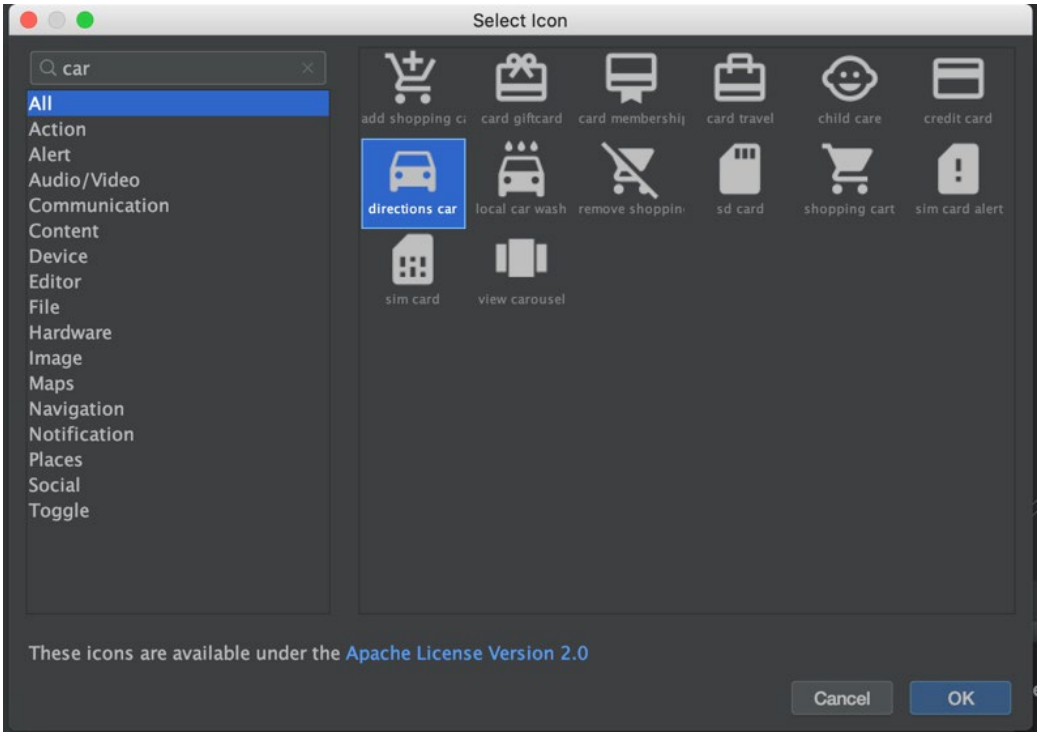

Figure 7.17: Choosing the car icon

8. Add a function to load vector drawable assets to your **`MapsActivity`** class:

```
    private fun getBitmapDescriptorFromVector(@DrawableRes
      vectorDrawableResourceId: Int): BitmapDescriptor? {
        val bitmap =
            ContextCompat.getDrawable(this,
              vectorDrawableResourceId)?.let { vectorDrawable ->
                vectorDrawable
                    .setBounds(0, 0, vectorDrawable.intrinsicWidth,
                      vectorDrawable.intrinsicHeight)

                val drawableWithTint = DrawableCompat
                  .wrap(vectorDrawable)
                DrawableCompat.setTint(drawableWithTint, Color.RED)

                val bitmap = Bitmap.createBitmap(
                    vectorDrawable.intrinsicWidth,
                    vectorDrawable.intrinsicHeight,
                    Bitmap.Config.ARGB_8888
                )
                val canvas = Canvas(bitmap)
                drawableWithTint.draw(canvas)
                bitmap
            }
        return BitmapDescriptorFactory.fromBitmap(bitmap).also {
            bitmap?.recycle()
        }
    }
```

9. At the top of the **`MapsActivity`** class, define two markers, one for the user and one for their car:

```
    private lateinit var mMap: GoogleMap

    private var userMarker: Marker? = null
    private var carMarker: Marker? = null
```

10. Update your **`getLastLocation`** function to take a lambda to execute once a location is obtained:

```
    @SuppressLint("MissingPermission")
    private fun getLastLocation(onLocation:
      (location: Location) -> Unit) {
```

```
        fusedLocationProviderClient.lastLocation
            .addOnSuccessListener { location: Location? ->
                location?.let { onLocation(it) }
            }
    }
```

11. Update your existing calls in **`onRequestPermissionsResult`** and **`onResume`** to pass in a **`lambda`** function:

```
    override fun onRequestPermissionsResult(
        requestCode: Int,
        permissions: Array<out String>,
        grantResults: IntArray
    ) {
        super.onRequestPermissionsResult(requestCode, permissions,
          grantResults)

       when (requestCode) {
            PERMISSION_CODE_REQUEST_LOCATION ->
              getLastLocation { location ->
                val userLocation = LatLng(location.latitude,
                  location.longitude)
                updateMapLocation(userLocation)
                userMarker = addMarkerAtLocation(userLocation, "You")
            }
        }
    }

    override fun onResume() {
        if (hasLocationPermissions) {
            getLastLocation { location ->
                val userLocation = LatLng(location.latitude,
                  location.longitude)
                updateMapLocation(userLocation)
                userMarker = addMarkerAtLocation(userLocation, "You")
            }
        }
    }
```

12. Still in **`MapsActivity`**, introduce a function to add a marker to the map at a given location:

```
    private fun addMarkerAtLocation(
        location: LatLng,
```

```
            title: String,
            markerIcon: BitmapDescriptor? = null
        ) = mMap.addMarker(
            MarkerOptions()
                .title(title)
                .position(location)
                .apply {
                    markerIcon?.let { icon(markerIcon) }
                }
        )
```

13. Now, add a function to mark the location of the car:

```
        private fun markParkedCar() {
            getLastLocation { location ->
                val userLocation = LatLng(location.latitude,
                  location.longitude)
                userMarker?.remove()
                carMarker?.remove()
                updateMapLocation(userLocation)
                carMarker = addMarkerAtLocation(
                    userLocation,
                    "Your Car",
                    getBitmapDescriptorFromVector
                      (R.drawable.ic_baseline_directions_car_24)
                )
                userMarker = addMarkerAtLocation(userLocation, "You")
                saveLocation(userLocation)
            }
        }
```

14. To receive user clicks on the **`I'm parked here`** button, you first need to keep a reference to the button. Do so by adding the field below to your MapsActivity class:

```
    class MapsActivity : AppCompatActivity(), OnMapReadyCallback {
        private val markLocationButton: View by lazy {
            findViewById(R.id.maps_mark_location_button)
        }
```

15. Lastly, when the user clicks the **`I'm parked here`** button, add or move the car icon to the user's current location:

```
class MapsActivity : AppCompatActivity(), OnMapReadyCallback {

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        markLocationButton.setOnClickListener {
            if (getHasLocationPermission()) {
                markParkedCar()
            }
        }
    }
}
```

16. Store the selected location in **`SharedPreferences`**. This function, placed in your activity, will help:

```
private fun saveLocation(latLng: LatLng) =
    getPreferences(MODE_PRIVATE)?.edit()?.apply {
        putString("latitude", latLng.latitude.toString())
        putString("longitude", latLng.longitude.toString())
        apply()
    }
```

17. Upon resuming the activity, restore any saved location and place the car icon at that location. You can use this function:

```
private fun restoreLocation() =
    getPreferences(Context.MODE_PRIVATE)?.let { sharedPreferences ->
        val latitude =
            sharedPreferences.getString("latitude", null)
            ?.toDoubleOrNull()?: return null
        val longitude =
            sharedPreferences.getString("longitude", null)
              ?.toDoubleOrNull()?: return null
        LatLng(latitude, longitude)
    }
```

18. Call **`restoreLocation`** from **`onMapReady(GoogleMap)`** to make sure you have a map to add a marker to:

```
override fun onMapReady(googleMap: GoogleMap) {
    ...

    restoreLocation()?.let { userLocation ->
        carMarker = addMarkerAtLocation(
            userLocation,
            "Your Car", getBitmapDescriptorFromVector
              (R.drawable.ic_baseline_directions_car_24)
        )
        userMarker = addMarkerAtLocation(userLocation, "You")
    }
}
```

CHAPTER 8: SERVICES, WORKMANAGER, AND NOTIFICATIONS

ACTIVITY 8.01: REMINDER TO DRINK WATER

Solution:

1. Create an empty **`Activity`** project. Name your app **`My Water Tracker`**, and set its package name to **`com.example.mywatterttracker`**.
2. Add the **`FOREGROUND_SERVICE`** permission to your **`AndroidManifest.xml`** file:

```
<manifest ...>

    <uses-permission android:name="android.permission
      .FOREGROUND_SERVICE"/>

    <application ...>
```

3. Create a new **`Service`** called **`WaterTrackingService`**:

```
class WaterTrackingService : Service() {
    override fun onBind(intent: Intent?): IBinder? = null
}
```

4. Add a private mutable (**`var`**) **`fluidBalanceMilliliters`** field to your class. Set its initial value to **`0f`** (the **`f`** tells Kotlin this is a float value). This field will store the current user's fluid balance. Also, add late initialization fields for **`NotificationCompat.Builder`** and **`serviceHandler`**, to be used later to construct the notification and to execute in the background, respectively:

```
class WaterTrackingService : Service() {
    private var fluidBalanceMilliliters = 0f
    private lateinit var notificationBuilder:
      NotificationCompat.Builder
    private lateinit var serviceHandler: Handler

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

5. Add a companion object to your **`WaterTrackingService`** class with two constants—one for the notification ID and the other for the key used to read the extra intake intent data:

```
companion object {
    const val EXTRA_INTAKE_AMOUNT_MILLILITERS = "intake"
    private const val NOTIFICATION_ID = 0x3A7A
}
```

6. Add the functions required to set up the notification:

```
private fun getPendingIntent() =
    PendingIntent.getActivity(this, 0, Intent(this,
      MainActivity::class.java), 0)

@RequiresApi(Build.VERSION_CODES.O)
private fun createNotificationChannel(): String {
    val channelId = "FluidBalanceTracking"
    val channelName = "Fluid Balance Tracking"
    val channel =
        NotificationChannel(channelId, channelName,
          NotificationManager.IMPORTANCE_DEFAULT)
    val service = getSystemService(Context.NOTIFICATION_SERVICE) as
      NotificationManager
    service.createNotificationChannel(channel)
    return channelId
}

private fun getNotificationBuilder(pendingIntent: PendingIntent,
  channelId: String) =
    NotificationCompat.Builder(this, channelId)
        .setContentTitle("Tracking your fluid balance")
        .setContentText("Tracking")
        .setSmallIcon(R.drawable.ic_launcher_foreground)
        .setContentIntent(pendingIntent)
        .setTicker("Fluid balance tracking started")
```

7. Now add a function to start the foreground service:

```
private fun startForegroundService(): NotificationCompat.Builder {
    val pendingIntent = getPendingIntent()

    val channelId = if (Build.VERSION.SDK_INT >=
      Build.VERSION_CODES.O) {
        createNotificationChannel()
    } else {
        ""
    }

    val notificationBuilder = getNotificationBuilder(pendingIntent,
      channelId)
    startForeground(NOTIFICATION_ID, notificationBuilder.build())
    return notificationBuilder
}
```

8. Add a function to update the water balance. Because we operate on two different threads, don't forget to synchronize the call:

```
private fun addToFluidBalance(amountMilliliters: Float) {
    synchronized(this) {
        fluidBalanceMilliliters += amountMilliliters
    }
}
```

9. Create a function to update the fluid balance every 5 seconds:

```
private fun updateFluidBalance() {
    serviceHandler.postDelayed({
        updateFluidBalance()
        addToFluidBalance(-0.144f)
        notificationBuilder.setContentText(
            "Your fluid balance: %.2f".format(fluidBalanceMilliliters)
        )
        startForeground(NOTIFICATION_ID, notificationBuilder.build())
    }, 5000L)
}
```

A few things to note here. We made a very inaccurate assumption that, since a human loses **`2500ml`** in a day, they lose roughly **`104.16ml`** per hour, **`1.736ml`** per minute, and so **`0.029ml`** per second, or **`0.144ml`** every 5 seconds (the value we used). We also ignored the fact that over time, we would probably get even less accurate values, as each call would not be precisely 5 seconds apart from the previous one (handlers are inaccurate that way). We could have used a field to store the last time the method was called and used that to get a more accurate result, but that would have complicated our example. We should have also used a string resource, and not a hardcoded string. Lastly, we should have used constants or variables instead of the magic numbers (**`-0.144f, 5000L`**). Again, for the sake of simplicity, we didn't. But you really should.

10. Next, make sure that when the service is created, the service is started in the foreground, a reference is stored to **`NotificationCompat.Builder`**, **`serviceHandler`** is instantiated, and you start updating the fluid balance:

```
override fun onCreate() {
    super.onCreate()

    notificationBuilder = startForegroundService()
    val handlerThread = HandlerThread("RouteTracking").apply
      { start() }
    serviceHandler = Handler(handlerThread.looper)
    updateFluidBalance()
}
```

11. Now, handle the addition of fluids (such as drinking a glass of water) by overriding **`onStartCommand(Intent?, Int, Int)`** and reading the provided **`Intent`**, if available:

```
override fun onStartCommand(intent: Intent?, flags: Int, startId:
  Int): Int {
    val returnValue = super.onStartCommand(intent, flags, startId)

    val intakeAmountMilliliters =
      intent?.getFloatExtra(EXTRA_INTAKE_AMOUNT_MILLILITERS, 0f)
    intakeAmountMilliliters?.let {
        addToFluidBalance(it)
    }

    return returnValue
}
```

12. Lastly, clean up when the service is destroyed, stopping the handler loop:

```
override fun onDestroy() {
    serviceHandler.removeCallbacksAndMessages(null)
}
```

13. To launch the service, you must first register it in your app's **`AndroidManifest.xml`** file:

```
<application ...>

    <service
        android:name=".WaterTrackingService"
        android:enabled="true"
        android:exported="true"/>

    <activity android:name=".MainActivity">
```

14. Then, in your **`MainActivity`** class, you can add a function to launch the service, with an optional intake value:

```
private fun launchTrackingService(intakeAmount: Float = 0f) {
    val serviceIntent =
      Intent(this, WaterTrackingService::class.java).apply {
        putExtra(EXTRA_INTAKE_AMOUNT_MILLILITERS, intakeAmount)
    }
    ContextCompat.startForegroundService(this, serviceIntent)
}
```

15. After that, call it from the **`onCreate(Bundle?)`** function, omitting the **`intakeAmount`** value:

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

    launchTrackingService()
}
```

16. To allow users to let the app know that they drank a glass of water, replace the **`Hello World! TextView`** in **`activity_main.xml`** with a button, updating its text and assigning it an ID like so:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout ...>

    <Button
        android:id="@+id/main_water_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Drank a Glass of Water"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Remember, you should not really hardcode text values. Instead, use **`strings.xml`**.

17. Hold a reference to your newly added view in your **`MainActivity`** class:

```
class MainActivity : AppCompatActivity() {
    private val waterButton: View by lazy {
        findViewById(R.id.main_water_button)
    }
    ...
}
```

18. Now, in your **`MainActivity`** class, attach a click listener to your button:

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

    launchTrackingService()

    waterButton.setOnClickListener {
        launchTrackingService(250f)
    }
}
```

    Remember to use constants rather than magic numbers (**`250f`** could be stored as **`GLASS_WATER_AMOUNT_MILLILITER`**, for example). This makes the code easier to understand and easier to maintain.

19. Run your app. As soon as the app starts, you should see a notification added to your status bar. It's a sticky notification, so it will keep running even if you stopped your **`MainActivity`**. It will be removed if, for example, you force stopped the app, uninstalled it, or restarted your device. The notification will count down from 5 and refresh every 5 seconds. If you click the button, the next refresh will reflect the added fluids.

CHAPTER 9: UNIT TESTS AND INTEGRATION TESTS WITH JUNIT, MOCKITO, AND ESPRESSO

ACTIVITY 9.01: DEVELOPING WITH TDD

Solution:

1. Let's start by adding the necessary files in Gradle:

```
    implementation "androidx.recyclerview:recyclerview:1.1.0"
    implementation 'androidx.test.espresso:espresso-core:3.3.0'
    testImplementation 'junit:junit:4.13.1'
    testImplementation 'org.mockito:mockito-core:3.6.0'
    testImplementation 'com.nhaarman.mockitokotlin2
      :mockito-kotlin:2.2.0'
    testImplementation 'org.robolectric:robolectric:4.4'
    testImplementation 'androidx.test.ext:junit:1.1.2'
    testImplementation 'androidx.test.espresso:espresso-intents
      :3.3.0'
    testImplementation 'com.android.support.test
      .espresso:espresso-contrib:3.3.0'
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation 'androidx.test
      .espresso:espresso-core:3.3.0'
    androidTestImplementation 'androidx.test:rules:1.3.0'
    androidTestImplementation 'com.android.support.test
      .espresso:espresso-contrib:3.3.0'
```

2. We can put these strings in the **`res/strings.xml`** file, which will be used across the application:

```
    <string name="submit">Submit</string>
    <string name="item_x">Item %d</string>
    <string name="you_clicked_y">You clicked %s</string>
```

3. Let's create a layout called **`activity_1.xml`**:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
  android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
```

```
    <EditText
        android:id="@+id/activity_1_edit_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="number" />

    <Button
        android:id="@+id/activity_1_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:text="@string/submit" />
</LinearLayout>
```

4. Next, let's create the **`Activity1`** class:

```
class Activity1 : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_1)
    }
}
```

5. Repeat the process for the **`activity_2.xml`** layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
  android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/activity_2_recycler_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

6. The **`Activity2`** class will be as follows:

```
class Activity2 : AppCompatActivity() {
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_2)
    }
}
```

7. Repeat the same process for the **`activity_3.xml`** layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
  android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:id="@+id/activity_3_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center" />

</LinearLayout>
```

8. The **`Activity3`** class will be as follows:

```
class Activity3 : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_3)
    }
}
```

9. Add all three activities to **`AndroidManifest.xml`**:

```
<application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".Activity1">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
```

```
                <category android:name=
                  "android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".Activity2" />
        <activity android:name=".Activity3" />
    </application>
```

10. Let's start creating the robots in the **`androidTest`** directory. For **`Activity1Robot`**, we have the following:

```
class Activity1Robot {

    fun insertText(text: String): Activity1Robot {
        onView(withId(R.id.activity_1_edit_text))
          .perform(replaceText(text))
        return this
    }

    fun submit(): Activity1Robot {
        onView(withId(R.id.activity_1_button)).perform(click())
        return this
    }
}
```

11. Because **`Activity2`** has a list of items that require special handling, we will next define **`Activity3Robot`** because it has a similar complexity to **`Activity1`**:

```
class Activity3Robot {

    private val myApplication =
      ApplicationProvider.getApplicationContext<Application>()

    fun verifyText(expectedItemText: String): Activity3Robot {
        onView(withId(R.id.activity_3_text_view))
         .check(matches(withText(myApplication
           .getString(R.string.you_clicked_y, expectedItemText))))
        return this
    }
}
```

12. Now, let's look at **`Activity2Robot`**. Here, we need to assert the total number of items in **`RecyclerView`**. The support library provides no method for this, but we can write a custom implementation that will check this. We can write a class that will implement **`ViewAssertion`** and there we will assert the count of **`RecyclerView`**:

