Android App

Manage Passwords (MaPass)

Introduction

In this project, our aim was to create a password management application for Android called MaPass. In today's digital world, the need for strong passwords is essential to protect our personal and sensitive information. However, managing these passwords can be quite challenging. MaPass is an innovative password management application that we have developed specifically for Android, combining security and convenience into one functional tool. With MaPass, you can securely store and organize all your passwords in one place, ensuring easy access whenever you need them.

The password manager application is a valuable tool for simplifying and securing your daily login process in various platforms and services. In an era where the number of personal accounts our is constantly growing, remembering passwords becomes more and more complex and error-prone. The password management application is a central and secure repository for all the passwords we use. You can save them your passwords for websites, apps, email, social networks and other services with convenience and safety. This eliminates the need to remember many passwords or to themyou are writing to unsafe places.

In addition, the application offers advanced security features. The codes access are stored with encryption, ensuring their confidentiality your information. With the help of this application, you have the ability to focus on safety and convenience. You no longer have to worry about her losing passwords, as you can access them with a safe way and with minimal effort. Enjoy the feeling of convenience and security as you manage your passwords with this great app.

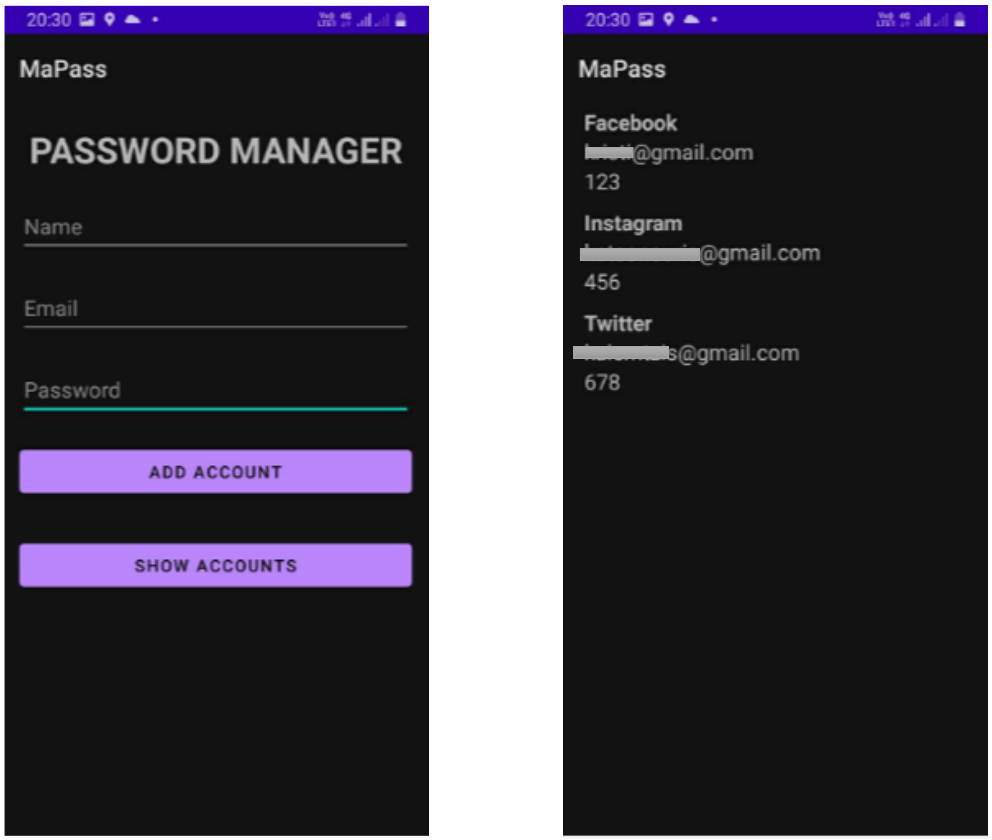


Image 1: Example of app display on android

To implement the application, the following classes were created in the empty activity of android studio (Hello World):

- Accounts
- EncryptionUtils
- FetchDataActivity
- LanguageActivity
- MainActivity
- MyAdapter
- MyDatabase
- MyInterface
- UpdataActivity

*These classes are located in the path (app/java/com/example/mapass)

Class analysis

Accounts

The Accounts class represents a user account.

By @Entity we mean that this class will internally map to one table in a database named tableName = “accounts”.

```
KristiCami
@Entity(tableName = "accounts")
```

This table contains four attributes (id, sName, email, password), with the @PrimaryKey to be automatically generated via the autoGenerate parameter.

```
4 usages
@PrimaryKey(autoGenerate = true)
private int id;
```

In the class we will also find a constructor and its two **setters, getters** methods they are necessary to link the code with other sections.

FetchDataActivity

The code presents a simple functionality for displaying and viewing data from a database.

Introduction of necessary libraries and classes for application development.

Define the required variables for the application.

Implementation of the **onCreate()** method, which is executed when the activity starts.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    binding = ActivityFetchDataActivityBinding.inflate(getLayoutInflater());
    setContentView(binding.getRoot());

    myDb = Room.databaseBuilder( context: this,MyDatabase.class, name: "accounts").allowMainThreadQueries().build();
    myInt=myDb.myInterface();

    accountsList=myInt.getAllAccounts();
    Toast.makeText( context: this, text: "You have "+accountsList.size() + " accounts",Toast.LENGTH_SHORT ).show();
    myAdapter=new MyAdapter( context: this,accountsList);
    binding.dataRecycler.setLayoutManager(new LinearLayoutManager( context: this));
    binding.dataRecycler.setAdapter(myAdapter);

}
```

Attach the layout file (**ActivityFetchDataActivityBinding**) to the activity.

Creating a database (**myDb**) using **Room**, which allows the data storage and retrieval.

Retrieve a linked database **myInt** object.

Get a list (**accountsList**) containing objects of type **Accounts** from the database data using **myInt**.

