During application development, we need to use persistent storage from time to time to facilitate a better user experience. Today we will have a look at how we can use SQLite to do the same when developing Android Mobile applications.

SQLite is an Open Source database, it supports standard relational DB features like SQL syntax, transactions and prepared statement. The db is a lightweight component and doesn't use too much memory on the runtime. However, SQLite implementations require file system access which can slow down the application and it's recommended to perform SQLite operations asynchronously.

There are two ways to implement an SQLite DB in your applications. One to work directly with the SQLite DB, which requires manual conversion of database objects to Java objects and vice versa. The other is to use an SQL object mapping library like Room which provides a quick way to implement the SQLite in our applications by taking away the unnecessary effort and allowing more time to work on core application feature development as opposed to writing code to convert the objects to SQL objects back and forth.

We will utilize Room as SQL object mapping library for this demo, but you can choose to work directly with SQLite for your CAs if you wish to do so.

When working with Room, we will work with three major components:

- Database: Represents an abstract database class, which provides one or more data access objects (DAO)
- DAO: It is an interface that lets us define how to get or change values stored within a DB
- Entity: Represents a value object in the DB

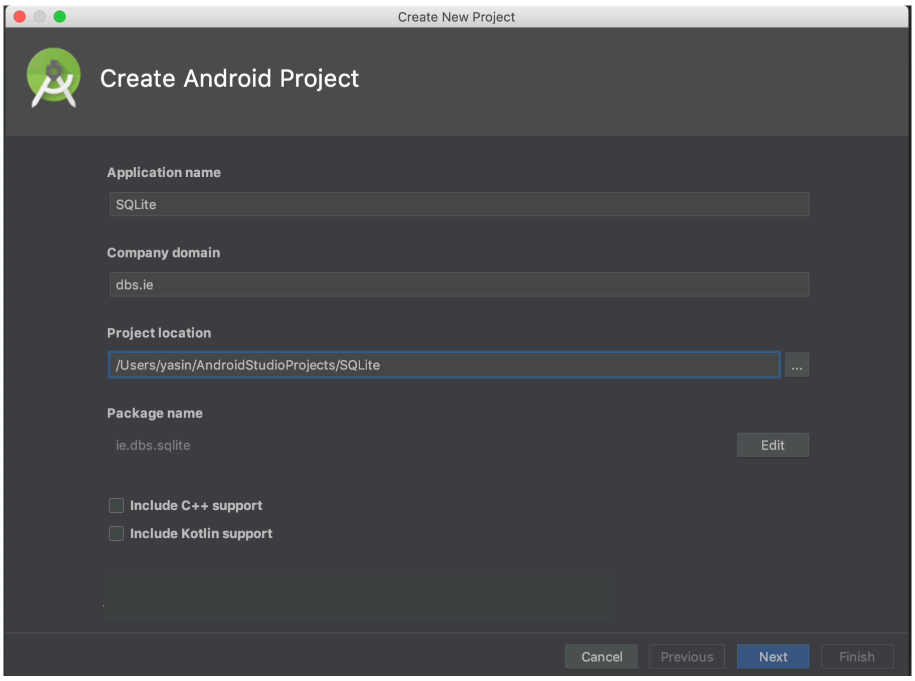
For each table in the database, we need to define a Java Class annotated with @Entity, and the primary key for the database table must be annotated with @PrimaryKey

For each @Entity, you will also need to define an interface for the DAO annotated with @Dao, within a DAO we need to define three annotations:

- @Query
- @Insert
- @Delete

@Query is asynchronous, while @Insert and @Delete are synchronous operations. To understand the usage better, we will start by looking at the implementation of above in our projects.