```
class RecyclerViewItemCountAssertion(private val expectedCount: Int)
  : ViewAssertion {

    override fun check(view: View, noViewFoundException:
      NoMatchingViewException?) {
        if (view is RecyclerView) {
            val adapter = view.adapter
            assertEquals(expectedCount, adapter!!.itemCount)
        }
    }
}
```

13. We can now create a Kotlin file in the **`androidTest`** folder, which we will provide with our own view assertion to keep it consistent with the Espresso syntax. In the **`MyViewAssertions.kt`** file, we will have the following code:

```
fun checkRecyclerViewItems(count: Int): ViewAssertion {
    return RecyclerViewItemCountAssertion(count)
}
```

14. Finally, we can create **`Activity2Robot`**:

```
class Activity2Robot {
    private val myApplication =
      ApplicationProvider.getApplicationContext<Application>()

    fun verifyItemNumber(expected: Int): Activity2Robot {
        onView(withId(R.id.activity_2_recycler_view))
          .check(checkRecyclerViewItems(expected))
        return this
    }

    fun verifyItemText(itemPosition: Int): Activity2Robot {
        onView(withId(R.id.activity_2_recycler_view))
          .perform(scrollToPosition
            <RecyclerView.ViewHolder>(itemPosition))
        return this
```

```
        }

        fun clickOnItem(itemPosition: Int): Activity2Robot {
            onView(withId(R.id.activity_2_recycler_view))
              .perform(scrollToPosition
                <RecyclerView.ViewHolder>(itemPosition))
            onView(withId(R.id.activity_2_recycler_view))
              .perform(actionOnItemAtPosition<RecyclerView.ViewHolder>
                (itemPosition, click()))
            return this
        }
    }
```

15. Now, let's create our test suite, which we will name **`UiTest`**. If we run this test, we will indeed see that it will fail:

```
    @LargeTest
    @RunWith(AndroidJUnit4::class)
    class UiTest {

        @JvmField
        @Rule
        var activityRule: ActivityTestRule<Activity1>
          = ActivityTestRule(Activity1::class.java)
        private val myApplication
          = ApplicationProvider.getApplicationContext<Application>()

        @Test
        fun testMyFlow() {
            val numberOfItems = 5

            Activity1Robot()
                .insertText(numberOfItems.toString())
                .submit()

            val selectedPosition = 3
            Activity2Robot()
                .verifyItemNumber(numberOfItems)
                .verifyItemText(selectedPosition)
                .clickOnItem(selectedPosition)

            val expectedTest =
              myApplication.getString(R.string.item_x,
              (selectedPosition + 1))
```

```
            Activity3Robot()
                .verifyText(expectedTest)
        }

    }
```

16. Now, let's create the **`Application`** class and replace the references in the robots and **`UiTest`** with the new **`Application`** class:

```
    class MyApplication : Application() {

        override fun onCreate() {
            super.onCreate()
        }
    }
```

17. Let's update the **`AndroidManifest.xml`** file to add the new **`Application`** class to the **`application`** tag with the **`android:name`** attribute.

18. Let's move on to the integration tests. Make sure to set up the Robolectric configurations. Before we write the tests, we will need to update **`Activity2`** and **`Activity3`** with the **`newIntent`** methods in order to properly assert the tests. We will also need to create the **`Item`** data class, which will be used to hold the data for **`Activity2`**:

```
     @Parcelize
    data class Item(val text: String) : Parcelable
```

19. **`Activity2`** will be updated as follows:

```
    Activity2 : AppCompatActivity() {

        companion object {
            const val EXTRA_ITEM_COUNT = "EXTRA_ITEM_COUNT"

            fun newIntent(context: Context, itemCount: Int)
              = Intent(context, Activity2::class.java)
                .putExtra(EXTRA_ITEM_COUNT, itemCount)
        }
    ...
    }
```

20. **`Activity3`** will be updated as follows:

```
class Activity3 : AppCompatActivity() {

    companion object {
        const val EXTRA_ITEM = "EXTRA_ITEM"

        fun newIntent(context: Context, item: Item) = Intent(context,
          Activity3::class.java).putExtra(EXTRA_ITEM, item)

...
}
```

21. Let's move on to the integration tests. For **`Activity1Test`**, we have the following code in which we insert the number **`5`** into a text field, click the button, and verify that **`Activity2`** will be opened and will have as input the number **`5`**:

```
@MediumTest
@RunWith(AndroidJUnit4::class)
class Activity1Test {

    @JvmField
    @Rule
    val rule = IntentsTestRule(Activity1::class.java)

    @Test
    fun 'test enter number and submit'() {
        onView(withId(R.id.activity_1_edit_text))
          .perform(replaceText(5.toString()))
        onView(withId(R.id.activity_1_button)).perform(click())
        intended(allOf(hasComponent(hasShortClassName(".Activity2")),
          hasExtra(Activity2.EXTRA_ITEM_COUNT, 5)))
    }
}
```

22. For **`Activity2Test`**, we will launch **`Activity2`** with five items, then verify the text on the fourth item, clicking that item and verifying that **`Activity3`** will be opened with the correct text as input:

```
@MediumTest
@RunWith(AndroidJUnit4::class)
class Activity2Test {

    @JvmField
    @Rule
    val rule = IntentsTestRule(Activity2::class.java, false, false)
    private val itemCount = 5

    @Before
    fun setUp() {
        rule.launchActivity(Intent()
          .putExtra(Activity2.EXTRA_ITEM_COUNT, itemCount))
    }

    @Test
    fun 'test click opens activity 3'() {
        val position = 3
        val itemText = rule.activity.getString(R.string.item_x,
          (position + 1))
        onView(withText(itemText)).check(matches(isDisplayed()))
        onView(withId(R.id.activity_2_recycler_view))
          .perform(scrollToPosition<RecyclerView
          .ViewHolder>(position))
        onView(withId(R.id.activity_2_recycler_view))
          .perform(actionOnItemAtPosition<RecyclerView.ViewHolder>
          (position, click()))
        intended(allOf(hasComponent(hasShortClassName(".Activity3")),
          hasExtra(Activity3.EXTRA_ITEM, Item(itemText))))

    }
}
```

23. For **`Activity3Test`**, we verify that **`Activity3`** is opened with certain text as input, and then verify that the text is displayed on the screen:

```
@MediumTest
@RunWith(AndroidJUnit4::class)
class Activity3Test {

    @JvmField
    @Rule
    val rule = IntentsTestRule(Activity3::class.java, false, false)

    private val item = Item("Text to display")

    @Before
    fun setUp() {
        rule.launchActivity(Intent()
          .putExtra(Activity3.EXTRA_ITEM, item))
    }

    @Test
    fun 'test displays correct text'() {
        onView(withId(R.id.activity_3_text_view))
          .check(matches(withText(rule.activity
          .getString(R.string.you_clicked_y, item.text))))
    }
}
```

24. If we were to run all these tests, they would fail. Now, let's move on to creating our logic. We will start with **`StringProvider`**:

```
class StringProvider(private val context: Context) {

    fun provideItemString(number: Int): String = ""

    fun provideYouClickedString(itemText: String) = ""
}
```

25. Now, we will need to test this. In order to do so, we will need **`mockito`**, and we will need the Mockito configuration for Kotlin defined in the *mockito-kotlin* section. After this, we can create **`StringProviderTest`**, which will look like this:

```
@RunWith(MockitoJUnitRunner::class)
class StringProviderTest {

    @InjectMocks
    lateinit var stringProvider: StringProvider
    @Mock
    lateinit var context: Context

    @Test
    fun provideItemString() {
        val number = 5
        val expected = "expected"
        whenever(context.getString(R.string.item_x,
          number)).thenReturn(expected)

        val result = stringProvider.provideItemString(number)

        assertEquals(expected, result)
    }

    @Test
    fun provideYouClickedString() {
        val itemText = "itemText"
        val expected = "expected"
        whenever(context.getString(R.string.you_clicked_y,
          itemText)).thenReturn(expected)

        val result = stringProvider.provideYouClickedString(itemText)

        assertEquals(expected, result)
    }
}
```

26. If we run the test, it will fail. This will allow us to correct our mistake and fix the implementation. When the test passes, we will end up with a **`StringProvider`** class that will look like this:

```
class StringProvider(private val context: Context) {

    fun provideItemString(number: Int): String
      = context.getString(R.string.item_x, number)

    fun provideYouClickedString(itemText: String)
      = context.getString(R.string.you_clicked_y, itemText)
}
```

27. Now, let's move to an empty **`ItemGenerator`** class, which will contain two methods, one to generate the items asynchronously, and the other to generate them synchronously. The reason for this approach is that we will need to extend this class in the integration test to get it to run fast:

```
open class ItemGenerator(
    private val timer: Timer,
    private val stringProvider: StringProvider,
    private val initialDelay: Long,
    private val countingIdlingResource: CountingIdlingResource
) {

    fun generateItemsAsync(itemCount: Int, callback:
      (List<Item>) -> Unit) {

    }

    open internal fun generateItems(itemCount: Int):
        List<Item> = listOf()

}
```

28. Next, there is the test for **`ItemGenerator`**. In order to generate data asynchronously with a callback, we can take advantage of Mockito's **`thenAnswer`** method. What we can do is call **`TimerTask`** every time it is scheduled on the timer. In the **`thenAnswer`** method, we can access the parameters in a method and invoke them. This is the approach we will use to handle an async call making it a sync call:

```
@RunWith(MockitoJUnitRunner::class)
class ItemGeneratorTest {

    private lateinit var itemGenerator: ItemGenerator
    @Mock
    lateinit var timer: Timer
    @Mock
    lateinit var stringProvider: StringProvider
    private val initialDelay = 5L
    @Mock
    lateinit var countingIdlingResource: CountingIdlingResource

    @Before
    fun setUp() {
        itemGenerator
          = ItemGenerator(timer, stringProvider, initialDelay,
            countingIdlingResource)
    }

    @Test
    fun generateItemsAsync() {
        val spy = spy(itemGenerator)
        val callback = mock<(List<Item>) -> Unit>()
        val itemCount = 10
        val items = listOf(Item("1"), Item("2"))
        doReturn(items).whenever(spy).generateItems(itemCount)
        whenever(timer.schedule(any(),
          eq(initialDelay))).thenAnswer {
            (it.arguments[0] as TimerTask).run()
        }

        spy.generateItemsAsync(itemCount, callback)

        verify(callback).invoke(items)
```

```
            verify(countingIdlingResource).increment()
            verify(countingIdlingResource).decrement()
        }

        @Test
        fun generateItems() {
            val itemCount = 10
            val expected = mutableListOf<Item>()
            for (i in 1..itemCount) {
                val itemText = "itemText$i"
                whenever(stringProvider.provideItemString(i))
                  .thenReturn(itemText)
                expected.add(Item(itemText))
            }

            val result = itemGenerator.generateItems(itemCount)

            assertEquals(expected, result)
        }
    }
```

29. We can then use the continuous test failure to update **`ItemGenerator`** until we get this:

```
    open class ItemGenerator(
        private val timer: Timer,
        private val stringProvider: StringProvider,
        private val initialDelay: Long,
        private val countingIdlingResource: CountingIdlingResource
    ) {

        fun generateItemsAsync(itemCount: Int, callback:
          (List<Item>) -> Unit) {
            countingIdlingResource.increment()
            timer.schedule(ItemGeneratorTask(itemCount, callback),
              initialDelay)
        }

        internal open fun generateItems(itemCount: Int): List<Item> {
            val result = mutableListOf<Item>()
            for (i in 1..itemCount) {
```

```
                result.add(Item(stringProvider.provideItemString(i)))
            }
            return result
        }


        inner class ItemGeneratorTask(
            private val itemCount: Int,
            private val callback: (List<Item>) -> Unit
        ) : TimerTask() {
            override fun run() {
                callback.invoke(generateItems(itemCount))
                countingIdlingResource.decrement()
            }

        }
    }
```

30. Now we are done with the unit tests. Let's move on to make the integration tests pass. We will start with **`Activity1Test`**. Here, we will need to connect the click listener to the button to open **`Activity2`**. We can update the **`onCreate`** method of **`Activity1`** to add the following:

```
 override fun onCreate(savedInstanceState: Bundle?) {
        ...
        findViewById<Button>(R.id.activity_1_button)
          .setOnClickListener {
            startActivity(
                Activity2.newIntent(
                    this,
                        findViewById<EditText>
                          (R.id.activity_1_edit_text)
                          .text.toString().toIntOrNull()
                    ?: 0
                )
            )
        }
        ...
    }
```

31. For **`Activity2`**, we will need to add **`ItemGenerator`** and an adapter to **`RecyclerView`** to render it. Let's start with the layout for the row in **`item.xml`**:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:minHeight="50dp"
    android:orientation="vertical">

    <TextView
        android:id="@+id/item_text_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center" />
</LinearLayout>
```

32. Next, let's create **`ItemAdapter`**, which will use the layout for each item:

```
class ItemAdapter(
    private val layoutInflater: LayoutInflater,
    private val onRowClickListener: (Item) -> Unit
) : RecyclerView.Adapter<ItemAdapter.ItemViewHolder>() {

    private val items = mutableListOf<Item>()

    fun addItems(items: List<Item>) {
        this.items.clear()
        this.items.addAll(items)
        notifyDataSetChanged()
    }

    ooverride fun onCreateViewHolder(parent: ViewGroup,
      viewType: Int): ItemViewHolder {
        return ItemViewHolder(layoutInflater.inflate
      (R.layout.item, parent, false))
    }

    override fun onBindViewHolder(holder: ItemViewHolder,
      position: Int) {
```

```
            holder.bind(items[position])
        }

        override fun getItemCount(): Int {
            return items.size
        }

        inner class ItemViewHolder(containerView: View) :
          RecyclerView.ViewHolder(containerView) {

            private val itemTextView: TextView
              = containerView.findViewById(R.id.item_text_view)

            init {
                containerView.setOnClickListener {
                    val position = adapterPosition
                    if (position > RecyclerView.NO_POSITION) {
                        onRowClickListener.invoke(items[position])
                    }
                }
            }

            fun bind(item: Item) {
                itemTextView.text = item.text
            }
        }
    }
```

33. Now, let's update the **`MyApplication`** class to create all the classes that will perform the logic:

```
        open class MyApplication : Application() {

        val countingIdlingResource = CountingIdlingResource("Timer
          resource")
        val timer = Timer()
        lateinit var stringProvider: StringProvider
        lateinit var itemGenerator: ItemGenerator

        override fun onCreate() {
            super.onCreate()
            stringProvider = StringProvider(this)
```

```
            itemGenerator = createItemGenerator()
        }

        protected open fun createItemGenerator(): ItemGenerator =
          ItemGenerator(timer, stringProvider, 1000,
    countingIdlingResource)
    }
```

34. Now, we need to update the **`onCreate`** method of **`Activity2`** to load the data in the list:

```
        private lateinit var adapter : ItemAdapter
        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_2)
            val recyclerView
              = findViewById<RecyclerView>(R.id.activity_2_recycler_view)
            recyclerView.layoutManager = LinearLayoutManager(this)
            adapter= ItemAdapter(LayoutInflater.from(this)) {
                startActivity(Activity3.newIntent(this, it))
            }
            recyclerView.adapter = adapter
            (application as MyApplication).itemGenerator
              .generateItemsAsync(intent.getIntExtra
                (EXTRA_ITEM_COUNT, 0)) {
                runOnUiThread {
                    adapter.addItems(it)
                }
            }
        }
```

35. In order to make the Robolectric test run fast, we can create a shadow implementation of **`ItemGenerator`** called **`InstantItemGenerator`**. We will create this file in the **`test`** folder. When the **`generateItemsAsync`** method is called, we will just generate the items instantly and invoke the callback:

```
    @Implements(ItemGenerator::class)
    class InstantItemGenerator {

        @RealObject
        lateinit var itemGenerator: ItemGenerator

        @Implementation
```

```
        fun generateItemsAsync(itemCount: Int, callback:
          (List<Item>) -> Unit) {
            callback.invoke(itemGenerator.generateItems(itemCount))
        }
    }
```

36. The **`@Implements`** annotation will tell Robolectric that this class is a shadow of the **`ItemGenerator`** class. This means that when Robolectric sees this **`Shadow`** class in its configuration, it will swap **`ItemGenerator`** instances with this shadow. **`@RealObject`** indicates that we will need the actual **`ItemGenerator`** instances here for the item generation. **`@Implementation`** will let Robolectric know which methods to invoke on the shadow. In this case, we will change the behavior of **`generateItemsAsync`** to generate the items instantly. We will need to modify **`Activity2Test`** in order to add the shadow to the configuration and **`LooperMode`** to the test method for threading issues:

```
@MediumTest
@Config(shadows = [InstantItemGenerator::class])
@RunWith(AndroidJUnit4::class)
class Activity2Test {
...

@LooperMode(LooperMode.Mode.PAUSED)
@Test
fun 'test click opens activity 3'() {
...
```

37. In order to make **`Activity3Test`** pass, we just need to update **`Activity3`** to set the text:

```
        verride fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_3)
            val text = intent.getParcelableExtra<Item>
              (EXTRA_ITEM)?.text.orEmpty()
            findViewById<TextView>(R.id.activity_3_text_view).text
              = (application as MyApplication).stringProvider
                .provideYouClickedString(text)
        }
```

38. Now, we can finally focus on the UI tests. Here, we just need to change the delay in the list generation to **0**. We can create **MyInstrumentedApp** in the **androidTest** package and create a custom runner:

```
class MyInstrumentedApplication : MyApplication() {

    override fun createItemGenerator(): ItemGenerator {
        return ItemGenerator(timer, stringProvider, 0,
          countingIdlingResource)
    }
}
```

39. Create the custom test runner in the **androidTest** folder. The runner will create an instance of **MyInstrumentedApplication** instead of **MyApplication**:

```
class MyTestRunner : AndroidJUnitRunner() {

    @Throws(Exception::class)
    override fun newApplication(
        cl: ClassLoader?,
        className: String?,
        context: Context?
    ): Application? {
        return super.newApplication(cl,
          MyInstrumentedApplication::class.java.name, context)
    }
}
```

40. Finally, add the Gradle configuration for the test runner to let Gradle know to execute our test runner instead of the default one:

```
android {
    ...
    defaultConfig {
        testInstrumentationRunner
          "com.android.testable.myapplication.MyTestRunner"
    }
    ...
}
```

41. If you run your UI test now, it should pass, indicating that we have completed the feature.

42. In order to run the local tests from **`Terminal`**, you can use **`gradlew`** (or **`gradlew.bat`** for Windows) and the **`test`** command (for example, **`./gradlew test`**). For the instrumented tests, you can use the **`./gradlew connectedAndroidTest`** command.