```
4 usages
ActivityFetchDataActivityBinding binding;
2 usages
MyDatabase myDb;
2 usages
MyInterface myInt;
3 usages
List<Accounts> accountsList;
2 usages
MyAdapter myAdapter;
```

Display a notification (**Toast**) informing the user of the number of accounts present in the **accountsList**.

```
Toast.makeText( context: this, text: "You have "+accountsList.size() + " accounts",Toast.LENGTH_SHORT ).show();
```

Create a **myAdapter** object of type **MyAdapter** to view the data from the **accountsList**.

Define the layout (**LinearLayoutManager**) of the recycler (**RecyclerView**) **dataRecycler** component used to display the data on the screen.

Define the adapter (**myAdapter**) for the **dataRecycler** to display the list data on the screen.

MainActivity

This class is responsible for the initial display of the screen and its management events.

Some important functions and variables used in the class **MainActivity** are as follows:

ActivityMainBinding binding: A variable referenced in the binding file(binding) automatically generated by Android Studio, which binds the components of the user interface in the XML file with the Java class.

MyDatabase myDb: An object of the **MyDatabase** class, representing the SQLite database. The database is created with the help of the library Room and is used to store the accounts.

MyInterface myInt: An object of the **MyInterface** interface, providing methods for the interaction with the database.

```
7 usages
ActivityMainBinding binding;
2 usages
MyDatabase myDb;
2 usages
MyInterface myInt;
```

In the **onCreate** method, which is called when the activity is started, the following actions:

Initialize the **binding** variable using the method **ActivityMainBinding.inflate(getLayoutInflater())**. This creates an object binding object that binds the user interface elements to the Java class.

Define the activity interface using the method **setContentView(binding.getRoot**()). This determines which layout file (layout file) will used to display the activity screen.

Create the SQLite database using the **databaseBuilder** method. THE database is called "accounts" and the parameter **allowMainThreadQueries()** allows database queries to be executed on the main execution thread (UI thread). The **fallbackToDestructiveMigration**() parameter is used to automatic deletion and re-creation of the database in case of changes in the base version.

Assign the database interface object to the variable **myInt** with the assign **myDb.myInterface().**

```
super.onCreate(savedInstanceState);
binding = ActivityMainBinding.inflate(getLayoutInflater());
setContentView(binding.getRoot());

myDb= Room.databaseBuilder( context: this,MyDatabase.class, name: "accounts").allowMainThreadQueries().fallbackToDestructiveMigration().build();
myInt=myDb.myInterface();

KristiCami
```

Define the click action for the **addAcc** button. When the button is clicked, thevalues of the **socialMediaName**, **mobileNumber** and **password** fields. Subsequently, an Accounts object is created with these values and inserted into the basedata via the **insert** method of the **myInt** interface. Finally, a message appearssuccess using the **makeText** and show method of the Toast class.

```
KristiCami
binding.addAcc.setOnClickListener(new View.OnClickListener() {
    KristiCami
    @Override
    public void onClick(View view) {
        String name = binding.socialMediaName.getText().toString();
        String email = binding.mobileNumber.getText().toString();
        String password = binding.password.getText().toString();

        Accounts accounts = new Accounts( id: 0,name,email,password);
        myInt.insert(accounts);
        Toast.makeText(getApplicationContext(),  text: "Account added successfully", Toast.LENGTH_SHORT).show();
    }
});
```

Set the click action for the showAccBtn button. When the button is clicked, it starts a new activity using the **startActivity** method. The new activity is the **FetchDataActivity**.

```
KristiCami
binding.showAccBtn.setOnClickListener(new View.OnClickListener() {
    KristiCami
    @Override
    public void onClick(View v) {
        startActivity(new Intent( packageContext: MainActivity.this, FetchDataActivity.class));
    }
});
```

The **MainActivity** class represents the home screen of the application and includes the logic for saving new accounts and viewing existing ones accounts.

MyAdapter

The **MyAdapter** class represents an adapter for the display data in a RecyclerView. This adapter binds the list data accounts (accountsList) with the GUI elements shown in the RecyclerView objects.

The important functions and variables used in MyAdapter class are the following:

Context context: The context of the adapter, which is used for the creating new activities and other actions in the application.

List<Accounts> accountsList: The list containing the **Accounts** objects that will appear in the RecyclerView.

```
3 usages
private Context context;
3 usages
private List<Accounts> accountsList;
```

MyAdapter(Context context, List<Accounts> accountsList) constructor: Accepts it adapter environment and the list of accounts and stores them in corresponding variables.

```
1 usage   KristiCami
public MyAdapter(Context context, List<Accounts> accountsList) {
    this.accountsList = accountsList;
    this.context = context;
}
```

The **onCreateViewHolder()** method: Creates a **MyViewHolder** object thatrepresents the view for each item in the list. Uses the inflator to load the layout from the single_row XML file and return it.

```
KristiCami
@NonNull
@Override
public MyViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
    View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.single_row,parent, attachToRoot: false);
    return new MyViewHolder(view);
}
```

The **onBindViewHolder()** method: Binds the account data to the respective views (TextViews) of MyViewHolder . It also sets the click action for each item of the list so that when an item is clicked, a new activity starts **UpdateActivity** and pass the necessary data through the Intent.

```
KristiCami
@Override
public void onBindViewHolder(@NonNull MyViewHolder holder, int position) {

    Accounts accounts = accountsList.get(position);
    holder.name.setText(accounts.getsName());
    holder.email.setText(accounts.getEmail());
    holder.password.setText(accounts.getPassword());

    KristiCami
    holder.itemView.setOnClickListener(new View.OnClickListener() {
        KristiCami
        @Override
        public void onClick(View v) {
            Intent intent=new Intent(context,UpdateActivity.class);
            intent.putExtra( name: "name",accounts.getsName());
            intent.putExtra( name: "id",accounts.getId());

            intent.putExtra( name: "email",accounts.getEmail());
            intent.putExtra( name: "name",accounts.getPassword());
            context.startActivity(intent);
        }
    });
}
```

The **getItemCount()** method: Returns the number of items in the list.