We will start by creating a new project, I am calling it SQLite


Create New Project

Create Android Project

Application name: SQLite

Company domain: dbs.ie

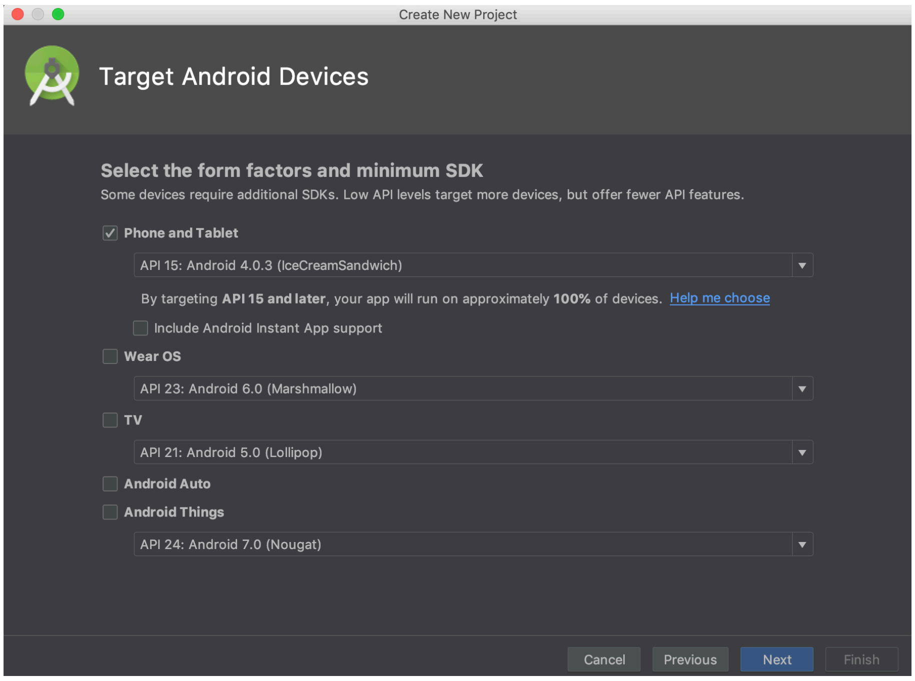
Project location: /Users/yasin/AndroidStudioProjects/SQLite

Package name: ie.dbs.sqlite

Include C++ support

Include Kotlin support

Let’s create a project called SQLite


Create New Project

Target Android Devices

Select the form factors and minimum SDK

Some devices require additional SDKs. Low API levels target more devices, but offer fewer API features.

✓ Phone and Tablet

API 15: Android 4.0.3 (IceCreamSandwich)

By targeting **API 15 and later**, your app will run on approximately **100%** of devices. Help me choose

Include Android Instant App support

Wear OS

API 23: Android 6.0 (Marshmallow)

TV

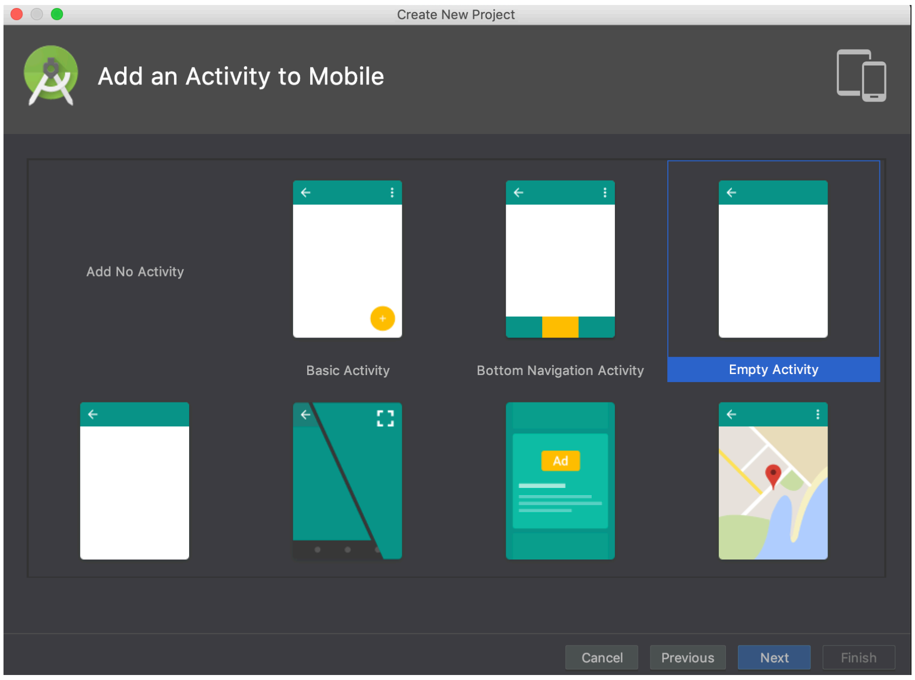
API 21: Android 5.0 (Lollipop)