CHAPTER 10: ANDROID ARCHITECTURE COMPONENTS

ACTIVITY 10.01: SHOPPING NOTES APP

Solution:

Let's start with our Room integration:

1. The code for **`Entity`** is as follows:

```
@Entity(tableName = "notes")
data class Note(
    @PrimaryKey(autoGenerate = true) @ColumnInfo(name = "id")
      val id: Long = 0,
    @ColumnInfo(name = "text") val text: String
)
```

2. Let's create **`NoteDao`**, as follows:

```
@Dao
interface NoteDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertNote(note: Note)

    @Query("SELECT * FROM notes")
    fun loadNotes(): LiveData<List<Note>>

    @Query("SELECT count(*) FROM notes")
    fun loadNoteCount(): LiveData<Int>
}
```

Notice that for the queries, we changed the return types to **`LiveData`**. This will solve two issues. First, the queries will be executed on a separate thread. Second, if the data changes, **`LiveData`** will notify the observers of the most recent changes.

3. Now, add the **`Note`** entity to **`NotesDatabase`**, as follows:

```
@Database(
    entities = [Note::class],
    version = 1
)
abstract class NotesDatabase : RoomDatabase() {

    abstract fun noteDao(): NoteDao
}
```

This is a simple database. Since we are not using a dependency injection framework, we will need to play around with the **`Application`** class to keep one instance of our database for the entire app. We will do this by extending the **`Application`** class and linking it in the manifest.

```
class NotesApplication : Application() {

    lateinit var notesDatabase: NotesDatabase

    override fun onCreate() {
        super.onCreate()
        notesDatabase = Room.databaseBuilder(applicationContext,
          NotesDatabase::class.java, "notes-db").build()
    }
}
```

Here, we have defined a new **`Application`** class that will have the **`onCreate`** method called when the app is opened. This will ensure that we have one instance of the database that can be used across different components.

4. In the **`AndroidManifest.xml`** file, we need to define the **`android:name`** attribute in the **`<application>`** tag to ensure that the code in our class can be executed:

```
    <application
        android:name=".NotesApplication"
```

Make sure to use the appropriate package if one is declared.

5. Now, let's define a repository. The **`Repository`** pattern is useful in situations where you have one or more sources of data (server, room, or memory) that can be combined, modified, and processed. This will help us centralize our access to the data and decouple the application code from the data sources. In our case, the only data source we have is Room, so our repository will act as a wrapper over **`NoteDao`**, which will access the data on a separate thread. The reason for multi-threading is because inserting the data would require a separate thread.
6. Let's start with a template of our repository in the form of an interface:

```
interface NoteRepository {

    fun insertNote(note: Note)

    fun getAllNotes(): LiveData<List<Note>>

    fun getNoteCount(): LiveData<Int>
}
```

7. Now, let's add the implementation of our repository:

```
class NoteRepositoryImpl(
    private val executor: Executor,
    private val noteDao: NoteDao
) : NoteRepository {

    override fun insertNote(note: Note) {
        executor.execute {
            noteDao.insertNote(note)
        }
    }

    override fun getAllNotes(): LiveData<List<Note>> {
        return noteDao.loadNotes()
    }

    override fun getNoteCount(): LiveData<Int> {
        return noteDao.loadNoteCount()
    }
}
```

The data retrieval process is automatically handled for us by Room. However, for inserting the data, we needed a separate thread, so in this example, we went with a Java executor. This will perform every insertion on a separate thread. This **`Executor`** will be injected through the constructor to give us the opportunity to unit test this repository.

8. Now, let's modify our **`NotesApplication`** class in order to provide one instance of the repository that will be used across the application:

```
class NotesApplication : Application() {

    lateinit var notesDatabase: NotesDatabase
    lateinit var noteRepository: NoteRepository

    override fun onCreate() {
        super.onCreate()
        notesDatabase =
            Room.databaseBuilder(applicationContext,
              NotesDatabase::class.java, "notes-db")
                .build()
        noteRepository = NoteRepositoryImpl(
            Executors.newSingleThreadExecutor(),
            notesDatabase.noteDao()
        )
    }
}
```

9. Now, let's unit test our repository. For this, we will need the Mockito library. This library will allow us to mock the instances of **`Executor`** and **`Dao`**. To add Mockito, go to **`app/build.gradle`** and add the following code:

```
testImplementation 'org.mockito:mockito-core:2.23.0'
```

To test the Kotlin code with Mockito, you will need to add a few configurations. In the **`test`** folder, create a new folder called **`resources`**, and inside that folder, create a new folder called **`mockito-extensions`**. Inside this folder, create a file named **`org.mockito.plugins.MockMaker`** and inside that file, add **`mock-maker-inline`** as a line of code. This configuration will allow you to mock Kotlin classes (which, by default, are final – Mockito is incapable of mocking final classes).

10. Let's see what our unit test will look like:

```
@RunWith(MockitoJUnitRunner::class)
class NoteRepositoryImplTest {

    @InjectMocks
    lateinit var noteRepository: NoteRepositoryImpl
    @Mock
    lateinit var executor: Executor
    @Mock
    lateinit var noteDao: NoteDao

    @Test
    fun insertNote() {
        val note = Note(10, "text")
        doAnswer {
            (it.arguments[0] as Runnable).run()
        }.'when'(executor).execute(ArgumentMatchers.any())

        noteRepository.insertNote(note)

        verify(noteDao).insertNote(note)

    }

    @Test
    fun getAllNotes() {
        val notes = mock(LiveData::class.java)
        'when'(noteDao.loadNotes()).thenReturn(notes as
          LiveData<List<Note>>)

        val result = noteRepository.getAllNotes()

        assertEquals(notes, result)
    }

    @Test
    fun getNoteCount() {
        val count = mock(LiveData::class.java)
```

```
            'when'(noteDao.loadNoteCount()).thenReturn(count as
              LiveData<Int>)

            val result = noteRepository.getNoteCount()

            assertEquals(count, result)
        }
    }
```

With Mockito, we were able to inject mocks, which are not real instances of our **`Executor`** and **`NoteDao`**. Then, before each test, we instructed the mocks on how to behave. For testing the insert, we told our mock **`Executor`** to execute any task on the spot, thus avoiding a threading issue during testing. When testing the loading phase, we instructed the mock **`NoteDao`** to return a mock **`LiveData`**. Then, we made sure that the result returned by our real repository (the target of the unit test should never be mocked) is the result returned by **`NoteDao`**.

Now, let's define our **`ViewModels`**. Here, we might have a problem. We don't instantiate **`ViewModels`** ourselves, which means it's going to be a little harder to pass our instance of the repository into **`ViewModel`**. Luckily, we have two alternatives in our case. The first is to define the repository in the constructor and use a **`Factory`** to pass the instance. The second is to use a specialized subclass of the **`ViewModel`** class, called **`AndroidViewModel`**, that contains the **`Application`** object as a parameter. Since we defined the instances in the **`Application`** class, we can use that constructor to access the repository instance. Since we will have two **`ViewModels`**, let's use both approaches.

11. Let's start with **`NoteListViewModel`**:

```
class NoteListViewModel(private val noteRepository:
  NoteRepository) : ViewModel() {

    fun getNoteListLiveData(): LiveData<List<Note>> =
      noteRepository.getAllNotes()
}
```

12. **`NoteListViewModel`** has the following test:

```
@RunWith(MockitoJUnitRunner::class)
class NoteListViewModelTest {

    @InjectMocks
    lateinit var noteListViewModel: NoteListViewModel
    @Mock
```

```
    lateinit var noteRepository: NoteRepository

    @Test
    fun getNoteListLiveData() {
        val notes = Mockito.mock(LiveData::class.java)
        Mockito.'when'(noteRepository.getAllNotes())
          .thenReturn(notes as LiveData<List<Note>>)

        val result = noteListViewModel.getNoteListLiveData()

        assertEquals(notes, result)
    }
}
```

13. Now, let's define **`CountNotesViewModel`**:

```
class CountNotesViewModel(application: Application) :
  AndroidViewModel(application) {

    private val noteRepository: NoteRepository = (application as
      NotesApplication).noteRepository

    fun insertNote(text: String) {
        noteRepository.insertNote(Note(0, text))
    }

    fun getNoteCountLiveData(): LiveData<Int> =
    noteRepository.getNoteCount()

}
```

14. The test associated with the preceding **`ViewModel`** class is as follows:

```
@RunWith(MockitoJUnitRunner::class)
class CountNotesViewModelTest {

    private lateinit var countNotesViewModel: CountNotesViewModel
    @Mock
    lateinit var application: NotesApplication
    @Mock
    lateinit var noteRepository: NoteRepository

    @Before
    fun setUp() {
```

```
            Mockito.'when'(application.noteRepository)
            .thenReturn(noteRepository)
            countNotesViewModel = CountNotesViewModel(application)
        }

        @Test
        fun insertNote() {
            val text = "text"
            countNotesViewModel.insertNote(text)
            Mockito.verify(noteRepository).insertNote(Note(0, text))
        }

        @Test
        fun getNoteCountLiveData() {
            val notes = Mockito.mock(LiveData::class.java)
            Mockito.'when'(noteRepository.getNoteCount())
              .thenReturn(notes as LiveData<Int>)

            val result = countNotesViewModel.getNoteCountLiveData()

            assertEquals(notes, result)
        }
    }
```

15. Now, let's build our UI. To do this, first, we need the **`RecyclerView`** library. Here, you need to add the following to **`app/build.gradle`**:

```
    implementation 'androidx.recyclerview:recyclerview:1.1.0'
```

16. Let's start with **`CountNotesFragment`**:

```
    class CountNotesFragment : Fragment() {

        override fun onCreateView(
            inflater: LayoutInflater,
            container: ViewGroup?,
            savedInstanceState: Bundle?
        ): View? {
            return inflater.inflate(R.layout.fragment_count_notes,
              container, false)
        }

        override fun onViewCreated(view: View, savedInstanceState:
          Bundle?) {
```

```
        super.onViewCreated(view, savedInstanceState)
        val viewModel =
            ViewModelProvider(requireActivity())
              .get(CountNotesViewModel::class.java)
        viewModel.getNoteCountLiveData()
          .observe(viewLifecycleOwner, Observer {
        view.findViewById<TextView>
          (R.id.fragment_count_notes_text_view).text =
            getString(R.string.total, it)
        })
        view.findViewById<Button>
          (R.id.fragment_count_notes_button).setOnClickListener {
            viewModel.insertNote(view.findViewById<EditText>
              (R.id.fragment_count_edit_text).text.toString())
        }
    }


}
```

17. We also need to add the associated **`fragment_count_notes.xml`** layout:

```
<?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:gravity="center"
    android:orientation="vertical">


    <TextView
        android:id="@+id/fragment_count_notes_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />


    <EditText
        android:id="@+id/fragment_count_edit_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginStart="20dp"
        android:layout_marginEnd="20dp" />


    <Button
        android:id="@+id/fragment_count_notes_button"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:text="@string/add_note" />

</LinearLayout>
```

18. Next, let's define an adapter for the list of **`Notes`** called **`NoteListAdapter`**:

```
class NoteListAdapter(private val inflater: LayoutInflater) :
    RecyclerView.Adapter<NoteListAdapter.NoteViewHolder>() {

    private val noteList = mutableListOf<Note>()

    override fun onCreateViewHolder(parent: ViewGroup, viewType:
      Int): NoteViewHolder {
        return NoteViewHolder(inflater.inflate
          (R.layout.view_note_item, parent, false))
    }

    override fun getItemCount() = noteList.size

    override fun onBindViewHolder(holder: NoteViewHolder,
      position: Int) {
        holder.bind(noteList[position])
    }

    fun replaceItems(notes: List<Note>) {
        noteList.clear()
        noteList.addAll(notes)
        notifyDataSetChanged()
    }

    inner class NoteViewHolder(containerView: View) :
        RecyclerView.ViewHolder(containerView) {

            private val noteTextView: TextView =
                containerView.findViewById<TextView>
                  (R.id.view_note_list_text_view)

            fun bind(note: Note) {
                noteTextView.text = note.text
        }
    }
}
```

19. We'll also add an associated layout file for the rows called **`view_note_item.xml`**:

```
<?xml version="1.0" encoding="utf-8"?> <TextView
  xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/view_note_list_text_view"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="10dp" />
```

20. Now, let's write **`NoteListFragment`**:

```
class NoteListFragment : Fragment() {


    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_note_list,
          container, false)
    }


    override fun onViewCreated(view: View, savedInstanceState:
      Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        val recyclerView = view.findViewById<RecyclerView>
          (R.id.fragment_note_list_recycler_view)
        recyclerView.layoutManager = LinearLayoutManager(context)
        val adapter =
          NoteListAdapter(LayoutInflater.from(context))
        recyclerView.adapter = adapter
        val viewModel =
            ViewModelProvider(requireActivity(), object :
              ViewModelProvider.Factory {
                override fun <T : ViewModel?> create(modelClass:
                  Class<T>): T {
                    return NoteListViewModel((requireActivity()
                      .application as
                        NotesApplication).noteRepository) as T
                }


            }).get(NoteListViewModel::class.java)
        viewModel.getNoteListLiveData()
          .observe(viewLifecycleOwner, Observer {
            adapter.replaceItems(it)
```

```
            })
        }
    }
```

This is where we define the custom factory for our **`ViewModel`**. This will allow us to inject the instance of **`NoteRepository`** through the constructor. This technique may come in useful when using dependency injection frameworks.

21. Now, let's define the associated layout file, called **`fragment_note_list.xml`**:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.recyclerview.widget.RecyclerView
  xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/fragment_note_list_recycler_view"/>
```

22. Finally, let's define the activity:

```
class NotesActivity : AppCompatActivity() {


    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_note)
    }
}
```

23. We need to add the following portrait layout to **`layout/activity_note.xml`**:

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


    <fragment
        android:id="@+id/activity_note_count_fragment"
        class="com.android.testable.notesapplication
          .CountNotesFragment"
        android:layout_width="match_parent"
```

```
            android:layout_height="0dp"
            android:layout_weight="1" />

        <fragment
            android:id="@+id/activity_note_list_fragment"
            class="com.android.testable.notesapplication
              .NoteListFragment"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1" />

    </LinearLayout>
```

24. We also need to add the landscape file; that is, **`layout-land/activity_note.xml`**:

```
    <?xml version="1.0" encoding="utf-8"?>
    <LinearLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:baselineAligned="false"
        android:orientation="horizontal"
        tools:context=".NotesActivity">

        <fragment
            android:id="@+id/activity_note_count_fragment"
            class="com.android.testable.notesapplication
              .CountNotesFragment"
            android:layout_width="0dp"
            android:layout_height="match_parent"
            android:layout_weight="1" />

        <fragment
            android:id="@+id/activity_note_list_fragment"
            class="com.android.testable.notesapplication
              .NoteListFragment"
            android:layout_width="0dp"
            android:layout_height="match_parent"
            android:layout_weight="1" />

    </LinearLayout>
```

25. Finally, let's make sure we have the following strings in **`strings.xml`**:

```
    <string name="total">Total %d</string>

        <string name="add_note">Add Note</string>
```

26. Now, run the application:

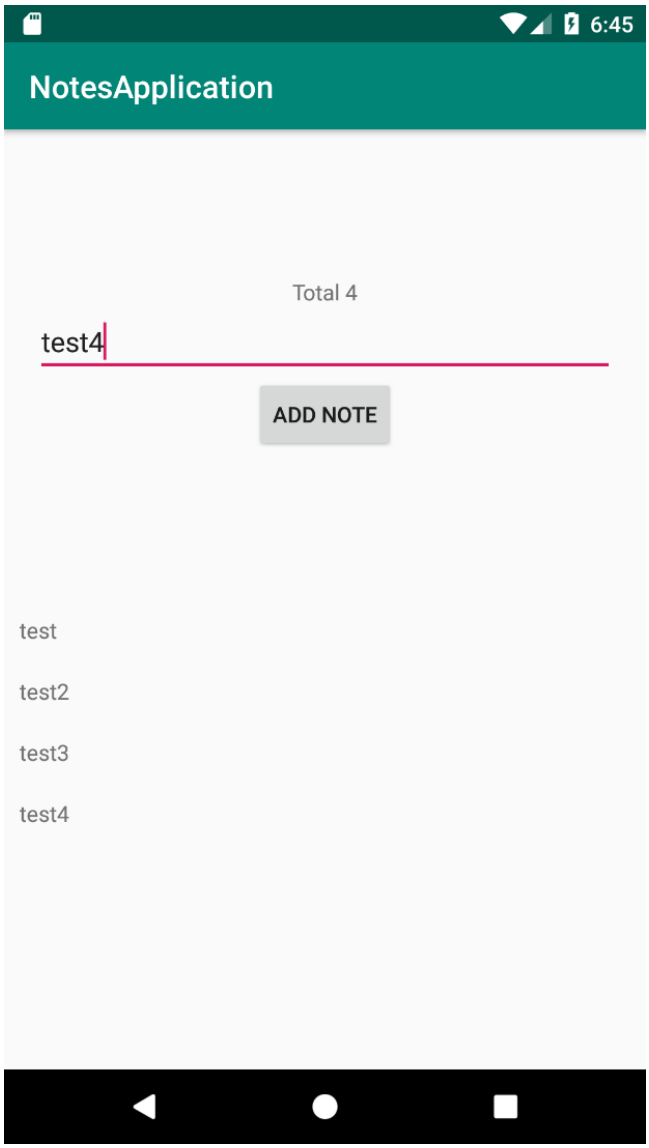

Figure 10.13: The life cycle of an activity compared to the ViewModel life cycle

Feel free to investigate the application and see if you can address any issues you may find. Here are a few to start with:

- If no text has been written when the button is clicked, a note shouldn't be saved.
- Add a test to **`NoteDatabase`**.
- Add a new field to the **`Note`** entity that will allow users to check and uncheck their notes.

CHAPTER 11: PERSISTING DATA

ACTIVITY 11.01: DOG DOWNLOADER

Solution:

1. Start by adding the following libraries to **app/build.gradle**:

```
    implementation 'commons-io:commons-io:2.6'
    implementation 'androidx.preference:preference:1.1.1'
    def lifecycle_version = "2.2.0"
    implementation "androidx.lifecycle:lifecycle-
      extensions:$lifecycle_version"
    implementation 'com.squareup.retrofit2:retrofit:2.6.2'
    implementation 'com.squareup.retrofit2:converter-gson:2.6.2'
    implementation 'com.google.code.gson:gson:2.8.6'
    def room_version = "2.2.5"
    implementation "androidx.room:room-runtime:$room_version"
    kapt "androidx.room:room-compiler:$room_version"
```

2. Create the **api** package.
3. Create a **Dog** data class, which will map the JSON data, containing the status and the list of URLs:

```
data class Dog(
    @SerializedName("status") val status: String,
    @SerializedName("message") val urls: List<String>
)
```

4. Now, create a **Retrofit** interface, which will define how we load the list of **Dogs** and how the download will be executed through a dynamic URL:

```
interface DownloadService {

    @GET("breed/hound/images/random/{number}")
    fun getDogs(@Path("number") number: Int): Call<Dog>

    @GET
    fun downloadFile(@Url fileUrl: String): Call<ResponseBody>
}
```

The **`@Path`** annotation allows us to dynamically set certain parts of the path and the **`@Url`** allows us to place a dynamic URL in the **`download`** function. The method will return a **`ResponseBody`** object, which will contain methods to allow us to access the bytes (through **`InputStream`**) of the file.