```
KristiCami
@Override
public int getItemCount() { return accountsList.size(); }
```

The **MyViewHolder** class: It is an inner classifier that holds the view references (TextViews) for each item in the list. These reports are initialized in the class constructor.

```
4 usages  KristiCami
public class MyViewHolder extends RecyclerView.ViewHolder{
    2 usages
    TextView name,email,password;
    1 usage  KristiCami
    public MyViewHolder(@NonNull View itemView) {
        super(itemView);
        name = itemView.findViewById(R.id.sName_one);
        email=itemView.findViewById(R.id.email_one);
        password = itemView.findViewById(R.id.pass_one);

    }
}
```

MyDatabase

The **MyDatabase** class represents a database in the framework of the architecture Room in Android Studio. This class extends **RoomDatabase** and defines the structure of the database as well as the relationships between the tables.

The important functions and variables used in the **MyDatabase** class are the folowing:

The @Database tag: This tag is used to indicate that the class represents a Room database. It takes some parameters, such as array **entities**, which describes the entity classes associated with the database, and the **version,** which specifies the version number of the database.

The **myInterface()** method: This abstract method provides a MyInterface object which represents the database access interface.

```
import androidx.room.Database;
import androidx.room.RoomDatabase;

import com.example.mapass.MyInterface;
7 usages  1 inheritor  KristiCami
@Database(entities = {Accounts.class},version = 4)
public abstract class MyDatabase extends RoomDatabase {
    3 usages  1 implementation  KristiCami
    public abstract MyInterface myInterface();
}
```

The **MyDatabase** class is abstract, which means it cannot create an object directly from it. Instead, it is used for create a subclass that implements the Room database functionality and provides the necessary **myInterface()** method to access the database.

MyInterface

The **MyInterface** interface represents a Data Access Object in its context library Room. This interface contains methods used to access and manage data in the database.

The important methods defined in the **MyInterface** interface are as follows:

void insert(Accounts entityClass): This method is used to insert (insert) an Accounts object in the database.

void update(Accounts entityClass): This method is used to update (update) an Accounts object in the database.

void deleteAcc(int id): This method is used to delete (delete) an Accounts object from the database, based on the id field.

List<Accounts> getAllAccounts(): This method is used to retrieve (query) all Accounts objects from the database.

```
9 usages  1 implementation  KristiCami
@Dao
public interface MyInterface {
    1 usage  1 implementation  KristiCami
    @Insert
    void insert(Accounts entityClass);
    1 implementation  KristiCami
    @Update
    void update(Accounts entityClass);
    1 usage  1 implementation  KristiCami
    @Query("DELETE from accounts where id=:id")
    void deleteAcc(int id);
    1 usage  1 implementation  KristiCami
    @Query("SELECT * From accounts")
    List<Accounts> getAllAccounts();
}
```

The **MyInterface** interface corresponds to the functionality of queries that performed on the Room database, such as insert, update, delete, and data recovery.

UpdateActivity

The **UpdateActivity** class is used to update and delete an account in the database.

The important variables and functions used in the UpdateActivity class are the folowing:

The variable **binding**: Object of the **ActivityUpdateBinding** class that provides access the UI elements defined in its XML file activity.

The variables **myDb** and **myInt**: Objects of class **MyDatabase** and **MyInterface** respectively, used to access the database.

The variables **name, email, password** and **id**: Receive the values of the parameters that are passed from the previous activity through the Intent object.

```
myDb= Room.databaseBuilder( context: this,MyDatabase.class, name: "accounts").allowMainThreadQueries().build();
myInt=myDb.myInterface();

String name = getIntent().getStringExtra( name: "name");
String email = getIntent().getStringExtra( name: "email");
String password = getIntent().getStringExtra( name: "password");
int id = getIntent().getIntExtra( name: "id", defaultValue: -1);

binding.socialMediaName.setText(name);
binding.mobileNumber.setText(email);
binding.password.setText(password);
```

The **onCreate()** method: This method is called when the activity is started and executes the code to initialize the interface elements and create them event listeners for the buttons.

The event listener for the **updateAcc** button: This listener reacts to its click "Update" button and performs the update function of the base account data. An **Accounts** object is created with the new values and the **update()** method of the **myInt** object is called to perform the update. Also, one appears pop-up message indicating that the account has been successfully updated and activity terminates.

```
KristiCami
binding.updateAcc.setOnClickListener(new View.OnClickListener() {
    KristiCami
    @Override
    public void onClick(View v) {
        String name = binding.socialMediaName.getText().toString();
        String email = binding.mobileNumber.getText().toString();
        String pass = binding.password.getText().toString();

        Accounts accounts = new Accounts(id,name,email,pass);
        myInt.update(accounts);
        Toast.makeText(getApplicationContext(), text: "Account updated successfully", Toast.LENGTH_SHORT).show();
        finish();
    }
});
```

The event listener for the **delAcc** button: This listener reacts to its click "Delete" button and performs the function of deleting the account from the database data. The **deleteAcc()** method of the **myInt** object is called to perform the deletion. A pop-up message appears stating that the account deleted successfully and the activity terminates.