Android Auto

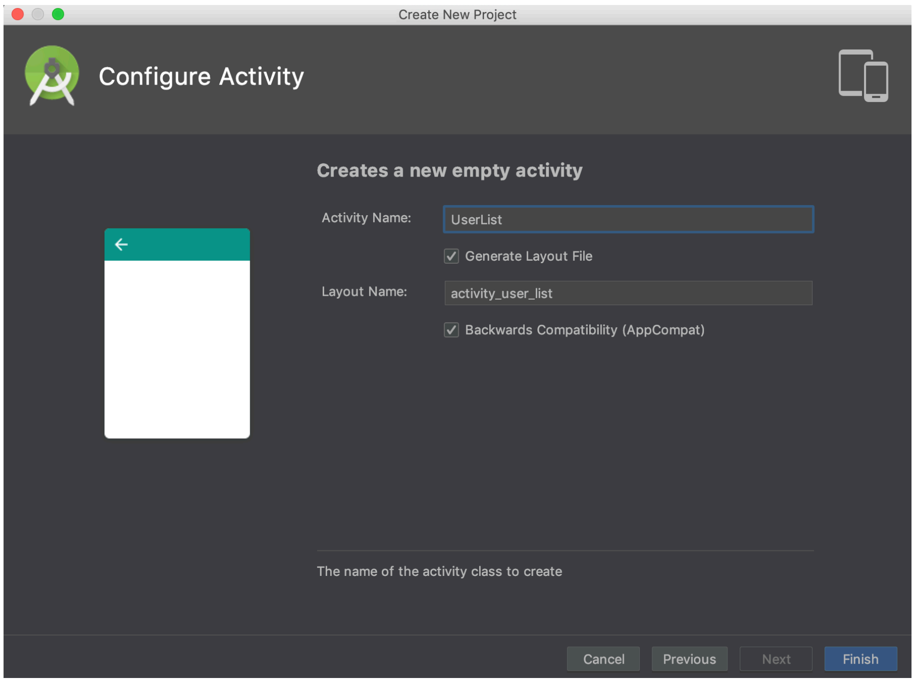
Android Things

API 24: Android 7.0 (Nougat)

Specify the targets for the project



Start by adding an empty activity



Creates a new empty activity

Activity Name: UserList

[✓] Generate Layout File

Layout Name: activity_user_list

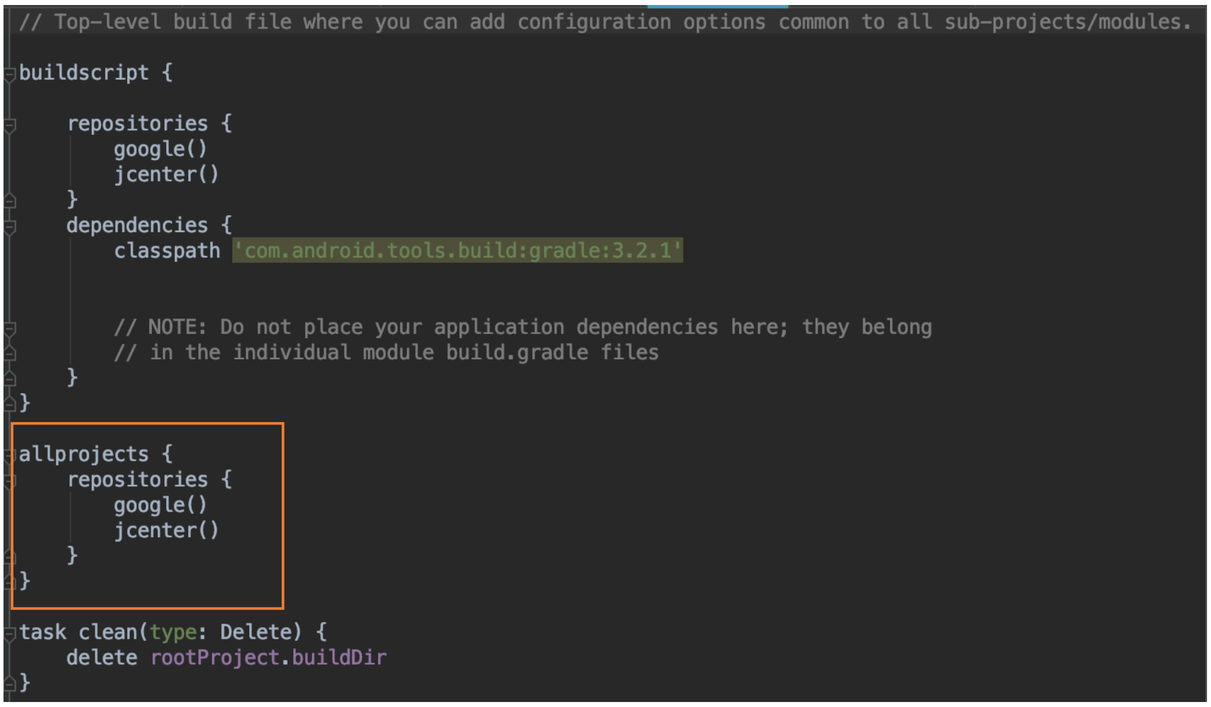
[✓] Backwards Compatibility (AppCompat)

The name of the activity class to create

Let's call the UserList

I have then added a splash screen to the project, follow the instructions from previous tutorials for this.