5. Add the **`INTERNET`** permission to the **`AndroidManifest.xml`** file:

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

6. Create the **`storage`** package and, inside this, create the **`room`** package.

7. Create a **`DogEntity`** class, which will contain an ID and the URL for the dog photo:

```
@Entity(tableName = "dogs")
data class DogEntity(
    @PrimaryKey(autoGenerate = true) @ColumnInfo(name = "id")
      val id: Long,
    @ColumnInfo(name = "url") val url: String
)
```

8. Create a **`DogDao`** interface, which will contain the method to insert a list of **`Dogs`**, query the existing dogs, and delete all the dogs in the table:

```
@Dao
interface DogDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertDogs(dogs: List<DogEntity>)

    @Query("SELECT * FROM dogs")
    fun loadDogs(): LiveData<List<DogEntity>>

    @Query("DELETE FROM dogs")
    fun deleteAll()
}
```

9. Create the **`DogDatabase`** class, which will connect the entity and the **`Dao`**:

```
@Database(
    entities = [DogEntity::class],
    version = 1
)
abstract class DogDatabase : RoomDatabase() {

    abstract fun dogDao(): DogDao
}
```

10. Create the **`xml`** resource directory inside the **`res`** folder.
11. Create a **`provider_paths.xml`** file inside the **`xml`** directory that will point to the external media folder. In this example, we will save the files directly in the root folder:

```
<?xml version="1.0" encoding="utf-8"?>
<paths>
    <external-cache-path
        name="my-media"
        path="." />
</paths>
```

12. Inside the **`storage`** package, create the **`filesystem`** package.
13. Create the **`FileToUriMapper`** class, which will convert a file into a **`Uri`** to allow us to test the other classes better:

```
class FileToUriMapper {

    fun getUriForFile(context: Context, file: File): Uri {
        return FileProvider.getUriForFile(
            context,
            "com.android.testable.remote_media_provider",
            file
        )
    }
}
```

14. Create a **`ProviderFileHandler`** class, which will be responsible for writing inside a file that will belong to **`FileProvider`**:

```
class ProviderFileHandler(
    private val context: Context,
    private val fileToUriMapper: FileToUriMapper
) {

    fun writeStream(name: String, inputStream: InputStream) {
        val fileToSave = File(context.externalCacheDir, name)
        val outputStream =
          context.contentResolver.openOutputStream(
            fileToUriMapper.getUriForFile(context, fileToSave),
            "rw"
        )
        IOUtils.copy(inputStream, outputStream)
    }
}
```

15. Make sure you have the provider in the **`AndroidManifest.xml`** file:

```
        <provider
            android:name="androidx.core.content.FileProvider"
            android:authorities="com.android.testable
              .remote_media_provider"
            android:exported="false"
            android:grantUriPermissions="true">
            <meta-data
                android:name="android.support
                  .FILE_PROVIDER_PATHS"
                android:resource="@xml/provider_paths" />
        </provider>
```

Make sure to set the same authority here as you did in **`FileToUriMapper`**.

16. Create a **`preferences.xml`** file in **`res/values`** where we will store our key for **`SharedPreferences`** and define our key:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="preference_key_nr_results"
      translatable="false">preference_key_nr_results</string>
</resources>
```

17. Create a package named **`preference`** in the **`storage`** package.
18. In the **`preference`** package, create **`DownloadPreferencesWrapper`**, which will be responsible for retrieving the number of results we want to display on the screen. In case there is no value saved, we will default to 10 results:

```
const val DEFAULT_NO_OF_RESULTS = 10

class DownloadPreferencesWrapper(
    private val context: Context,
    private val sharedPreferences: SharedPreferences
) {

    fun getNumberOfResults(): Int {
        return sharedPreferences.getString(
            context.getString
              (R.string.preference_key_nr_results),
            DEFAULT_NO_OF_RESULTS.toString()
        ).orEmpty().toIntOrNull() ?: DEFAULT_NO_OF_RESULTS
    }
}
```

19. Now, move on to the **`Repository`** aspect and create a new package called **`repository`**.
20. Create a class named **`Result`**, which will have three outputs: **`Loading`**, **`Success`**, and **`Error`**. We can achieve this through the Kotlin **`sealed`** class feature:

```
sealed class Result<T> {

    class Loading<T> : Result<T>()
    data class Success<T>(val data: T) : Result<T>()
    class Error<T> : Result<T>()
}
```

21. Define a model that will be used by our UI layer:

```
data class DogUi(val url: String)
```

22. Define a mapper class that will convert one type of model into another:

```
class DogMapper {


    fun mapServiceToEntity(dog: Dog): List<DogEntity> =
      dog.urls.map {
        DogEntity(0, it)
      }


    fun mapEntityToUi(dogEntity: DogEntity): DogUi =
      DogUi(dogEntity.url)
}
```

23. Next, define our repository interface and name it **`DownloadRepository`**:

```
interface DownloadRepository {


    fun loadDogList(): LiveData<Result<List<DogUi>>>


    fun downloadFile(url: String): LiveData<Result<Unit>>
}
```

24. Provide the implementation for the repository. The implementation for retrieving the list of URLs will set the **`Loading`** state first, and then it will monitor any changes in the database and start the request. When the request finishes, it inserts the data in the **`Database`**, which should then provide notification regarding the changes to the data model:

`DownloadRepositoryImpl.kt`

```
26    override fun loadDogList(): LiveData<Result<List<DogUi>>> {
27        val result = MediatorLiveData<Result<List<DogUi>>>()
28        result.postValue(Result.Loading())
29        result.addSource(dogDao.loadDogs()) { dogEntities ->
30            Result.Success(dogEntities.map {
              dogMapper.mapEntityToUi(it) })
31        }
32        downloadService.getDogs(downloadPreferencesWrapper
            .getNumberOfResults())
33            .enqueue(object : Callback<Dog> {
34                override fun onResponse(call: Call<Dog>,
                    response: Response<Dog>) {
35                    if (response.isSuccessful) {
36                        executor.execute {
37                            dogDao.deleteAll()
38                            dogDao.insertDogs(dogMapper
                                .mapServiceToEntity
                                (response.body()!!))
39                        }
40                    } else {
41                        result.postValue(Result.Error())
42                    }
```

```
43                }
44
45                override fun onFailure(call: Call<Dog>,
                    t: Throwable) {
46                    result.postValue(Result.Error())
47                }
48
49            })
50        return result
51    }
```

The complete code for this step can be found at http://packt.live/2LRdtMz.

The implementation for downloading a file will set the **`Loading`** state when the download is started. Then, if the connection to the server isn't established, or the server replies with an error or the download cannot be performed, it will set the state to **`Error`**. If the download is completed successfully, it will show a success message:

DownloadRepositoryImpl.kt

```
            override fun downloadFile(url: String):
              LiveData<Result<Unit>> {
                val result = MutableLiveData<Result<Unit>>()
                result.postValue(Result.Loading())
                downloadService.downloadFile(url)
                .enqueue(object : Callback<ResponseBody> {


                    override fun onFailure(call: Call<ResponseBody>,
                      t: Throwable) {
                        result.postValue(Result.Error())
                        t.printStackTrace()
                    }



                    override fun onResponse(
                        call: Call<ResponseBody>,
                        response: Response<ResponseBody>
                    ) {
                    if (response.isSuccessful) {
                        executor.execute {
                            try {
                                response.body()?.let {
                                    val name = url.substring
                                      (url.lastIndexOf("/") + 1)
                                    providerFileHandler.writeStream(
```

```
                                        name,
                                        response.body()
                                          !!.byteStream()
                                    )
                                    result.postValue
                                      (Result.Success(Unit))
                                }
                            } catch (e: Exception) {
                            e.printStackTrace()
                            result.postValue(Result.Error())
                        }


                    }
                } else {
                    result.postValue(Result.Error())
                }
            }


        })
        return result
    }
```

The complete code for this step can be found at http://packt.live/39RFbAF.

25. Now, create the **`Application`** class, which will initialize all of the required instances and provide access to the **`Repository`** instance to the rest of the app. Make sure to add the **`android:name`** attribute to **`AndroidManifest`** and to the **`application`** tag:

```
class RemoteProviderApplication : Application() {

    lateinit var downloadRepository: DownloadRepository
    lateinit var preferencesWrapper: DownloadPreferencesWrapper

    override fun onCreate() {
        super.onCreate()

        val retrofit = Retrofit.Builder()
            .baseUrl("https://dog.ceo/api/")
            .addConverterFactory(GsonConverterFactory.create())
            .build()
```

```
            val downloadService =
              retrofit.create<DownloadService>
                (DownloadService::class.java)
            val database =
                Room.databaseBuilder(applicationContext,
                  DogDatabase::class.java, "dog-db")
                    .build()
            preferencesWrapper = DownloadPreferencesWrapper(
                this,
                PreferenceManager.getDefaultSharedPreferences(this)
            )
            downloadRepository = DownloadRepositoryImpl(
                preferencesWrapper,
                ProviderFileHandler(
                    this,
                    FileToUriMapper()
                ),
                downloadService,
                database.dogDao(),
                DogMapper(),
                Executors.newSingleThreadExecutor()
            )
        }
    }
```

26. Move on to **`MainViewModel`**, which will have a reference to **`Repository`** and will use **`MediatorLiveData`** to connect to the LiveData repository and that will allow the UI to observe only once, but to make multiple calls to retrieve the data from the repository:

```
    class MainViewModel(private val downloadRepository:
    DownloadRepository) : ViewModel() {


        private val dogsLiveData:
          MediatorLiveData<Result<List<DogUi>>> by lazy {
            MediatorLiveData<Result<List<DogUi>>>()
        }


        private val downloadResult: MediatorLiveData<Result<Unit>>
          by lazy {
            MediatorLiveData<Result<Unit>>()
        }


        fun getDogs() {
```

```
            dogsLiveData.addSource(downloadRepository.loadDogList())
              {
                dogsLiveData.postValue(it)
            }
        }

        fun getDogsLiveData(): LiveData<Result<List<DogUi>>> {
            return dogsLiveData
        }

        fun downloadFile(url: String) {
            downloadResult.addSource(downloadRepository
              .downloadFile(url)) {
                downloadResult.postValue(it)
            }
        }

        fun getDownloadLiveData(): LiveData<Result<Unit>> =
          downloadResult

    }
```

27. Now, build the UI. Create the **`activity_main.xml`** file in the **`layout`** folder, which will contain **`RecyclerView`** and **`ProgressBar`**, which is indefinite and can be used to show the status of the download request:

`activity_main.xml`

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/activity_main_recycler_view"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<ProgressBar
    android:id="@+id/activity_main_progress_bar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:elevation="5dp"
    android:indeterminate="true"
    android:visibility="gone"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

The complete code for this step can be found at http://packt.live/3c4nXmF.

28. Create the layout for the rows, which will be one **`TextView`**, and display the URL:

`view_dog_item.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com
  /apk/res/android"
    android:id="@+id/view_dog_item_url_text_view"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="10dp" />
```

29. Create an adapter, which will handle the display for every row and will have a reference to a lambda that will be invoked when a row is clicked:

```
class MainAdapter(
    private val layoutInflater: LayoutInflater,
    private val onRowClickListener: (DogUi) -> Unit
) : RecyclerView.Adapter<MainAdapter.DogViewHolder>() {

    private val dogs = mutableListOf<DogUi>()

    override fun onCreateViewHolder(parent: ViewGroup,
      viewType: Int): DogViewHolder =
        DogViewHolder(layoutInflater.inflate(R.layout.view_dog_item,
          parent, false))

    override fun getItemCount(): Int = dogs.size

    override fun onBindViewHolder(holder: DogViewHolder,
      position: Int) {
        holder.bind(dogs[position])
    }

    fun  updateDogs(dogs: List<DogUi>) {
        this.dogs.clear()
        this.dogs.addAll(dogs)
        this.notifyDataSetChanged()
    }

    inner class DogViewHolder(containerView: View) :
        RecyclerView.ViewHolder(containerView) {
```

```
            private val urlTextView: TextView =
                containerView.findViewById<TextView>
                  (R.id.view_dog_item_url_text_view)

                init {
                    containerView.setOnClickListener {
                    val position = adapterPosition
                    if (position > RecyclerView.NO_POSITION) {
                        onRowClickListener.invoke(dogs[position])
                    }
                }
            }

            fun bind(dog: DogUi) {
                urlTextView.text = dog.url
            }
        }
    }
```

30. Finally, add **`ViewModel`** and **`Adapter`** to **`MainActivity`** and implement the **`download`** function:

`MainActivity.kt`

```
        val downloadRepository = (application as
          RemoteProviderApplication).downloadRepository
            mainViewModel = ViewModelProvider(this, object :
              ViewModelProvider.Factory {
                override fun <T : ViewModel?> create(modelClass:
                  Class<T>): T {
                    return MainViewModel(downloadRepository) as T
                }
            }).get(MainViewModel::class.java)

            val progressBar = findViewById<ProgressBar>
              (R.id.activity_main_progress_bar)
            mainViewModel.getDownloadLiveData()
            .observe(this, Observer { result ->
            when (result) {
                is Result.Loading -> {
                    progressBar.visibility = View.VISIBLE
                }
                is Result.Success -> {
```

```
                    progressBar.visibility = View.GONE
                    Toast.makeText(this, getString(R.string.success),
                      Toast.LENGTH_LONG)
                    .show()
                }
                is Result.Error -> {
                    progressBar.visibility = View.GONE
                    Toast.makeText(this, getString(R.string.error),
                      Toast.LENGTH_LONG)
                    .show()
                }
            }
        })
```

The complete code for this step can be found at http://packt.live/3qEEjq1.

If you run the code, you will see the following output:

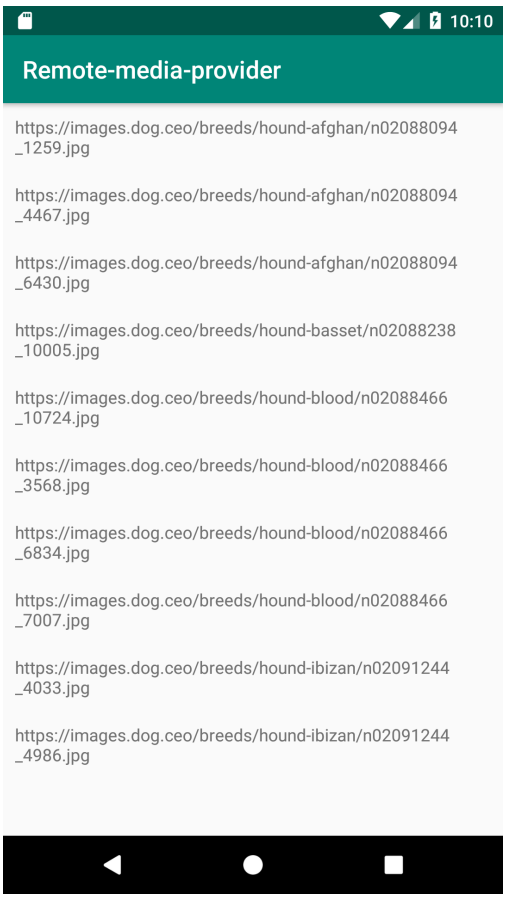

Figure 11.12: Output of the activity's main screen

31. Clicking the items will trigger the download for each individual item. You can view the files using **`Device File Explorer`**:

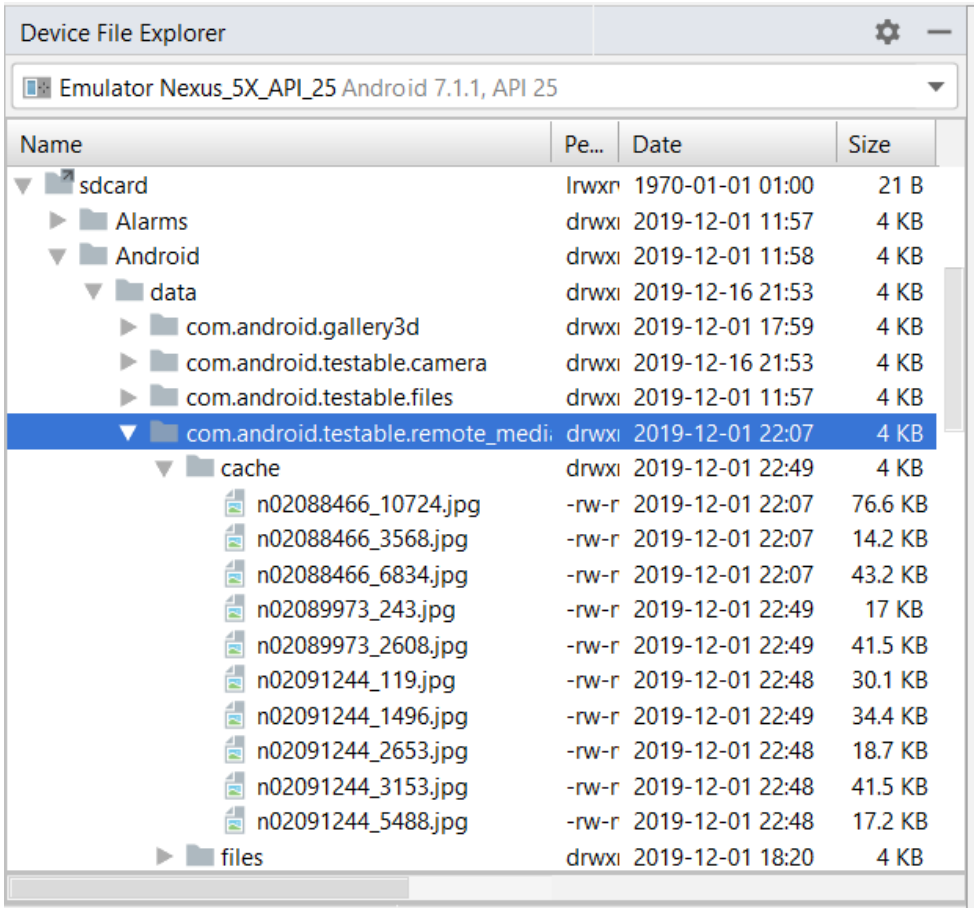

Figure 11.13: Viewing downloaded files through Device File Explorer

32. Now, define the menu for the main activity, but first we need a **`Settings`** icon. We can extract that by right-clicking on the **`res`** folder and selecting **`New`** | **`Vector Asset`**. Select a Settings icon from that list and save it as **`ic_settings`**.