```
KristiCami
binding.delAcc.setOnClickListener(new View.OnClickListener() {
    KristiCami
    @Override
    public void onClick(View v) {
        myInt.deleteAcc(id);
        Toast.makeText(getApplicationContext(), text: "Account deleted successfully", Toast.LENGTH_SHORT).show();
        finish();
    }
});
```

EncryptionUtils

The EncryptionUtils class provides methods for encrypting passwords using the PBEWithMD5AndDES algorithm.

The important variables and methods used in the EncryptionUtils class are the folowing:

The constant variables **ENCRYPTION_ALGORITHM, ENCRYPTION_PASSWORD** and **SALT**: Define the encryption algorithm, the password for the encryption and the salt used to increase its security algorithm.

```
4 usages
private static final String ENCRYPTION_ALGORITHM = "PBEWithMD5AndDES";
2 usages
private static final String ENCRYPTION_PASSWORD = "j^D0bqKzV@y#4&8";
4 usages
private static final byte[] SALT = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08};
```

The **encrypt(String password)** method: Uses the PBEWithMD5AndDES algorithm to encrypt a password. Creates a **Cipher** and configures it for encryption mode with the secret key and the salt. The password is converted to a byte array and encrypted. Finally, it returns the encrypted result in Base64 format.

```
2 usages  Andreascy01
@RequiresApi(api = Build.VERSION_CODES.O)
public static String encrypt(String password) {
    try {
        SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(ENCRYPTION_ALGORITHM);
        KeySpec keySpec = new PBEKeySpec(ENCRYPTION_PASSWORD.toCharArray(), SALT, iterationCount: 65536, keyLength: 256);
        SecretKey secretKey = keyFactory.generateSecret(keySpec);

        Cipher cipher = Cipher.getInstance(ENCRYPTION_ALGORITHM);
        AlgorithmParameterSpec paramSpec = new PBEParameterSpec(SALT, iterationCount: 100);
        cipher.init(Cipher.ENCRYPT_MODE, secretKey, paramSpec);

        byte[] encryptedBytes = cipher.doFinal(password.getBytes( charsetName: "UTF-8"));
        return Base64.getEncoder().encodeToString(encryptedBytes);
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

The **decrypt(String encryptedPassword)** method: Uses the algorithm PBEWithMD5AndDES to decrypt an encrypted code access. Creates a **Cipher** and configures it for operation decryption with the secret key and the salt. The encrypted code access code is Base64 decoded and decrypted. Finally, he returns him decrypted password in String format.

```
4 usages  Andreascy01
@RequiresApi(api = Build.VERSION_CODES.O)
public static String decrypt(String encryptedPassword) {
    try {
        SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(ENCRYPTION_ALGORITHM);
        KeySpec keySpec = new PBEKeySpec(ENCRYPTION_PASSWORD.toCharArray(), SALT, iterationCount: 65536, keyLength: 256);
        SecretKey secretKey = keyFactory.generateSecret(keySpec);

        Cipher cipher = Cipher.getInstance(ENCRYPTION_ALGORITHM);
        AlgorithmParameterSpec paramSpec = new PBEParameterSpec(SALT, iterationCount: 100);
        cipher.init(Cipher.DECRYPT_MODE, secretKey, paramSpec);

        byte[] decodedBytes = Base64.getDecoder().decode(encryptedPassword);
        byte[] decryptedBytes = cipher.doFinal(decodedBytes);
        return new String(decryptedBytes, charsetName: "UTF-8");
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

The **EncryptionUtils** class is useful for securely storing and transporting critical

information, such as passwords, via plain text in an Android app.

LanguageManager

The **LanguageManager** class provides a method to update the settings language in an Android application.

The important variables and methods used in the **LanguageManager** class are the folowing:

The variable **ct**: Represents the Context to which the changes are applied language.

The **LanguageManager(Context context)** constructor: Accepts a **Context** object when creating the **LanguageManager** to initialize the **ct** variable.

```
// Retrieving the stored language preference
4 usages
private Context ct;

Andreascy01
public LanguageManager(Context context) { this.ct = context; }
```

The **updateResource(String code)** method: Takes a language code as input and updates the app's language settings. First, it creates a new object **Locale** based on language code. Then it changes the default locale language in the **Configuration** object and updates the **Context** with the new one local language. Finally, the application resources are updated with the new settings language.

```
Andreascy01
public void updateResource(String code)
{
    Locale locale = new Locale(code);
    Locale.setDefault(locale);
    Resources resources = ct.getResources();
    Configuration configuration = resources.getConfiguration();
    configuration.setLocale(locale);

    ct = ct.createConfigurationContext(configuration);
    resources.updateConfiguration(configuration, resources.getDisplayMetrics());
}
```

By using this class, application resources such as texts and symbols can be dynamically updated to match the selected language.

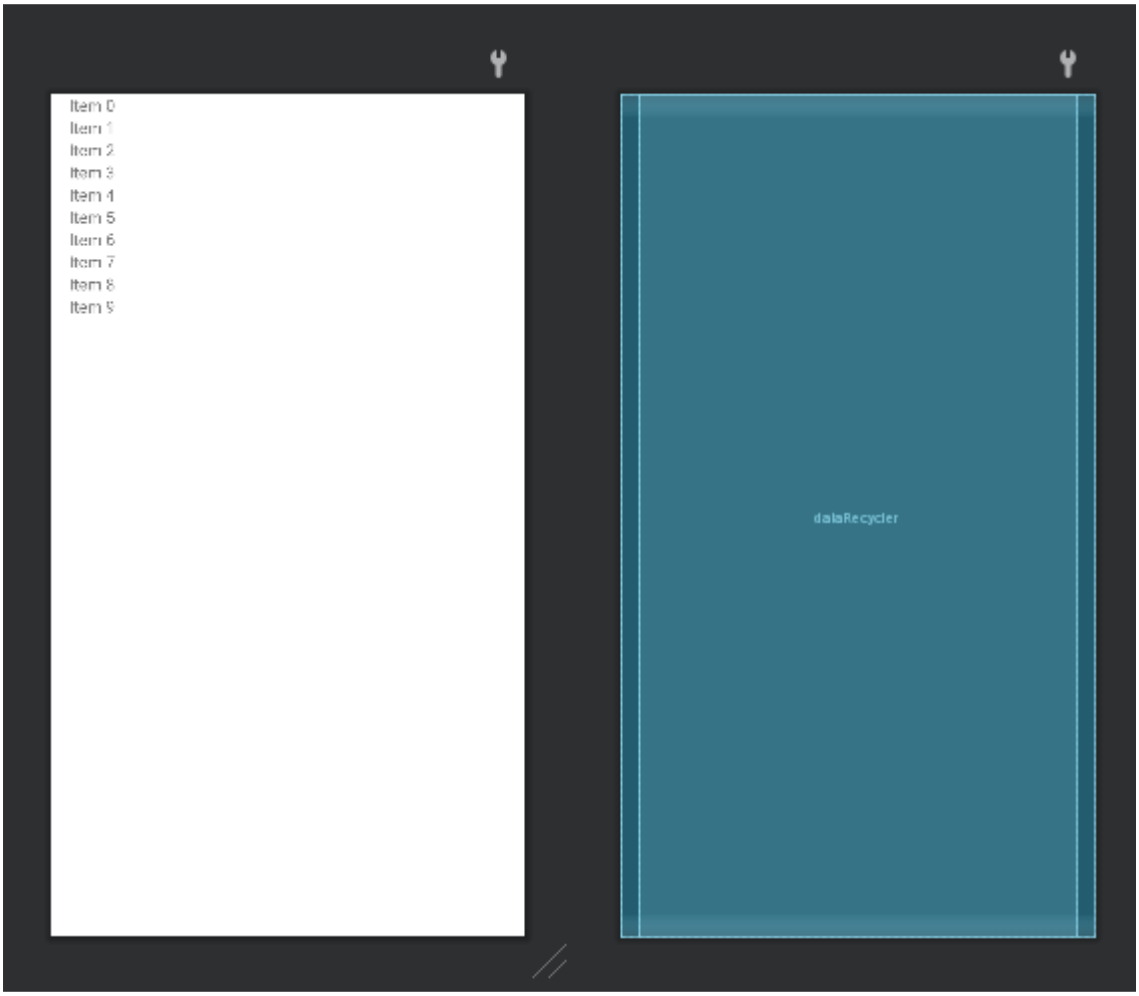
You also had to convince four xml files to be created:

- activity_fetch_data_activity.xml
- activity_main.xml
- activity_update.xml
- single_row.xml
- dialog_image.xml

These files are located in the path (app/res/layout)

Analysis of xml files

activity_fetch_data_activity.xml



The important functions and properties used in XML code are as follows:

The original **LinearLayout** tag: Defines a linear layout that places its elements of child vertically.

The additional attributes **xmlns:android, xmlns:app and xmlns:tools:** define them rectangular namespaces for Android libraries such as UI (android) components, customization features (app) and development tools (tools).

The **tools:context:** attribute defines the class to use as the context for the static template controller (template controller), in this case **MainActivity**.

```
tools:context=".MainActivity"
```

The **android:orientation:** attribute sets the orientation of the linear layout, in in this case vertical.

The **android:paddingLeft** and **android:paddingRight**: attributes define the indentation on the left and to the right of the linear layout, to add joints.