33. Now, we can create a **`menu`** folder in the **`res`** folder and inside that we create the **`menu_main.xml`** file with the following specifications:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com
  /apk/res/android">
    <item
```

```
        android:id="@+id/menu_item_settings"
        android:icon="@drawable/ic_settings"
        android:title="@string/settings" />
</menu>
```

34. Define the **`preference_settings.xml`** file inside **`res/xml`**:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:app="http://schemas.android.com
  /apk/res-auto">

    <EditTextPreference
        app:key="@string/preference_key_nr_results"
        app:title="@string/number_of_items" />

</PreferenceScreen>
```

35. Also, define **`SettingsFragment`**, which will display just the one preference and, as a summary, it will display the existing value from **`SharedPreferences`**:

```
class SettingsFragment : PreferenceFragmentCompat() {
    override fun onCreatePreferences(savedInstanceState: Bundle?,
      rootKey: String?) {
        setPreferencesFromResource(R.xml.preferences_settings,
          rootKey)
        val resultsPreference =
            findPreference<EditTextPreference>
              (getString(R.string.preference_key_nr_results))
        val preferencesWrapper =
            (requireActivity().application as
               RemoteProviderApplication).preferencesWrapper
        resultsPreference?.summary =
          preferencesWrapper.getNumberOfResults().toString()

        resultsPreference?.onPreferenceChangeListener =
            Preference.OnPreferenceChangeListener { _, newValue
              ->
                resultsPreference?.summary = newValue?.toString()
                true
            }
    }
}
```

36. Insert the fragment in a new activity called **`SettingsActivity`**, which will have **`activity_settings.xml`** as the layout:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com
  /apk/res/android"
    android:id="@+id/activity_settings_fragment_container"
    class="com.android.testable.remote_media_provider
      .SettingsFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

37. The code inside **`SettingsActivity`** will be as follows:

```
class SettingsActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_settings)
    }
}
```

38. Add the activity to the **`AndroidManifest.xml`** file:

```
<activity android:name=".SettingsActivity" />
```

39. Finally, start it from **`MainActivity`** when the **`Settings`** option is selected:

```
    override fun onCreateOptionsMenu(menu: Menu?): Boolean {
        menuInflater.inflate(R.menu.menu_main, menu)
        return true
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        return when (item.itemId) {
            R.id.menu_item_settings -> {
                startActivity(Intent(this,
                  SettingsActivity::class.java))
                true
            }
            else -> {
                super.onOptionsItemSelected(item)
```

```
                }
            }

        }
```

Running this code and selecting the **`Settings`** option will give us the following screen. If you go and set a new numeric value, when you return to **`MainActivity`**, the list will refresh itself with the new number of items:

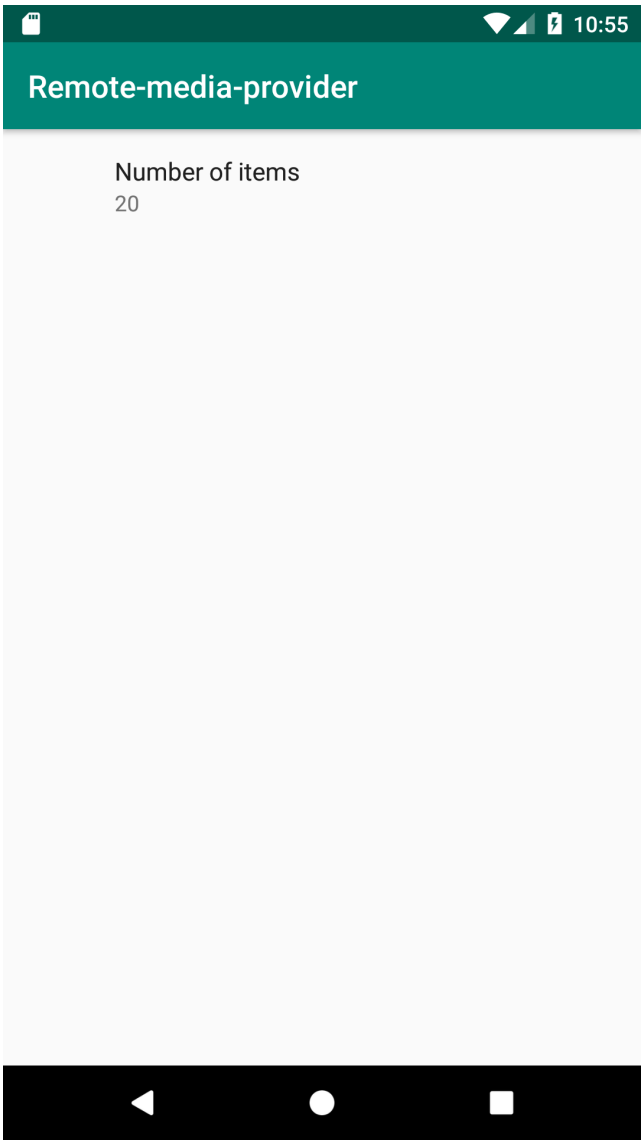

Figure 11.14: The output of SettingsActivity

CHAPTER 12: DEPENDENCY INJECTION WITH DAGGER AND KOIN

ACTIVITY 12.01: INJECTED REPOSITORIES

Solution:

1. Start with the **`app/build.gradle`** dependency configuration:

```
    implementation
      "androidx.constraintlayout:constraintlayout:2.0.4"
    implementation 'androidx.recyclerview:recyclerview:1.1.0'
    def lifecycle_version = "2.2.0"
    implementation "androidx.lifecycle:lifecycle-
      extensions:$lifecycle_version"
    implementation 'com.squareup.retrofit2:retrofit:2.6.2'
    implementation 'com.squareup.retrofit2:converter-gson:2.6.2'
    implementation 'com.google.code.gson:gson:2.8.6'
    implementation 'com.google.dagger:dagger:2.29.1'
    kapt 'com.google.dagger:dagger-compiler:2.29.1'
    testImplementation 'junit:junit:4.12'
    testImplementation 'android.arch.core:core-testing:2.1.0'
    testImplementation 'org.mockito:mockito-core:3.2.4'
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation 'androidx.test:rules:1.3.0'
    androidTestImplementation 'androidx.test
      .espresso:espresso-core:3.3.0'
    androidTestImplementation 'com.android.support.test
      .espresso:espresso-contrib:3.0.2'
```

2. Add the Kotlin application processor plugin to the **`app/build.gradle`** file:

```
apply plugin: 'kotlin-kapt'
```

3. Create the **`api`** package.

4. Add the model for **`api`**:

```
data class Post(
    @SerializedName("id") val id: Long,
    @SerializedName("userId") val userId: Long,
    @SerializedName("title") val title: String,
    @SerializedName("body") val body: String
)
```

5. Add the class responsible for retrieving the list of posts:

```
interface PostService {

    @GET("posts")
    fun getPosts(): Call<List<Post>>
}
```

6. Create the **`repository`** package and add the following interface:

```
interface PostRepository {

    fun getPosts(): LiveData<List<Post>>
}
```

7. Create the implementation for the preceding class in which the error scenarios will not be handled:

```
class PostRepositoryImpl(private val postService: PostService) :
  PostRepository {

    override fun getPosts(): LiveData<List<Post>> {
        val result = MutableLiveData<List<Post>>()
        postService.getPosts().enqueue(object :
          Callback<List<Post>> {

            override fun onFailure(call: Call<List<Post>>,
              t: Throwable) {
            }

            override fun onResponse(call: Call<List<Post>>,
              response: Response<List<Post>>) {
                if (response.isSuccessful) {
                    result.postValue(response.body())
                }
            }
        })
        return result
    }
}
```

8. And now, let's write one test for this class:

```
@RunWith(MockitoJUnitRunner::class)
class PostRepositoryImplTest {

    @get:Rule
    val rule = InstantTaskExecutorRule()
    @InjectMocks
    lateinit var postRepository: PostRepositoryImpl
    @Mock
    lateinit var postService: PostService
    @Mock
    lateinit var call: Call<List<Post>>

    @Before
    fun setUp() {
        Mockito.'when'(postService.getPosts()).thenReturn(call)
    }

    @Test
    fun getPosts_success() {
        val postList = listOf(
            Post(1, 1, "title1", "body1"),
            Post(2, 2, "title2", "body2")
        )
        Mockito.'when'(call.enqueue(Mockito.any())).thenAnswer {
            (it.arguments[0] as Callback<List<Post>>)
              .onResponse(call, Response.success(postList))
        }

        val result = postRepository.getPosts()

        assertEquals(postList, result.value)
    }
}
```

9. Next, let's define **`PostViewModel`**:

```
class PostViewModel(private val postRepository:
  PostRepository) : ViewModel() {

    fun getPosts(): LiveData<List<Post>> =
      postRepository.getPosts()
}
```

10. And now, let's write the test for this class:

```
@RunWith(MockitoJUnitRunner::class)
class PostViewModelTest {

    @InjectMocks
    lateinit var postViewModel: PostViewModel
    @Mock
    lateinit var postRepository: PostRepository

    @Test
    fun getPosts() {
        val expected = Mockito.mock(LiveData::class.java)
        Mockito.'when'(postRepository.getPosts())
            .thenReturn(expected as LiveData<List<Post>>?)

        val result = postViewModel.getPosts()

        assertEquals(expected, result)
    }
}
```

11. Now, let's create the **`view_post_row.xml`** file in which we define the layout for every row:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
  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:padding="10dp">

    <TextView
        android:id="@+id/view_post_row_title"
```

```
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

        <TextView
            android:id="@+id/view_post_row_body"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="5dp"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf
              ="@id/view_post_row_title" />

    </androidx.constraintlayout.widget.ConstraintLayout>
```

12. Next, let's create the **`PostAdapter`** class responsible for binding the row layout:

```
    class PostAdapter(private val layoutInflater: LayoutInflater) :
        RecyclerView.Adapter<PostAdapter.PostViewHolder>() {

        private val posts = mutableListOf<Post>()

        override fun onCreateViewHolder(parent: ViewGroup, viewType:
          Int): PostViewHolder =
            PostViewHolder(layoutInflater.inflate
              (R.layout.view_post_row, parent, false))

        override fun getItemCount(): Int = posts.size

        override fun onBindViewHolder(holder: PostViewHolder,
          position: Int) {
            holder.bind(posts[position])
        }

        fun updatePosts(posts: List<Post>) {
            this.posts.clear()
            this.posts.addAll(posts)
            this.notifyDataSetChanged()
        }
```

```
        inner class PostViewHolder(containerView: View) :
          RecyclerView.ViewHolder(containerView) {

            private val titleTextView: TextView =
                containerView.findViewById<TextView>
                  (R.id.view_post_row_title)
            private val bodyTextView: TextView =
                containerView.findViewById<TextView>
                  (R.id.view_post_row_body)

            fun bind(post: Post) {
                bodyTextView.text = post.body
                titleTextView.text = post.title
            }
        }
    }
```

13. Now, add following code to the **`activity_main.xml`** file:

```
    <?xml version="1.0" encoding="utf-8"?>
    <androidx.recyclerview.widget.RecyclerView
      xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/activity_main_recycler_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
```

14. Next, let's add the following to the **`onCreate`** method of **`MainActivity`**:

```
    class MainActivity : AppCompatActivity() {

        private lateinit var postAdapter: PostAdapter

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

            postAdapter = PostAdapter(LayoutInflater.from(this))
            val recyclerView = findViewById<RecyclerView>
              (R.id.activity_main_recycler_view)
            recyclerView.adapter = postAdapter
            recyclerView.layoutManager = LinearLayoutManager(this)
        }
    }
```

15. Make sure the **`MAIN`** intent filter is added to **`AndroidManifest.xml`**:

```
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name=
                  "android.intent.action.MAIN" />

                <category android:name=
                  "android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
```

16. Let's create an empty **`Application`** class named **`MyApplication`**:

```
class MyApplication : Application() {

    override fun onCreate() {
        super.onCreate()
    }
}
```

17. Now, let's set up Dagger in the project. We can start with **`NetworkModule`** in which we will provide a dependency to Retrofit and one to **`PostService`**:

```
@Module
class NetworkModule {

    @Singleton
    @Provides
    fun provideRetrofit(): Retrofit {
        return Retrofit.Builder()
            .baseUrl("https://jsonplaceholder.typicode.com/")
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }

    @Singleton
    @Provides
    fun providePostService(retrofit: Retrofit): PostService {
        return retrofit.create<PostService>
          (PostService::class.java)
    }
}
```

18. Next, let's create a **`RepositoryModule`** class:

```
@Module
class RepositoryModule {

    @Singleton
    @Provides
    fun providePostRepository(postService: PostService):
      PostRepository {
        return PostRepositoryImpl(postService)
    }
}
```

19. We will need a **`MainActivityModule`** class in which we will provide the **`ViewModelProvider.Factory`** instance with **`PostViewModel`**:

```
@Module
class MainActivityModule {

    @Provides
    fun provideViewModelFactory(postRepository: PostRepository):
      ViewModelProvider.Factory {
        return object : ViewModelProvider.Factory {
            override fun <T : ViewModel?>
              create(modelClass: Class<T>): T {
                return PostViewModel(postRepository) as T
            }
        }
    }
}
```

20. We will need a subcomponent that will use the preceding module, which will contain a method to inject dependencies into **`MainActivity`**:

```
@Subcomponent(modules = [MainActivityModule::class])
interface MainActivitySubcomponent {

    fun inject(mainActivity: MainActivity)
}
```

21. Now, we will need to move on to creating the **`ApplicationComponent`** interface, which contains a method to create the preceding subcomponent and will initialize the dependency graph with **`NetworkModule`** and **`RepositoryModule`**:

```
@Singleton
@Component(modules = [NetworkModule::class, RepositoryModule::class])
interface ApplicationComponent {

    fun createActivitySubcomponent(): MainActivitySubcomponent
}
```

22. In the **`MyApplication`** class, initialize **`ApplicationComponent`**:

```
lateinit var  applicationComponent : ApplicationComponent

    override fun onCreate() {
        super.onCreate()
        applicationComponent =
          DaggerApplicationComponent.create()
    }
```

23. Finally, inject **`ViewModelProvider.Factory`** into **`MainActivity`** and obtain the **`ViewModel`** reference:

```
    @Inject
    lateinit var factory: ViewModelProvider.Factory
    private lateinit var postAdapter: PostAdapter

    override fun onCreate(savedInstanceState: Bundle?) {
        (application as MyApplication).applicationComponent
          .createActivitySubcomponent()
            .inject(this)
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        postAdapter = PostAdapter(LayoutInflater.from(this))
        val recyclerView = findViewById<RecyclerView>
            (R.id.activity_main_recycler_view)
        recyclerView.adapter = postAdapter
        recyclerView.layoutManager = LinearLayoutManager(this)
        val viewModel = ViewModelProvider(this,
          factory).get(PostViewModel::class.java)
        viewModel.getPosts().observe(this, Observer {
```

```
                postAdapter.updatePosts(it)
            })
        }
```

24. Add the internet permission to **`AndroidManifest.xml`**:

```
<manifest ...>
...
    <uses-permission android:name=
      "android.permission.INTERNET" />
...
</manifest>
```

25. If you run the app at this point, the posts should be displayed on the screen:

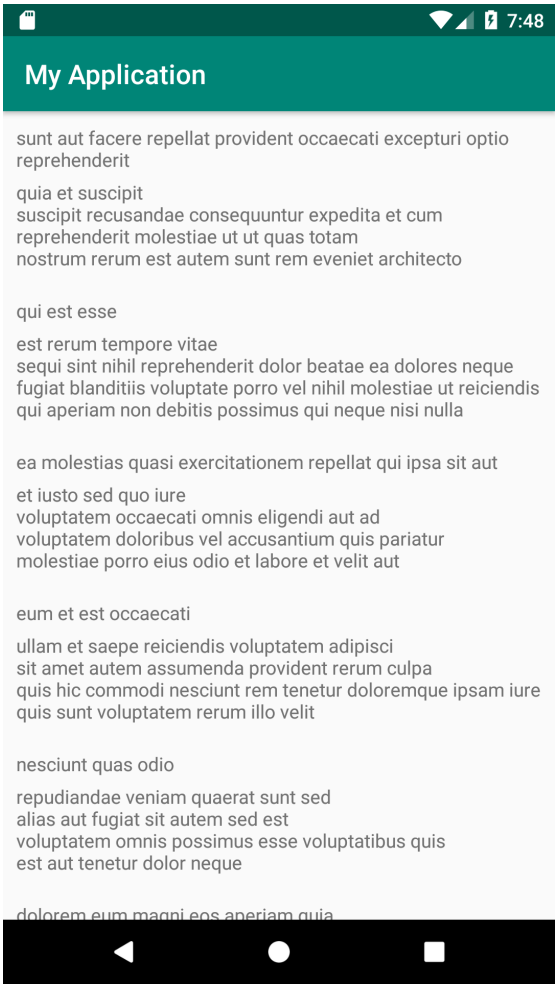

Figure 12.7: Displaying the list of posts in a Dagger application

26. Let's work on setting up the dummy data for the UI test. In order to do this, we will need to create a class that will output a list of dummy posts. Normally, we would need to go to the API level to make sure that the components work well together. However, in order to simplify the work, we will work on the repository layer. What we need to do is create **`DummyRepository`**, which will return a list of posts, and, using Dagger, we will inject it into **`ViewModel`**:

```
class DummyRepository : PostRepository {

    override fun getPosts(): LiveData<List<Post>> {
        val liveData = MutableLiveData<List<Post>>()
        liveData.postValue(
            listOf(
                Post(1L, 1L, "Title 1", "Body 1"),
                Post(2L, 1L, "Title 2", "Body 2"),
                Post(3L, 1L, "Title 3", "Body 3")
            )
        )
        return liveData
    }
}
```

27. Next, we have to tweak **`RepositoryModule`** in order to be able to extend it from the test, so we will have to make the class open as well as the function we want to override:

```
@Module
open class RepositoryModule {

    @Singleton
    @Provides
    open fun providePostRepository(postService: PostService):
      PostRepository {
        return PostRepositoryImpl(postService)
    }
}
```

28. Now, we should move on to the **androidTest** package and create a **TestRepositoryModule** class, which will extend **RepositoryModule**:

```
@Module
class TestRepositoryModule : RepositoryModule() {

    @Singleton
    @Provides
    override fun providePostRepository(postService: PostService):
      PostRepository {
        return DummyRepository()
    }
}
```

29. Next, we will need to modify the **MyApplication** class in order to provide the repository module programmatically and make it extendable:

```
open class MyApplication : Application() {

    lateinit var applicationComponent: ApplicationComponent

    override fun onCreate() {
        super.onCreate()
        applicationComponent =
          DaggerApplicationComponent.builder()
            .repositoryModule(createRepositoryModule())
            .build()
    }

    open fun createRepositoryModule(): RepositoryModule {
        return RepositoryModule()
    }
}
```

30. Next, let's create a **`TestApplication`** class in the **`androidTest`** package that will extend from **`MyApplication`** and inject **`TestRepositoryModule`**:

```
class TestApplication :MyApplication() {

    override fun createRepositoryModule(): RepositoryModule {
        return TestRepositoryModule()
    }
}
```

31. Now, let's create a **`TestRunner`** class and add **`TestApplication`** as the target of the test:

```
class MyTestRunner : AndroidJUnitRunner() {

    @Throws(Exception::class)
    override fun newApplication(
        cl: ClassLoader?,
        className: String?,
        context: Context?
    ): Application? {
        return super.newApplication(cl,
          TestApplication::class.java.name, context)
    }
}
```

32. Now, add the test runner to the **`app/build.gradle`** configuration:

```
android {
    ...
    defaultConfig {
        ...
        testInstrumentationRunner
          "com.android.myapplication.MyTestRunner"
    }
}
```

33. Let's create the **`MainActivityTest`** class in which we assert that the dummy data will be displayed on the screen:

```
@RunWith(AndroidJUnit4::class)
class MainActivityTest {
    @JvmField
    @Rule
    var activityRule: ActivityTestRule<MainActivity> =
      ActivityTestRule(MainActivity::class.java)