```
android:paddingLeft="16dp"
android:paddingRight="16dp">
```

The **<androidx.recyclerview.widget.RecyclerView>** element: Defines a RecyclerView, which is a scrolling area used to display a list of items. It is defined with the identifier **@+id/dataRecycler** and is set to occupies the same space as the parent linear layout.

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

activity_main.xml

The important functions and properties used in XML code are as follows:

The original **ScrollView** tag: Defines a ScrollView that allows content to be scrolled if it exceeds the limited space that is visible.

The additional attributes **xmlns:android, xmlns:app** and **xmlns:tools:** define them rectangular namespaces for Android libraries such as UI (android) components, customization features (app) and development tools (tools).

The **<LinearLayout>** element: Defines a linear layout that places its elements of child vertically.

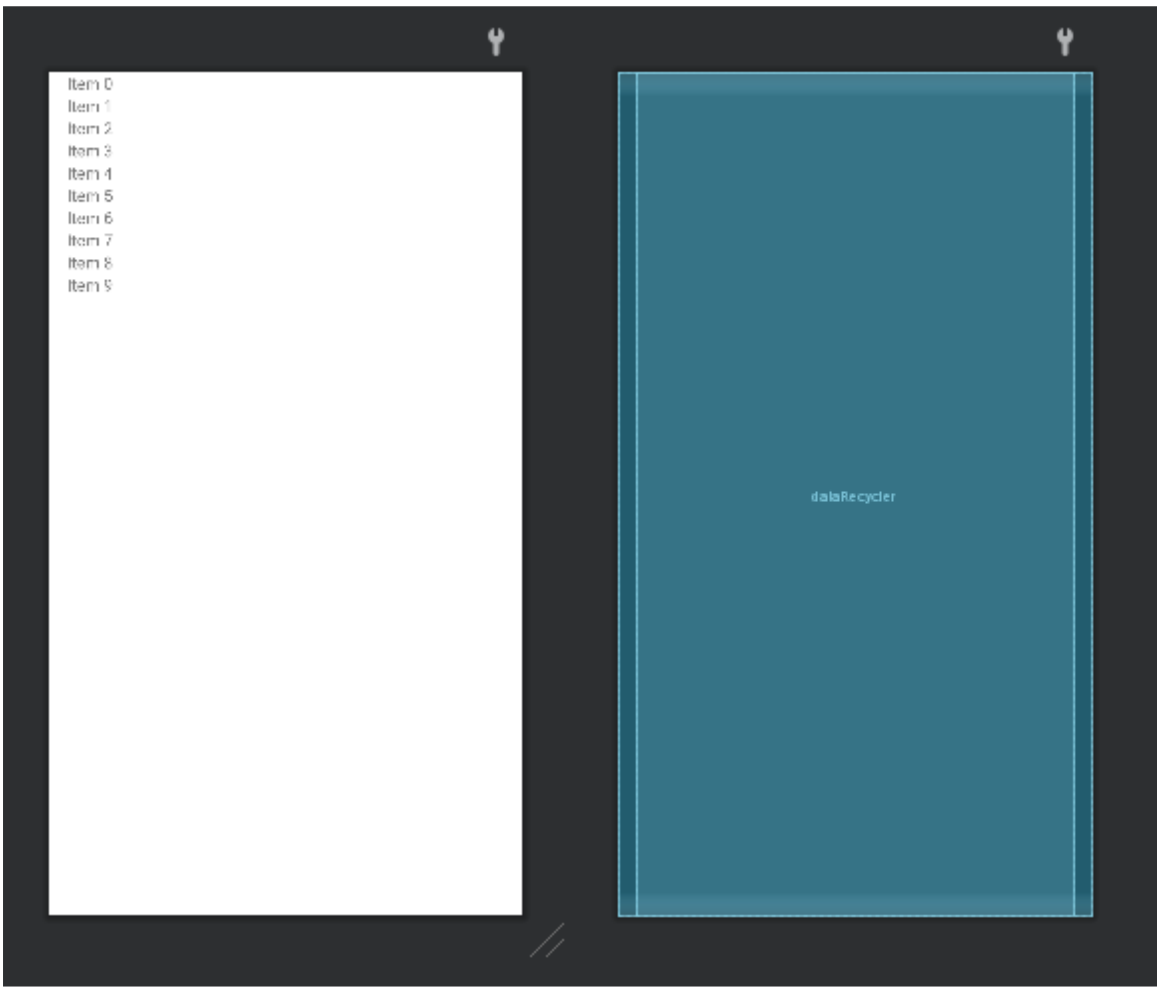
```
tools:context=".MainActivity"
android:orientation="vertical"
android:paddingLeft="16dp"
android:paddingRight="16dp"
android:layout_height="match_parent"
android:layout_width="match_parent">
```

The **tools:context:** attribute defines the class to use as the context for the static template controller (template controller), in this case **MainActivity.**

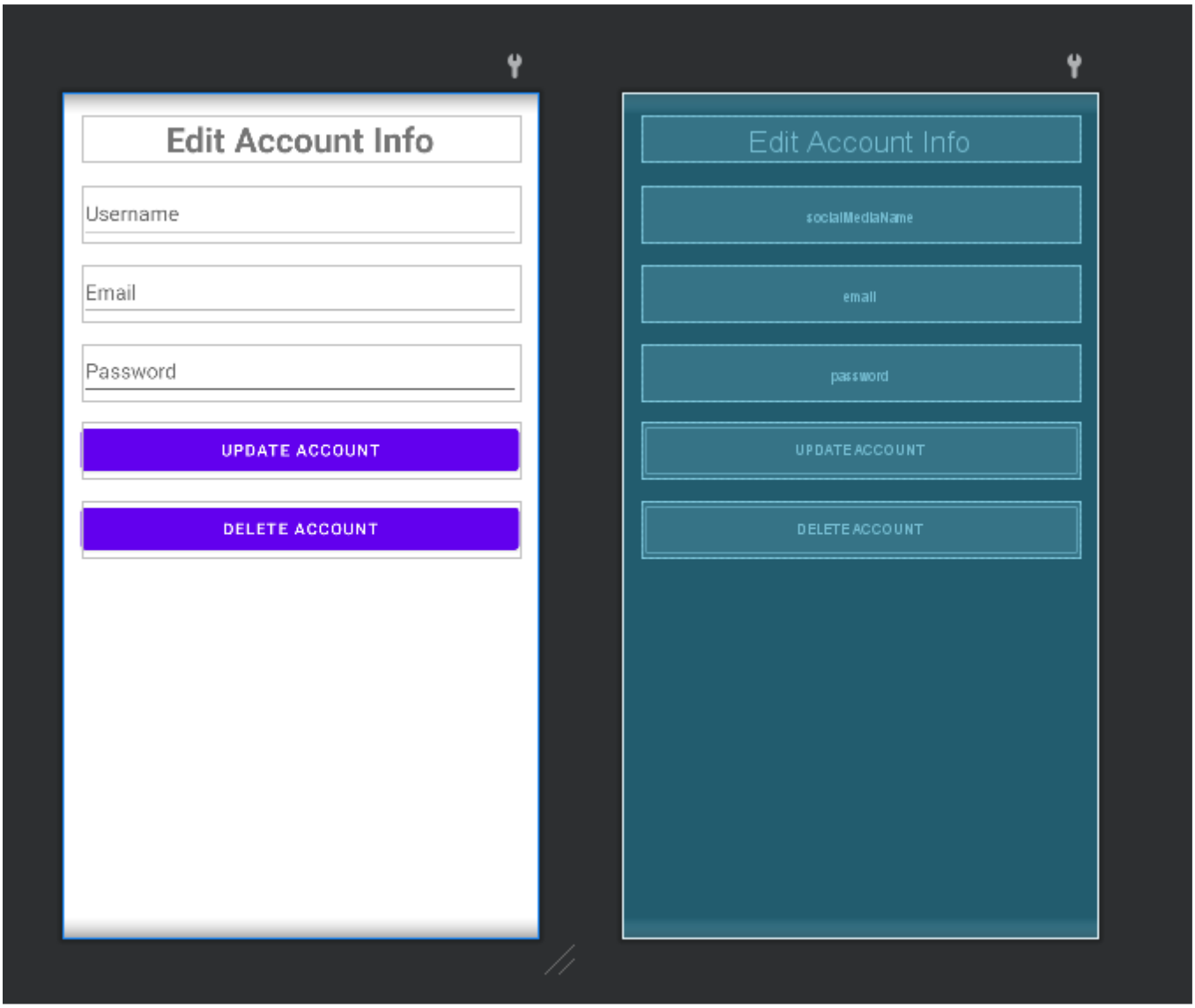
The attributes **android:layout_height** and b: Define the height and width of the linear layout as "match_parent", meaning it will take up all available space.

The **<TextView>, <EditText>** and **<Button>** elements: Define respectively a text, a text field and a button.

The attributes used in the above elements: Define various properties for each element, such as their size, alignment, and content.



activity_update.xml



Let's describe the main functions and properties used in it code:

The **xmlns:android, xmlns:app** and **xmlns:tools**: attributes define the rectangles namespaces for Android libraries such as UI (android) components, customization features (app) and development tools (tools).

The **<LinearLayout>** element: Defines a linear layout that places its elements of child vertically.

The **tools:context:** attribute defines the class to use as the context for the static template controller (template controller), in this case **MainActivity**.

The attributes **android:layout_height** and **android:layout_width:** Define the height and the width of the linear layout as "match_parent", which means it will take up the whole available space.

```
android:layout_width="match_parent"
android:layout_height="match_parent"
```

The **<TextView>, <EditText>** and **<Button>** elements: Define respectively a text, a text input field and a button.

```
<Button
    android:layout_marginTop="20dp"
    android:id="@+id/update_acc"
    android:layout_gravity="center_horizontal"
    android:text="Update Account"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

The attributes used in the above elements: Define various properties for every element, such as size, alignment, font type, and their content.

The above code describes a layout that contains a title, three input fields text and two buttons.

single_row.xml

Let's describe the main functions and properties used in it code:

The **xmlns:android:** attribute defines the rectangular namespace for its UI elements Android library.

The **<LinearLayout>** element: Defines a linear layout that places its elements of child vertically.

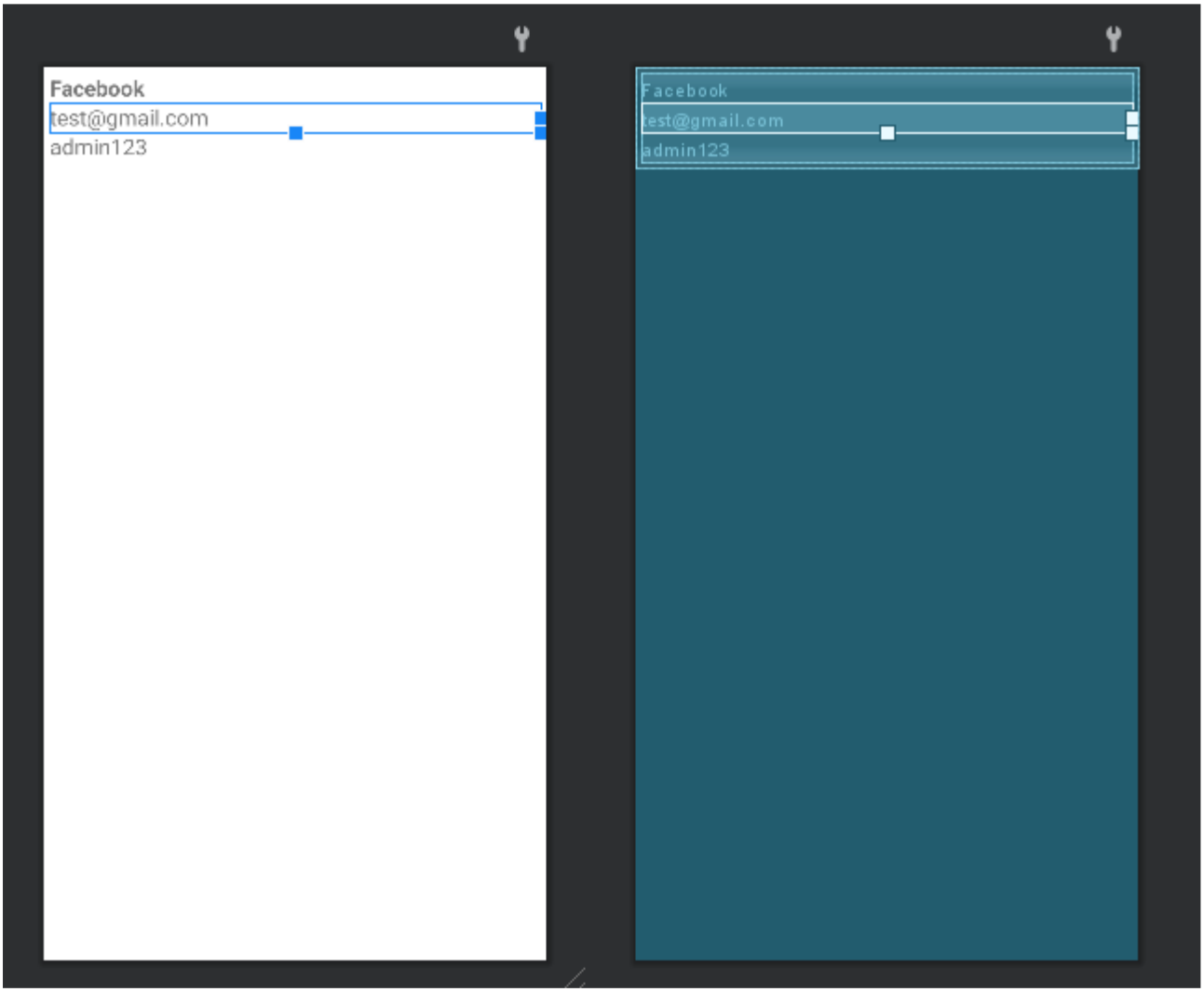
```
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:orientation="vertical"
android:padding="5dp">
```

The attributes **android:layout_height** and **android:layout_width:** Define its height linear layout as "wrap_content", meaning it will wrap around the content of, and its width as "match_parent", which means it will grab the available one space to its parent.