    @Test
    fun testDisplaysPosts() {
        onView(withText("Title 1")).check(matches(isDisplayed()))
        onView(withText("Body 1")).check(matches(isDisplayed()))
        onView(withText("Title 2")).check(matches(isDisplayed()))
        onView(withText("Body 2")).check(matches(isDisplayed()))
        onView(withText("Title 3")).check(matches(isDisplayed()))
        onView(withText("Body 3")).check(matches(isDisplayed()))
    }
}
```

If the test is successful, this means that the test setup we have used for Dagger has worked.

ACTIVITY 12.02: KOIN-INJECTED REPOSITORIES

Solution:

1. Update the **`app/build.gradle`** file to include Koin:

```
    implementation
  "androidx.constraintlayout:constraintlayout:2.0.4"
    implementation 'androidx.recyclerview:recyclerview:1.1.0'
    def lifecycle_version = "2.2.0"
    implementation "androidx.lifecycle:lifecycle-
      extensions:$lifecycle_version"
    implementation 'com.squareup.retrofit2:retrofit:2.6.2'
    implementation 'com.squareup.retrofit2:converter-gson:2.6.2'
    implementation 'com.google.code.gson:gson:2.8.6'
    implementation 'org.koin:koin-android:2.2.0-rc-4'
    implementation "org.koin:koin-android-viewmodel:2.2.0-rc-4"

    testImplementation 'junit:junit:4.12'
    testImplementation 'android.arch.core:core-testing:2.1.0'
```

```
    testImplementation 'org.mockito:mockito-core:3.2.4'
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation 'androidx.test:rules:1.3.0'
    androidTestImplementation 'androidx.test
      .espresso:espresso-core:3.3.0'
    androidTestImplementation 'com.android.support.test
      .espresso:espresso-contrib:3.0.2'
```

2. Delete all the Dagger modules, components, and subcomponents.
3. Add the required Koin modules and configure Koin in the **`MyApplication`** class:

```
open class MyApplication : Application() {

    private val networkModule = module {
        single {
            Retrofit.Builder()
                .baseUrl("https://jsonplaceholder.typicode.com/")
                .addConverterFactory(GsonConverterFactory
                  .create())
                .build()
        }

        single {
            providePostService(get())
        }
    }
    private val repositoryModule = module {
        single {
            providePostRepository(get())
        }
    }
    private val viewModelModule = module {
        viewModel {
            PostViewModel(get())
        }
    }

    override fun onCreate() {
        super.onCreate()
        startKoin {
            androidContext(this@MyApplication)
```

```
            modules(listOf(networkModule, repositoryModule,
              viewModelModule))
        }
    }

    private fun providePostService(retrofit: Retrofit):
      PostService {
        return retrofit.create(PostService::class.java)
    }

    open fun providePostRepository(postService: PostService):
      PostRepository {
        return PostRepositoryImpl(postService)
    }
}
```

4. Inject **`PostViewModel`** into **`MainActivity`** using Koin:

```
class MainActivity : AppCompatActivity() {

    private lateinit var postAdapter: PostAdapter

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

        postAdapter = PostAdapter(LayoutInflater.from(this))
        val recyclerView =
         findViewById<RecyclerView>
           (R.id.activity_main_recycler_view)
        recyclerView.adapter = postAdapter
        recyclerView.layoutManager = LinearLayoutManager(this)
        val viewModel: PostViewModel = getViewModel()
        viewModel.getPosts().observe(this, Observer {
            postAdapter.updatePosts(it)
        })
    }
}
```

5. Change **`TestApplication`** to return **`DummyRepository`**:

```
class TestApplication : MyApplication() {

    override fun providePostRepository(postService: PostService):
      PostRepository {
```

```
            return DummyRepository()
        }
    }
```

The final output will be as follows:

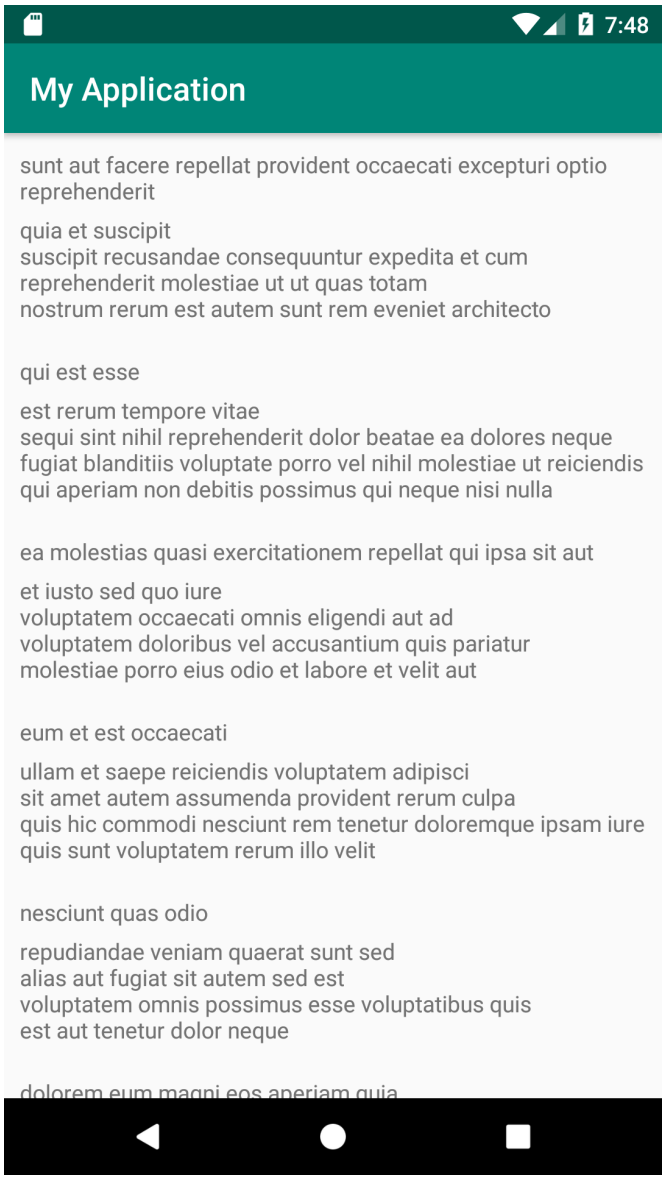

Figure 12.8: Displaying the list of posts in a Koin application

If you run the application and the test, they should provide the same output as in the previous activity, as presented in *Figure 12.8*.

CHAPTER 13: RXJAVA AND COROUTINES

ACTIVITY 13.01: CREATING A TV GUIDE APP

Solution:

Here is one way you can develop the TV Guide app:

1. Create a new project in Android Studio named **`TV Guide`** with a package name of **`com.example.tvguide`**.
2. Add the **`INTERNET`** permission in the **`AndroidManifest.xml`** file:

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

3. Open your **`app/build.gradle`** file and add the kotlin-parcelize plugin at the end of the plugins block:

```
plugins {
    ...
    id 'kotlin-parcelize'
}
```

   This will allow you to use Parcelable for the model class.

4. Add Java 8 compatibility in your **`app/build.gradle`** file's **`android`** block:

```
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}

kotlinOptions {
    jvmTarget = '1.8'
}
```

5. Add the RecyclerView, Glide, Retrofit, RxJava, RxAndroid, Moshi, ViewModel, and LiveData libraries to your project by adding the following in your **`app/build.gradle`** file:

```
implementation 'androidx.recyclerview:recyclerview:1.1.0'
implementation 'com.squareup.retrofit2:retrofit:2.9.0'

implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.9'
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-
android:1.3.9'

implementation 'com.squareup.retrofit2:converter-moshi:2.9.0'

implementation 'com.github.bumptech.glide:glide:4.11.0'
implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.2.0'
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.2.0'
```

6. Add a **`layout_margin`** dimension value by creating a **`dimens.xml`** file in the **`res/values`** directory with the following:

```
<resources>
    <dimen name="layout_margin">16dp</dimen>
</resources>
```

This will be used for the view margins in the layout.

7. Create a **`view_tv_show_item.xml`** layout file with **`ImageView`** for the poster and **`TextView`** for the name of the TV show:

```
<ImageView
    android:id="@+id/tv_poster"
    android:layout_width="match_parent"
    android:layout_height="240dp"
    android:contentDescription="Poster"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:src="@tools:sample/backgrounds/scenic" />

<TextView
    android:id="@+id/tv_show_title"
    android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:layout_marginStart="@dimen/layout_margin"
        android:layout_marginEnd="@dimen/layout_margin"
        android:ellipsize="end"
        android:gravity="center"
        android:lines="1"
        android:textSize="20sp"
        app:layout_constraintEnd_toEndOf="@id/tv_poster"
        app:layout_constraintStart_toStartOf="@id/tv_poster"
        app:layout_constraintTop_toBottomOf="@id/tv_poster"
        tools:text="TV Show" />
```

This layout file will be used for each TV show in the list.

8. Remove the Hello World TextView in **`activity_main.xml`** and add a RecyclerView for the list of TV shows:

```
        <androidx.recyclerview.widget.RecyclerView
            android:id="@+id/tv_show_list"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            app:layoutManager
              ="androidx.recyclerview.widget.GridLayoutManager"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            app:spanCount="2"
            tools:listitem="@layout/view_tv_show_item" />
```

This RecyclerView with **`LinearLayoutManager`** will display the list of TV shows.

9. Create a model class, **`TVShow`**, in a new package, **`com.example.tvguide.model`**:

```
    @Parcelize
    data class TVShow(
        val backdrop_path: String? = "",
        val first_air_date: String = "",
        val id: Int = 0,
        val name: String = "",
        val original_language: String = "",
        val original_name: String = "",
        val overview: String = "",
        val popularity: Float = 0f,
```

```
    val poster_path: String? = "",
    val vote_average: Float = 0f,
    val vote_count: Int = 0
) : Parcelable
```

This will be the model class representing a **`TVShow`** object from the API.

10. Create a new activity named **`DetailsActivity`** with **`activity_details.xml`** as the layout file.

11. Open the **`AndroidManifest.xml`** file and add the **`parentActivityName`** attribute in the **`DetailsActivity`** declaration:

```
    <activity android:name=".DetailsActivity"
            android:parentActivityName=".MainActivity" />
```

This adds an up icon in the details activity for going back to the main screen.

12. Open the **`activity_details.xml`** file. Add the views for the details of the TV show:

```
<ImageView
    android:id="@+id/tv_poster"
    android:layout_width="match_parent"
    android:layout_height="240dp"
    android:layout_margin="@dimen/layout_margin"
    android:contentDescription="Poster"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:src="@tools:sample/avatars" />

<TextView
    android:id="@+id/title_text"
    style="@style/TextAppearance.AppCompat.Medium"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/layout_margin"
    app:layout_constraintEnd_toEndOf="@+id/tv_poster"
    app:layout_constraintStart_toStartOf="@+id/tv_poster"
    app:layout_constraintTop_toBottomOf="@+id/tv_poster"
    tools:text="Name" />

<TextView
    android:id="@+id/release_text"
```

```
        style="@style/TextAppearance.AppCompat.Medium"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/layout_margin"
        app:layout_constraintEnd_toEndOf="@+id/title_text"
        app:layout_constraintStart_toStartOf="@+id/title_text"
        app:layout_constraintTop_toBottomOf="@+id/title_text"
        tools:text="Release Date" />

    <TextView
        android:id="@+id/overview_text"
        style="@style/TextAppearance.AppCompat.Medium"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/layout_margin"
        app:layout_constraintEnd_toEndOf="@+id/release_text"
        app:layout_constraintStart_toStartOf="@+id/release_text"
        app:layout_constraintTop_toBottomOf="@+id/release_text"
        tools:text="Overview" />
```

This will add **`ImageView`** for the poster and multiple TextViews for the name, release date, and overview of the TV show on the details screen.

13. Open **`DetailsActivity`**. Add the following:

```
    companion object {
        const val EXTRA_TV_SHOW = "tvshow"
        const val IMAGE_URL = "https://image.tmdb.org/t/p/w185/"
    }

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

            val titleText: TextView = findViewById(R.id.title_text)
            val releaseText: TextView = findViewById(R.id.release_text)
            val overviewText: TextView = findViewById(R.id.overview_text)
            val poster: ImageView = findViewById(R.id.tv_poster)

            val tvShow = intent.getParcelableExtra<TVShow>(EXTRA_TV_SHOW)
            tvShow?.run {
                titleText.text = name
```

```
            releaseText.text = "First Air Date:
              ${first_air_date.take(4)}"
            overviewText.text = "Overview: $overview"

            Glide.with(this@DetailsActivity)
                .load($IMAGE_URL$poster_path)
                .placeholder(R.mipmap.ic_launcher)
                .fitCenter()
                .into(poster)
        }
    }
```

This will display the poster, name, release, and overview of the TV show selected.

14. Create a **`TVShowAdapter`** adapter class for the list of TV shows with the following contents:

```
class TVShowAdapter(private val clickListener: TVClickListener) :
  RecyclerView.Adapter<TVShowAdapter.TVShowViewHolder>() {

    private val tvShows = mutableListOf<TVShow>()

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

    override fun getItemCount() = tvShows.size

    override fun onBindViewHolder(holder: TVShowViewHolder,
      position: Int) {
        val tvShow = tvShows[position]
        holder.bind(tvShow)
        holder.itemView.setOnClickListener {
          clickListener.onShowClick(tvShow) }
    }

    fun addTVShows(shows: List<TVShow>) {
        tvShows.addAll(shows)
        notifyItemRangeInserted(0, shows.size)
```

```
        }


        class TVShowViewHolder(itemView: View) :
          RecyclerView.ViewHolder(itemView) {
            private val imageUrl = "https://image.tmdb.org/t/p/w185/"
            private val titleText: TextView by lazy {
                itemView.findViewById(R.id.tv_show_title)
            }
            private val poster: ImageView by lazy {
                itemView.findViewById(R.id.tv_poster)
            }

            fun bind(show: TVShow) {
                titleText.text = show.name

                Glide.with(itemView.context)
                    .load("$imageUrl${show.poster_path}")
                    .placeholder(R.mipmap.ic_launcher)
                    .fitCenter()
                    .into(poster)
            }
        }

        interface TVClickListener {
            fun onShowClick(show: TVShow)
        }
    }
```

This class will be the RecyclerView's adapter and ViewHolder.

15. Create another class named **`TVResponse`** in the **`com.example.tvguide.model`** package:

```
data class TVResponse (
    val page: Int,
    val results: List<TVShow>
)
```

This will be the model class for the response you get from the API endpoint for the TV shows on air.

16. Create **`TelevisionService`** in the **`com.example.tvguide.api`** package:

```
interface TelevisionService {

    @GET("tv/on_the_air")
    suspend fun getTVShows(@Query("api_key") apiKey: String):
      TVResponse
}
```

This will define the endpoint you will use to retrieve the TV shows that are on the air.

17. Create a **`TVShowRepository`** class with a constructor for **`tvService`**:

```
class TVShowRepository(private val tvService: TelevisionService) {
... }
```

18. Add **`tvShowsLiveData`** for the list of TV shows, errorLiveData for the error message, and replace the **`apiKey`** with the API key you got from The Movie Database API:

```
private val apiKey = "your_api_key_here"

private val tvShowsLiveData = MutableLiveData<List<TVShow>>()
private val errorLiveData = MutableLiveData<String>()

val tvShows: LiveData<List<TVShow>>
    get() = tvShowsLiveData
val error: LiveData<String>
    get() = errorLiveData
```

19. Create a suspending function, **`fetchTVShows`**, to retrieve the list from the endpoint:

```
suspend fun fetchTVShows() {
        try {
            val shows = tvService.getTVShows(apiKey)
            tvShowsLiveData.postValue(shows.results)
        } catch (exception: Exception) {
            errorLiveData.postValue("An error occurred:
              ${exception.message}")
        }
    }
```

20. Create a **`TVShowViewModel`** class with a constructor for **`tvShowRepository`**:

```
class TVShowViewModel(private val tvShowRepository: TVShowRepository)
:
  ViewModel() {
    ...
}
```

21. Add a **`getTVShows`** function that returns a LiveData for the list of TV shows and **`getError`** function that returns a LiveData for error message:

```
fun getTVShows(): LiveData<List<TVShow>>
  = tvShowRepository.tvShows.map { shows ->
        shows.sortedBy { it.name }
    }

fun getError(): LiveData<String> = tvShowRepository.error
```

22. Add the **`fetchTVShows`** function with the coroutine using **`viewModelScope`** to fetch the TV shows from **`tvShowRepository`** when the **`TVShowViewModel`** initializes:

```
init {
    fetchTVShows()
}

private fun fetchTVShows() {
        viewModelScope.launch(Dispatchers.IO)  {
            tvShowRepository.fetchTVShows()
        }
    }
```

23. Create an application class named **`TVApplication`** with a property for **`tvShowRepository`**:

```
class TVApplication : Application() {
    lateinit var tvShowRepository: TVShowRepository

}
```

This will be the application class for the app. It will hold a reference to **`tvShowRepository`**.

24. Override the **`onCreate`** function of the **`TVApplication`** class and initialize the **`tvService`** and **`tvShowRepository`** objects:

```
override fun onCreate() {
        super.onCreate()

        val retrofit = Retrofit.Builder()
            .baseUrl("https://api.themoviedb.org/3/")
            .addConverterFactory(MoshiConverterFactory.create())
            .build()
        val tvService = retrofit.create(TelevisionService::class.java)
        tvShowRepository = TVShowRepository(tvService)
    }
```

25. Set **`TVApplication`** as the value for the **`android:name`** attribute of the application in the **`AndroidManifest.xml`** file:

```
<application
    ...
    android:name=".TVApplication"
    ... />
```

26. Open **`MainActivity`** and add define a field for the adapter for the TV shows:

```
 private val tvShowAdapter by lazy {
    TVShowAdapter(object : TVShowAdapter.TVClickListener {
        override fun onShowClick(show: TVShow) {
            openShowDetails(show)
        }
    })
}
```

    This will create an adapter with the list of TV shows. When a TV show is clicked, the **`openShowDetails`** function will be called.

27. In the **`onCreate`** function, set the adapter for the **`tv_show_list`** RecyclerView:

```
val tvShowRecyclerView: RecyclerView = findViewById(R.id.tv_show_list)
tvShowRecyclerView.adapter = tvShowAdapter
```

28. After that line, add the following code to initialize **`tvShowRepository`** and **`tvShowViewModel`**:

```
val tvShowRepository = (application as TVApplication).
tvShowRepository
        val tvShowViewModel = ViewModelProvider(this, object:
          ViewModelProvider.Factory {
            override fun <T : ViewModel?>
              create(modelClass: Class<T>): T {
                return TVShowViewModel(tvShowRepository) as T
            }
        }).get(TVShowViewModel::class.java)
```

29. Then, below that, create an observer for **`getTVshows`** and **`getError`** from **`tvShowViewModel`**:

```
tvShowViewModel.getTVShows().observe(this, { shows ->
    tvShowAdapter.addTVShows(shows)
}
tvShowViewModel.getError().observe(this, { error ->
    Toast.makeText(this, error, Toast.LENGTH_LONG).show()
})
```

This will update the activity's list with the TV shows fetched.

30. Add the **`openShowDetails`** function to open the details screen when clicking on a TV show from the list:

```
private fun openShowDetails(tvShow: TVShow) {
        val intent = Intent(this, DetailsActivity::class.java).apply
{
            putExtra(DetailsActivity.EXTRA_TV_SHOW, tvShow)
        }
        startActivity(intent)
}
```

31. Run your application. The app will display a list of TV shows. Click on a TV show, and you will see its details, such as the release year and an overview:

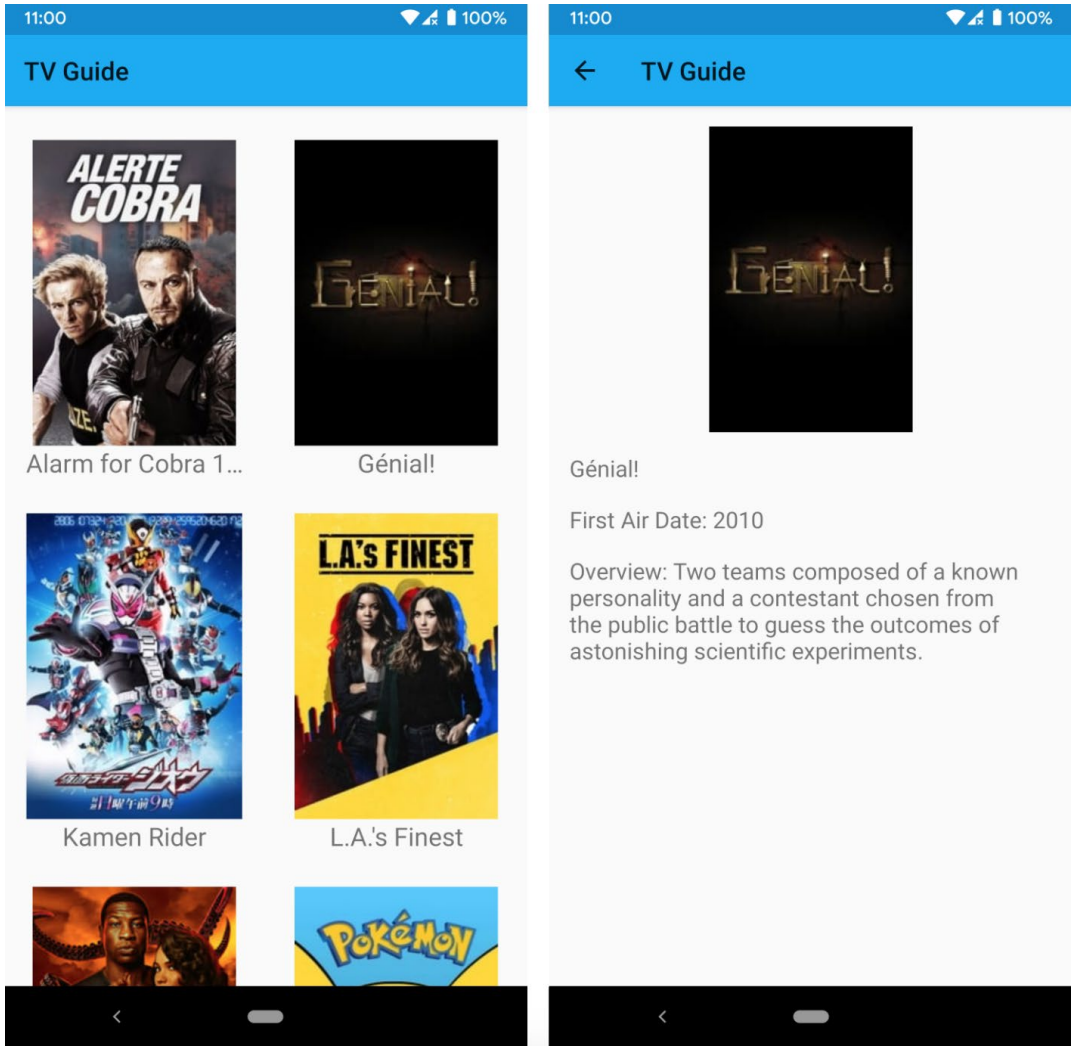

Figure 13.12: The main screen and details screen of the TV Guide app

CHAPTER 14: ARCHITECTURE PATTERNS

ACTIVITY 14.01: REVISITING THE TV GUIDE APP

Solution:

You can use the TV Guide app you developed in the previous chapter or make a copy of it. Here is one way you can improve the app using the MVVM architectural pattern with data binding, the Repository pattern with Room, and WorkManager:

1. Open the TV Guide app in Android Studio.
2. Open the **`app/build.gradle`** file and add the **`kotlin-kapt`** plugin at the end of the plugins block:

```
plugins {
    ...
    id 'kotlin-kapt'
}
```

3. Add the data binding dependency in the **`android`** block:

```
buildFeatures {
    dataBinding true
}
```

   This will enable data binding in your application.

4. Add the dependencies for the Room library and WorkManager:

```
implementation 'androidx.room:room-runtime:2.2.5'
implementation 'androidx.room:room-ktx:2.2.5'
kapt 'androidx.room:room-compiler:2.2.5'

implementation 'androidx.work:work-runtime:2.4.0'
```

   This will allow you to use Room and WorkManager in your project.

5. Create a **`RecyclerViewBinding`** class that contains the binding adapter for the **`RecyclerView`** list:

```
@BindingAdapter("list")
fun bindTVShows(view: RecyclerView, tvShows: List<TVShow>?) {
    val adapter = view.adapter as TVShowAdapter
    adapter.addTVShows(tvShows ?: emptyList())
}
```

This adds the **app:list** attribute for **RecyclerView**, wherein you can pass the list of TV shows that will be set to the adapter to update the **RecyclerView** content.

6. Open the **activity_main.xml** file and wrap everything inside a **layout** tag:

```
<layout 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">

    <androidx.constraintlayout.widget.ConstraintLayout
        ...
</layout>
```

This will allow the data binding library to generate a binding class for the layout.

7. Inside the **layout** tag and before the **ConstraintLayout** tag, add a data element with a variable for the **viewModel**:

```
<data>
    <variable
        name="viewModel"
        type="com.example.tvguide.TVShowViewModel" />
</data>
```

The **viewModel** layout variable corresponds to your **TVShowViewModel** class.

8. In **RecyclerView**, add the list to be displayed with **app:list**:

```
app:list="@{viewModel.TVShows}"
```

This will pass the **TVShows LiveData** from the **getTVShows** function of **TVShowViewModel** into **RecyclerView**.

9. Open **MainActivity**, remove the line for **setContentView**, and add the following:

```
val binding: ActivityMainBinding = DataBindingUtil.
setContentView(this,
  R.layout.activity_main
```

10. Remove the observer from **TVShowViewModel** and replace it with the data binding code:

```
binding.viewModel = tvShowViewModel
binding.lifecycleOwner = this
```

This binds the **`tvShowViewModel`** to the **`viewModel`** layout variable in the **`activity_main.xml`** file.

11. Run your application. It will display a list of TV shows. Clicking on a TV show will open a details screen where you can see additional information about the show, such as the release year and a plot overview:

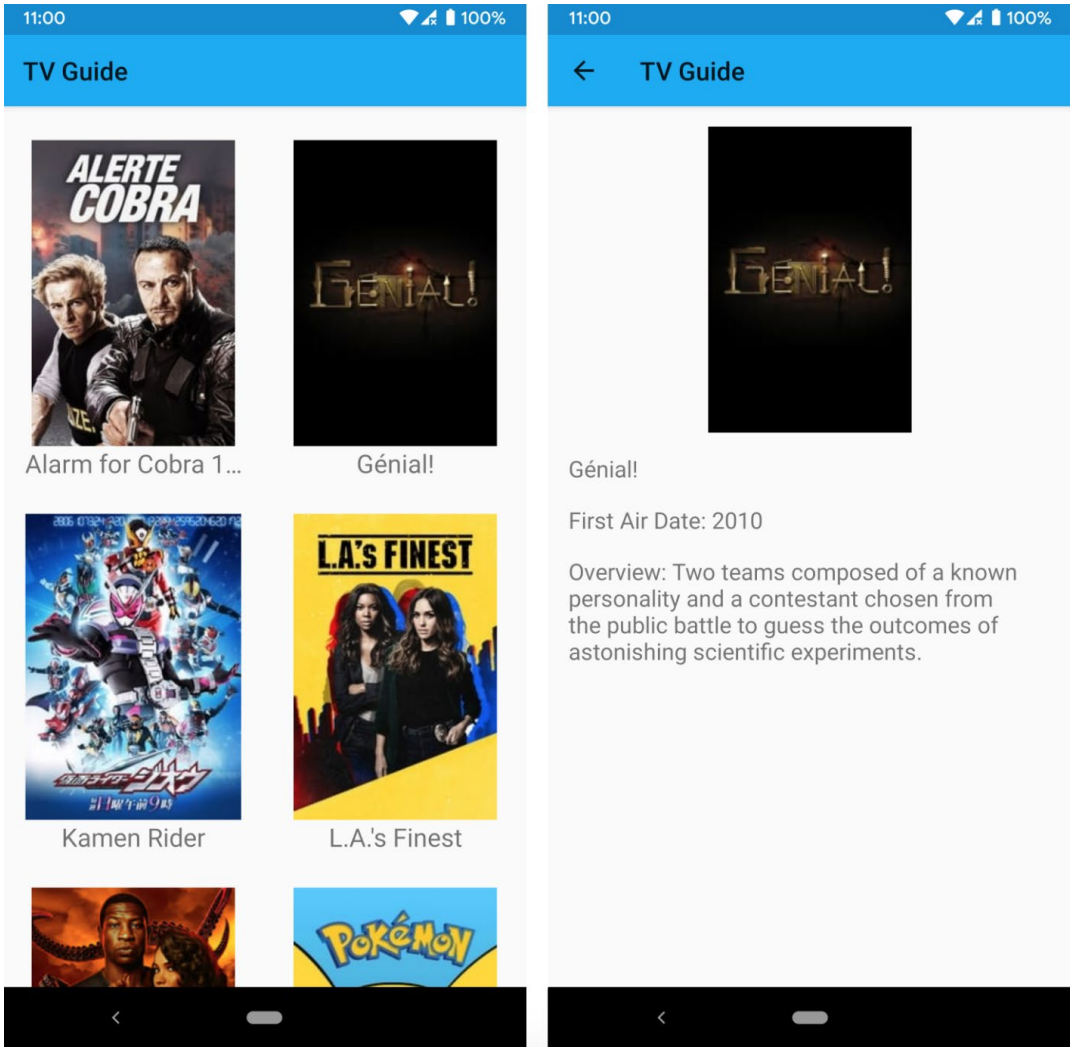

Figure 14.7: The TV Guide app looks the same with data binding

You have added data binding in the TV Guide app. In the next steps, you will be using Room to cache the list of TV shows in the local database.

12. Open the **`TVShow`** class and add an **`Entity`** annotation for it:

```
@Entity(tableName = "tvshows",  primaryKeys = [("id")])
data class TVShow( ... )
```

This creates a **`tvShows`** table for the list of TV shows, with **`id`** as the primary key.

13. Create a **`TVDao`** data access object for accessing the TV shows table in a new package called **`com.example.tvguide.database`**:

```
@Dao
interface TVDao {
@Insert(onConflict = OnConflictStrategy.REPLACE)
fun addTVShows(tvShows: List<TVShow>)

@Query("SELECT * FROM tvshows")
fun getTVShows(): List<TVShow>
}
```

This class has a function for getting the list of TV shows from the database and another function for adding a list to the database.

14. Create a **`TVDatabase`** class in the **`com.example.tvguide.database`** package:

```
@Database(entities = [TVShow::class], version = 1)
abstract class TVDatabase : RoomDatabase() {

    abstract fun tvDao(): TVDao

    companion object {
        @Volatile
        private var instance: TVDatabase? = null
        fun getInstance(context: Context): TVDatabase {
            return instance ?: synchronized(this) {
                instance ?: buildDatabase(context).also
                  { instance = it }
            }
        }

        private fun buildDatabase(context: Context): TVDatabase {
            return Room.databaseBuilder(context,
```

```
                TVDatabase::class.java, "tvshows-db")
                  .build()
            }
        }
    }
```

This database has a **`version`** of **`1`**, a single entity for **`TVShow`**, and a data access object for the TV shows.

15. Update the **`TVShowRepository`** class with a constructor for **`tvDatabase`**:

```
class TVShowRepository(private val tvService: TelevisionService,
private
  val tvDatabase: TVDatabase) { ... }
```

16. Update the **`fetchTVShows`** function to get the TV shows from the database. If there's nothing yet, retrieve the list from the endpoint and save it:

```
suspend fun fetchTVShows() {
    val tvDao: TVDao = tvDatabase.tvDao()
    var shows = tvDao.getTVShows()
    if (shows.isEmpty()) {
        try {
            val tvResponse = tvService.getTVShows(apiKey)
            shows = tvResponse.results
            tvDao.addTVShows(shows)
        } catch (exception: Exception) {
           errorLiveData.postValue("An error occurred:
             ${exception.message}")
        }
    }

    tvShowsLiveData.postValue(shows)
}
```

17. Run your application. It will display a list of TV shows. If you turn off mobile data or disconnect from the wireless network, you will still see the list because it is now cached in the database:

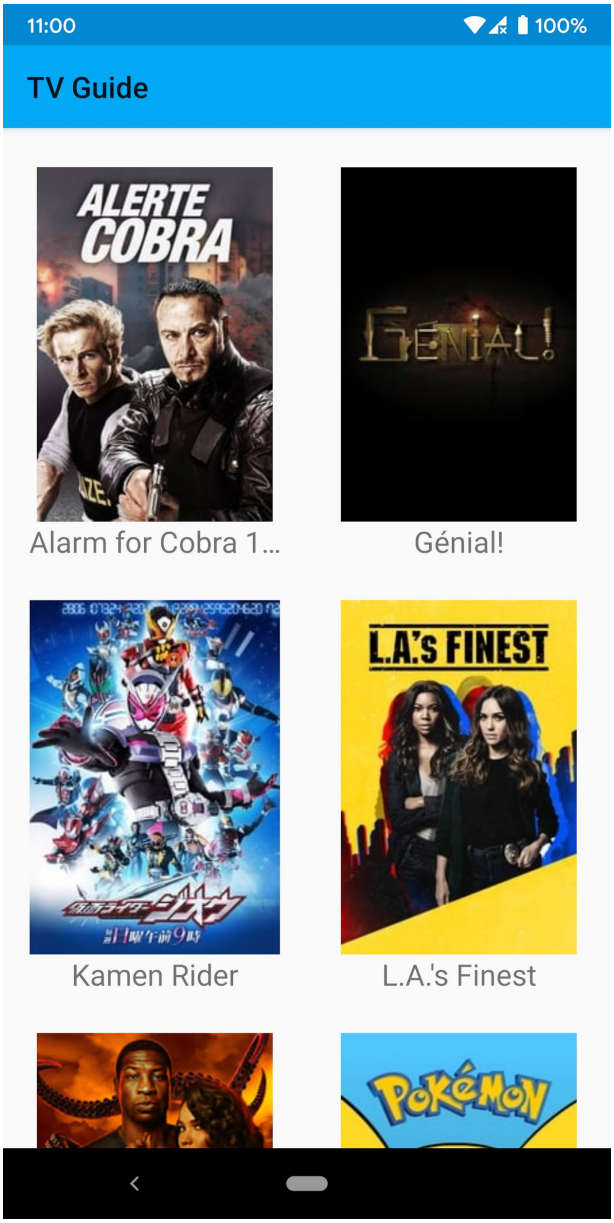

Figure 14.8: The main screen of the TV Guide app with the list of TV shows on offer

18. When you click on a TV show, the details screen will be displayed:

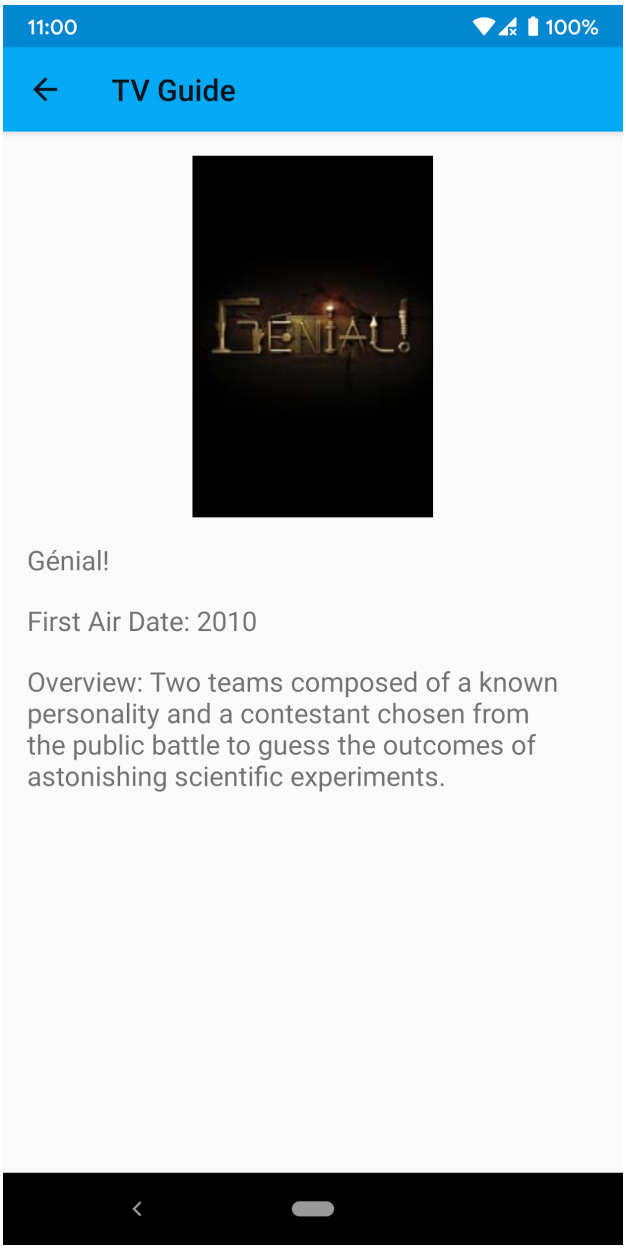

Figure 14.9: The details screen showing more information about the chosen TV show

You have cached the list of TV shows in the local database. In the next steps, you will be adding **`WorkManager`** to schedule a task for fetching data from the server and saving it to the local database at regular intervals.

19. Open **`TVShowRepository`** and add a suspending function for fetching TV shows from the network and saving them to the database:

```
    suspend fun fetchTVShowsFromNetwork() {
        val tvDao: TVDao = tvDatabase.tvDao()
        var shows = tvDao.getTVShows()
        if (shows.isEmpty()) {
            try {
                val tvResponse = tvService.getTVShows(apiKey)
                shows = tvResponse.results
                tvDao.addTVShows(shows)
            } catch (exception: Exception) {
                errorLiveData.postValue("An error occurred:
                ${exception.message}")
            }
        }
    }
```

This will be the function that will be called by the **`Worker`** class that will be running to fetch and save the TV shows.

20. Create the **`TVShowWorker`** class:

```
class TVShowWorker(private val context: Context, params:
  WorkerParameters) : Worker(context, params) {
    override fun doWork(): Result {
        val tvShowRepository = (context as
          TVApplication).tvShowRepository
        CoroutineScope(Dispatchers.IO).launch {
            tvShowRepository.fetchTVShowsFromNetwork()
        }
        return Result.success()
    }
}
```

21. Open **`TVApplication`** and at the end of the **`onCreate`** function, schedule **`TVShowWorker`** to retrieve and save the shows:

```
override fun onCreate() {
    ...

    val constraints =
        Constraints.Builder().setRequiredNetworkType(NetworkType
          .CONNECTED).build()
    val workRequest = PeriodicWorkRequest
        .Builder(TVShowWorker::class.java, 1, TimeUnit.HOURS)
        .setConstraints(constraints)
        .addTag("tvshow-work")
        .build()
    WorkManager.getInstance(applicationContext).enqueue(workRequest)
}
```

    This schedules **`TVShowWorker`** to run every hour when the device is connected to the network.

22. Run your application. It will display the list of TV shows. Now, the list of TV shows will be fetched and saved at scheduled intervals, even when the app is closed.

CHAPTER 15: ANIMATIONS AND TRANSITIONS WITH COORDINATORLAYOUT AND MOTIONLAYOUT

ACTIVITY 15.01: PASSWORD GENERATOR

Solution:

Here is one way we can create the *Password Generator* app:

1. Create a new project in Android Studio 4.0 or higher with **`Password Generator`** as the name. Set its package name to **`com.example.passwordgenerator`** and **`Minimum SDK`** to **`API 21: Android 5.0 Lollipop`**.
2. Add the **`MaterialComponents`** dependency to your **`app/build.gradle`** file:

```
implementation 'com.google.android.material:material:1.2.1'
```

   We will add this so that we can use **`TextInputLayout`** and **`TextInputEditText`** for the input text field for the password length.

3. Make sure that the dependency for **`ConstraintLayout`** is version 2 or above, for example:

```
implementation
  'androidx.constraintlayout:constraintlayout:2.0.4'
```

   This will allow us to use **`MotionLayout`** in our layout files.

4. Open the **`themes.xml`** file and make sure that the activity's theme is using a theme from **`MaterialComponents`**. See the following example:

```
<style name="AppTheme"
  parent="Theme.MaterialComponents.Light.DarkActionBar">
```

   We need to do this as the **`TextInputLayout`** and **`TextInputEditText`** we will be using later require your activity to use a **`MaterialComponents`** theme.

5. Open **`activity_main.xml`**. Remove the **`Hello World TextView`** and add the input text field for the length:

```
<com.google.android.material.textfield.TextInputLayout
        android:id="@+id/length_text_layout"
        style="@style/Widget.MaterialComponents
          .TextInputLayout.OutlinedBox"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:hint="Password Length (6-20)"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent">

        <com.google.android.material.textfield
          .TextInputEditText
            android:id="@+id/length_text"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:inputType="number"
            android:maxLength="2"
            android:textSize="18sp" />
    </com.google.android.material.textfield.TextInputLayout>
```

6. Add the checkboxes for uppercase, numbers, and special characters below the length text field layout:

```
    <CheckBox
        android:id="@+id/uppercase_check"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:text="Add Uppercase Characters"
        app:layout_constraintTop_toBottomOf
          ="@id/length_text_layout" />

    <CheckBox
        android:id="@+id/number_check"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:text="Add Numbers"
        app:layout_constraintTop_toBottomOf
          ="@id/uppercase_check" />

    <CheckBox
        android:id="@+id/special_check"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
```

```
        android:text="Add Special Characters"
        app:layout_constraintTop_toBottomOf
          ="@id/number_check" />
```

7. Add the **`Generate Password`** button at the bottom of the checkboxes:

```
    <Button
        android:id="@+id/generate_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:text="Generate Password"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf
          ="@id/special_check" />
```

8. Create another activity. Go to the **`File`** menu and click on **`New`** | **`Activity`** | **`Empty Activity`**. Name it **`OutputActivity`**. Make sure **`Generate Layout File`** is checked so that **`activity_output`** will be created.

9. Let's customize the activity transition from the input screen (**`MainActivity`**) to **`OutputActivity`**. Open **`themes.xml`** and update the activity theme with the **`windowActivityTransitions`**, **`windowEnterTransition`**, and **`windowExitTransition`** style attributes:

```
        <item name="android:windowActivityTransitions">
          true</item>
        <item name="android:windowEnterTransition">
          @android:transition/slide_right</item>
        <item name="android:windowExitTransition">
          @android:transition/fade</item>
```

   This will enable the activity transition, add a slide right enter transition, and add an exit transition to fade on the activity.

10. Open **`MainActivity`**. At the end of the **`onCreate`** function, add the following code:

```
        val lengthText: EditText =
          findViewById(R.id.length_text)
        val uppercaseCheckbox: CheckBox =
          findViewById(R.id.uppercase_check)
        val numberCheckbox: CheckBox =
          findViewById(R.id.number_check)
        val specialCheckbox: CheckBox =
          findViewById(R.id.special_check)
        val button: Button =
          findViewById(R.id.generate_button)
        button.setOnClickListener {
```

```
            val length = length_text.text.toString().toInt()
            if (length < 6 || length > 20) {
                Snackbar.make(it, "Length must be from 6
                  to 20", Snackbar.LENGTH_SHORT).show()
                return@setOnClickListener
            }

            val intent = Intent(this,
              OutputActivity::class.java).apply {
                putExtra("length", lengthText.text.toString())
                putExtra("uppercase",
                  uppercaseCheck.isChecked)
                putExtra("numbers", numberCheck.isChecked)
                putExtra("special", speciaCheck.isChecked)
            }
            startActivity(intent, ActivityOptions
              .makeSceneTransitionAnimation(this).toBundle())
        }
```

This will add a **`ClickListener`** component on the **`Generate`** button. When it's tapped, the system will open **`OutputActivity`** and pass the length, uppercase, number, and special character values as intent extras.

11. Open the **`activity_output.xml`** file and change **`androidx.constraintlayout.widget.ConstraintLayout`** to the following:

```
androidx.constraintlayout.motion.widget.MotionLayout
```

This will allow us to use **`MotionLayout`** for the output screen.

12. Add **`app:layoutDescription="@xml/motion_scene"`** and **`app:motionDebug="SHOW_ALL"`** to the **`MotionLayout`** tag. The first will set the **`res/xml/motion_scene.xml`** file as **`motion_scene`** for **`MotionLayout`**. The latter will allow us to see the animation path and progress in the editor and on the device. **`activity_output`** will now look like the following:

```
<androidx.constraintlayout.motion.widget.MotionLayout
    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"
    app:layoutDescription="@xml/motion_scene"
    app:motionDebug="SHOW_ALL"
```

```
        tools:context=".OutputActivity">
        ...
    </androidx.constraintlayout.motion.widget.MotionLayout>
```

13. Add three instances of **`TextView`** to the output activity for the three passwords generated:

```
        <TextView
            android:id="@+id/password1_text"
            style="@style/TextAppearance.AppCompat.Headline"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginBottom="60dp"
            app:layout_constraintBottom_toTopOf
              ="@id/password2_text"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            tools:text="First Password" />

        <TextView
            android:id="@+id/password2_text"
            style="@style/TextAppearance.AppCompat.Headline"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            tools:text="Second Password" />

        <TextView
            android:id="@+id/password3_text"
            style="@style/TextAppearance.AppCompat.Headline"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="60dp"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf
              ="@+id/password2_text"
            tools:text="Third Password" />
```

14. Add a **`Copy`** button at the bottom of the screen:

```
<Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="60dp"
        android:text="Copy"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent" />
```

15. Open **`OutputActivity`**. Add the following function:

```
    private fun generatePassword(
        length: Int,
        addUpperCase: Boolean,
        addNumbers: Boolean,
        addSpecial: Boolean
    ): String {
        val password = mutableListOf<Char>()

        val lowercaseCharacters = ('a'..'z').toList()
        val upperCaseCharacters
          = lowercaseCharacters.map { it.toUpperCase() }
        val numbers = ('0'..'9').toList()
        val specialCharacters = "~!@#$%^&*()_+-="

        val characters = mutableListOf<Char>()
        characters.addAll(lowercaseCharacters)
        if (addUpperCase) {
            characters.addAll(upperCaseCharacters)
            password.add(upperCaseCharacters.random())
        }

        if (addNumbers) {
            characters.addAll(numbers)
            password.add(numbers.random())
        }

        if (addSpecial) {
            val specials
```

```
                = specialCharacters.toCharArray().toList()
            characters.addAll(specials)
            password.add(specials.random())
        }

        while (password.size < length) {
            password.add(characters.random())
        }

        password.shuffle()

        return password.joinToString("")
    }
```

This will generate the password depending on the input provided by the user.

16. At the end of the **`onCreate`** function, add the following code:

```
        val length: Int = intent
          ?.getStringExtra("length")?.toInt() ?: 0
        val upperCase: Boolean = intent
          ?.getBooleanExtra("uppercase", false) ?: false
        val numbers: Boolean = intent
          ?.getBooleanExtra("numbers", false) ?: false
        val special: Boolean = intent
          ?.getBooleanExtra("special", false) ?: false
        val password1: TextView =
          findViewById(R.id.password1_text)
        password1.text = generatePassword(
            length = length,
            addUpperCase = upperCase,
            addNumbers = numbers,
            addSpecial = special
        )
        val password2: TextView = findViewById
          (R.id.password2_text)
        password2.text = generatePassword(
            length = length,
            addUpperCase = upperCase,
            addNumbers = numbers,
            addSpecial = special
        )
        val password3: TextView = findViewById
          (R.id.password3_text)
        password3.text = generatePassword(
```

```
            length = length,
            addUpperCase = upperCase,
            addNumbers = numbers,
            addSpecial = special
        )
        val button: Button = findViewById(R.id.button)
        button.setOnClickListener {
            val clipboard = getSystemService
              (Context.CLIPBOARD_SERVICE) as ClipboardManager

            val password = when {
                password1.isVisible -> {
                        password1.text.toString()
                }
                password2.isVisible -> {
                        password2.text.toString()
                }
                else -> {
                    password3.text.toString()
                }
            }

            val clip: ClipData = ClipData
              .newPlainText("password", password)
            clipboard.setPrimaryClip(clip)

            Snackbar.make(it, "Password has been copied!",
              Snackbar.LENGTH_SHORT).show()
        }
```

This will generate the three passwords based on the user input and add a **`ClickListener`** component to the **`Copy`** button for the user to copy the selected password to the clipboard.

17. In **`OutputActivity`**, we will be creating an animation per password **`TextView`**. When the user has selected one, we'll be moving the selected password to the center and hiding the others. We'll also show the **`Copy`** button.

18. We'll first create **`ConstraintSet`** for the default view in the **`motion_scene.xml`** file:

```
    <ConstraintSet android:id="@+id/passwords_start">
        <Constraint
            android:id="@id/password2_text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
        <Constraint
            android:id="@id/password1_text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginBottom="60dp"
            app:layout_constraintBottom_toTopOf
                ="@id/password2_text"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent" />
        <Constraint
            android:id="@id/password3_text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="60dp"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf
              ="@+id/password2_text" />
        <Constraint
            android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginBottom="60dp"
            android:visibility="gone"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            tools:visibility="visible" />
     </ConstraintSet>
```

19. We'll then add the first **`ConstraintSet`** for when the first password is selected:

```
        <ConstraintSet android:id="@+id/password1_end">
          <Constraint
              android:id="@id/password1_text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              app:layout_constraintBottom_toBottomOf="parent"
              app:layout_constraintEnd_toEndOf="parent"
              app:layout_constraintStart_toStartOf="parent"
              app:layout_constraintTop_toTopOf="parent" />
          <Constraint
              android:id="@id/password2_text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:visibility="invisible"
              app:layout_constraintBottom_toBottomOf="parent"
              app:layout_constraintStart_toStartOf="parent"
              app:layout_constraintTop_toTopOf="parent"
              tools:visibility="visible" />
          <Constraint
              android:id="@id/password3_text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:visibility="invisible"
              app:layout_constraintBottom_toBottomOf="parent"
              app:layout_constraintEnd_toEndOf="parent"
              tools:visibility="visible" />
          <Constraint
              android:id="@+id/button"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:layout_marginBottom="60dp"
              android:visibility="visible"
              app:layout_constraintBottom_toBottomOf="parent"
              app:layout_constraintEnd_toEndOf="parent"
              app:layout_constraintStart_toStartOf="parent" />
      </ConstraintSet>
```

20. Next, let's add **`ConstraintSet`** for when the second password is selected:

```
        <ConstraintSet android:id="@+id/password2_end">
          <Constraint
              android:id="@id/password1_text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:visibility="invisible"
              app:layout_constraintStart_toStartOf="parent"
              app:layout_constraintTop_toTopOf="parent"
              tools:visibility="visible" />
          <Constraint
              android:id="@id/password3_text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:visibility="invisible"
              app:layout_constraintBottom_toBottomOf="parent"
              app:layout_constraintEnd_toEndOf="parent"
              tools:visibility="visible" />
          <Constraint
              android:id="@+id/button"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:layout_marginBottom="60dp"
              android:visibility="visible"
              app:layout_constraintBottom_toBottomOf="parent"
              app:layout_constraintEnd_toEndOf="parent"
              app:layout_constraintStart_toStartOf="parent" />
       </ConstraintSet>
```

21. Next, let's add **`ConstraintSet`** for when the third password is selected:

```
      <ConstraintSet android:id="@+id/password3_end">
          <Constraint
              android:id="@id/password3_text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              app:layout_constraintBottom_toBottomOf="parent"
              app:layout_constraintEnd_toEndOf="parent"
              app:layout_constraintStart_toStartOf="parent"
              app:layout_constraintTop_toTopOf="parent" />
```

```
        <Constraint
            android:id="@id/password1_text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:visibility="invisible"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            tools:visibility="visible" />
        <Constraint
            android:id="@id/password2_text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:visibility="invisible"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            tools:visibility="visible" />
        <Constraint
            android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginBottom="60dp"
            android:visibility="visible"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent" />
    </ConstraintSet>
```

22. Finally, add **`Transition`** for when each password is selected:

```
    <Transition
        app:constraintSetEnd="@id/password2_end"
        app:constraintSetStart="@id/passwords_start"
        app:duration="2000">
        <OnClick
```

```
            app:clickAction="transitionToEnd"
            app:targetId="@id/password2_text" />
    </Transition>

    <Transition
        app:constraintSetEnd="@id/password1_end"
        app:constraintSetStart="@id/passwords_start"
        app:duration="2000">
        <OnClick
            app:clickAction="transitionToEnd"
            app:targetId="@id/password1_text" />
    </Transition>

    <Transition
        app:constraintSetEnd="@id/password3_end"
        app:constraintSetStart="@id/passwords_start"
        app:duration="2000">
        <OnClick
            app:clickAction="transitionToEnd"
            app:targetId="@id/password3_text" />
    </Transition>
```

23. Run the application by going to the **Run** menu and clicking the **Run app** menu item. Note the transition between the input and output screen. When the Android UI is opening **OutputActivity**, you will notice that the views are sliding right, and while closing, the views fade out.

24. Input a length, select uppercase, numbers, and special characters, and tap on the **Generate** button. Three passwords will be displayed.

25. Select one and the rest will move out of view. A **Copy** button will also be displayed. Click it and check whether the password you selected is now on the clipboard. The initial and final state of the output screen will be similar to *Figure 15.22*:

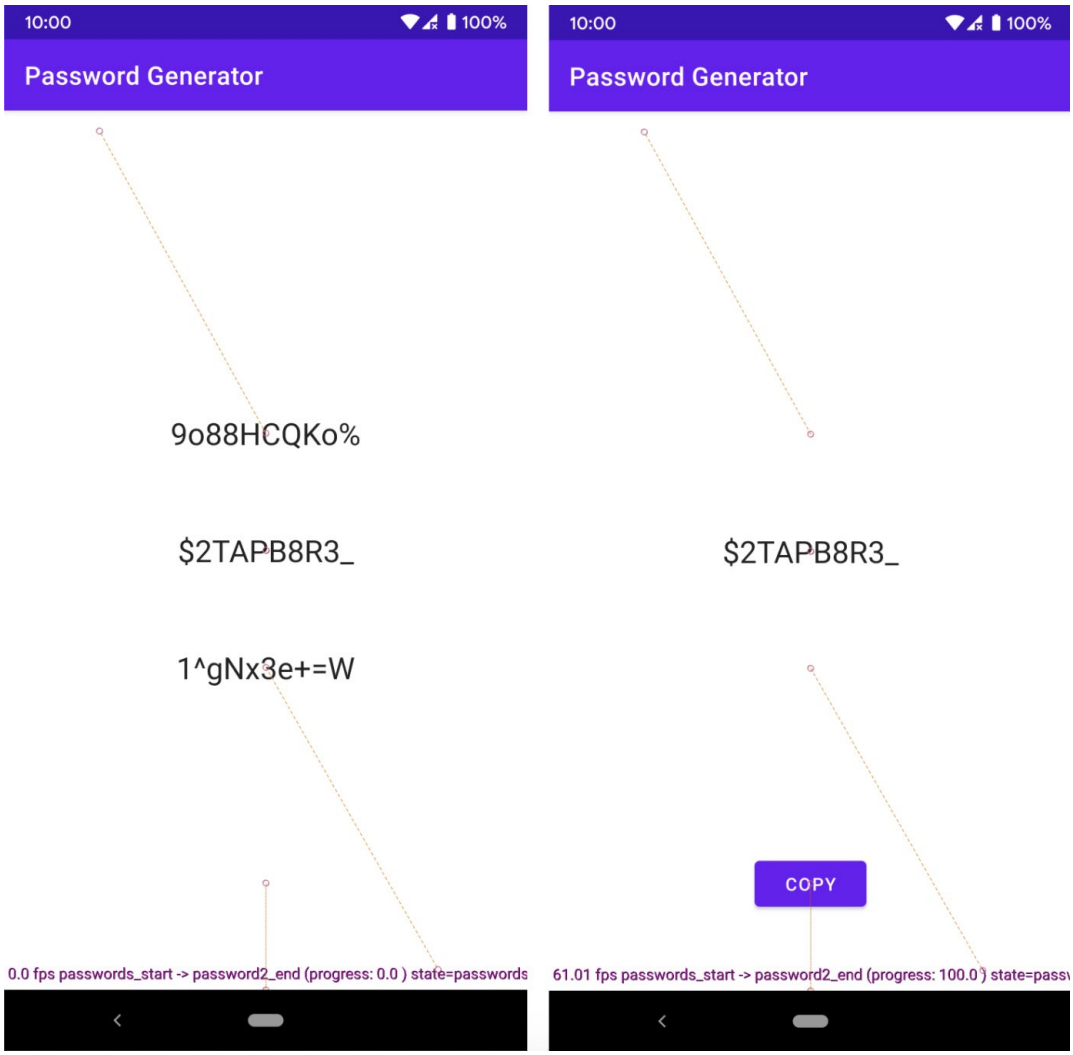

Figure 15.22: The start and end state of MotionLayout in the Password Generator app