<TextView> elements: Define a text with various properties, such as size their font, style and content.

```
<TextView
    android:id="@+id/sName_one"
    android:textSize="18sp"
    android:textStyle="bold"
    android:text="Facebook"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

The code above describes a layout that contains three TextView elements. EachTextView contains information about a social network account, such as name account, email and password.



dialog_image.xml

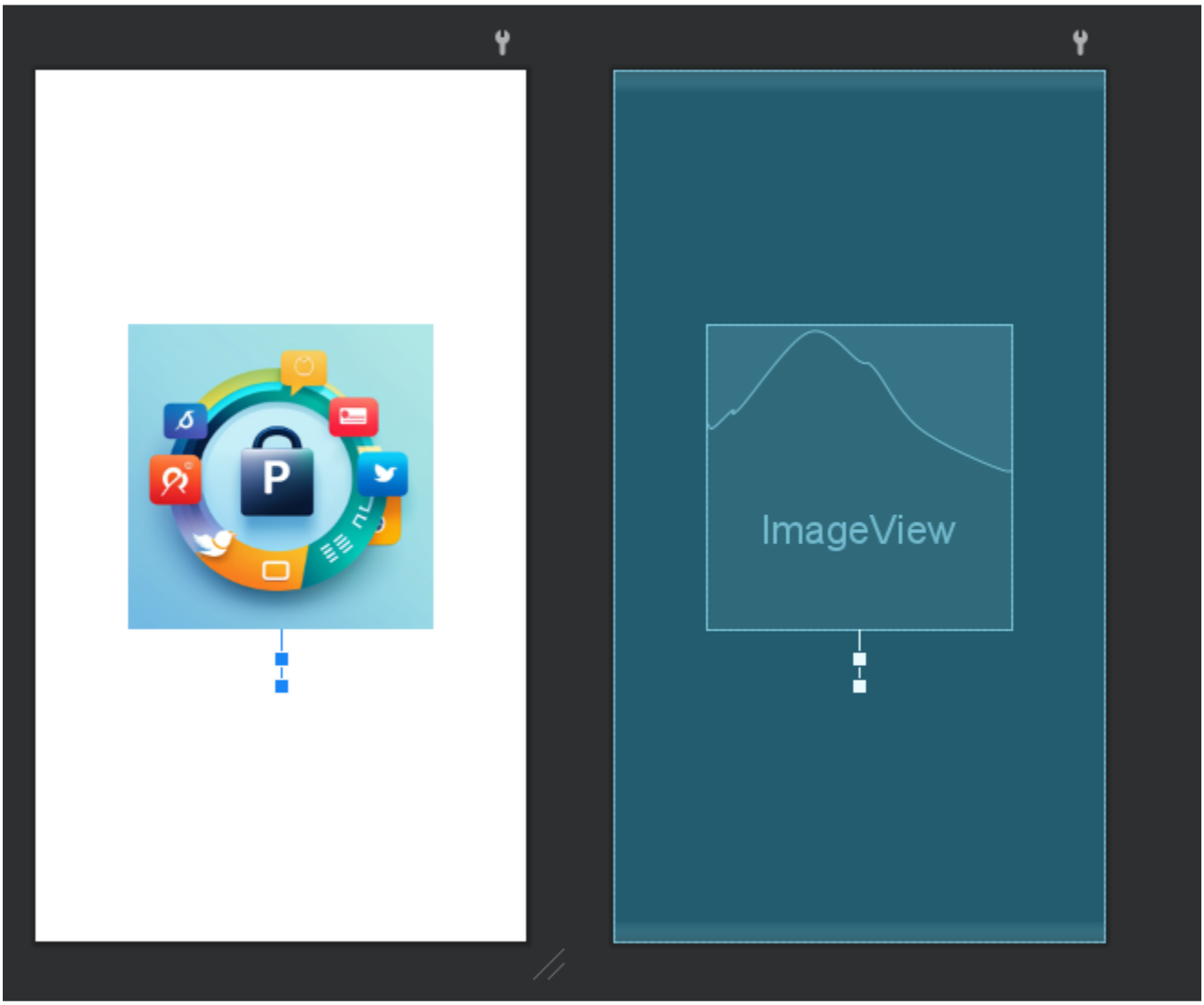
Let's describe the main functions and properties used in it code:

The **android:orientation:** attribute sets the orientation of the linear layout. In in this case, it is defined as "vertical", which means that the elements are placed vertically.

The **<ImageView>** element: Displays an image within the dialog. **android:src** defines the source of the image to be displayed. In this case, a file is used image named "applogo".

```
<ImageView
    android:id="@+id/dialogImage"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/applogo"/>
```

The **<TextView>** element: Displays a text within the dialog. **android:text** defines the text to be displayed. **android:textColor** sets the text color. The **android:textSize** sets the text size. **android:textStyle** defines its style text. In this case, the text will be adjusted to the center of its parent.



Room services were used as a database that store the data locally in the mobile memory. Specifically to achieve this we had to modify the build.gradle(app) file and add the below code in dependencies(Save data in a local database using Room | Android Developers, link: https://developer.android.com/training/data-storage/room)

```
//room
def room_version = "2.5.1"
implementation "androidx.room:room-runtime:$room_version"
annotationProcessor "androidx.room:room-compiler:$room_version"
```

For our convenience we have added the following for more direct access to classes and methods

```
buildFeatures { viewBinding true }
```