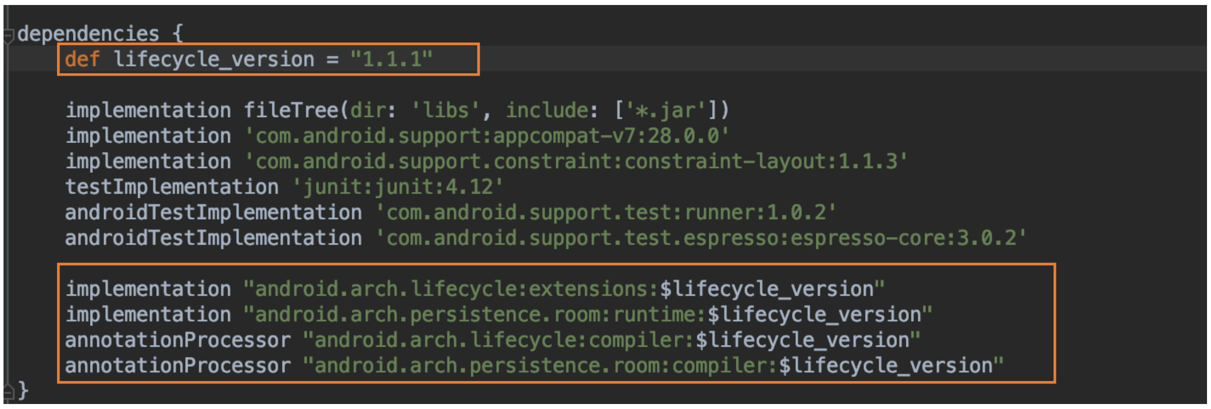
Then we start by adding the Room as SQL object mapping to our project. So, to get started the first thing we will need to do is make sure google() is included in our gradle repositories. (For this one, this should go to the gradle file for the project level)



```
// Top-level build file where you can add configuration options common to all sub-projects/modules.

buildscript {

    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.2.1'


        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

You don’t need to do anything on this step, if it is already there.

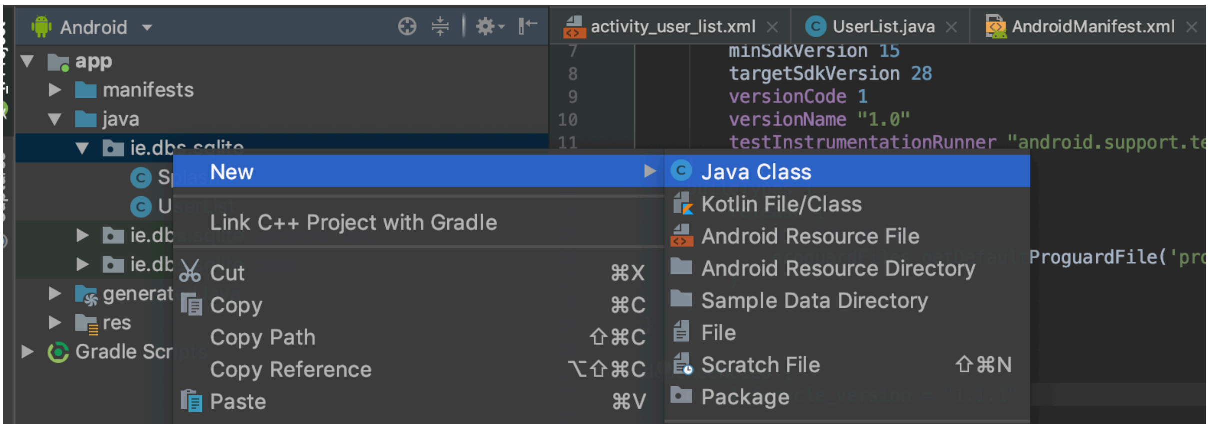
Then we will update the gradle file for module app to include the following dependencies:



```
dependencies {
    def lifecycle_version = "1.1.1"

    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:28.0.0'
    implementation 'com.android.support.constraint:constraint-layout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'

    implementation "android.arch.lifecycle:extensions:$lifecycle_version"
    implementation "android.arch.persistence.room:runtime:$lifecycle_version"
    annotationProcessor "android.arch.lifecycle:compiler:$lifecycle_version"
    annotationProcessor "android.arch.persistence.room:compiler:$lifecycle_version"
}
```

Now, we can start creating our database objects. So, first we will start by creating the User object, to contain all the fields our API endpoint returns for a user object.

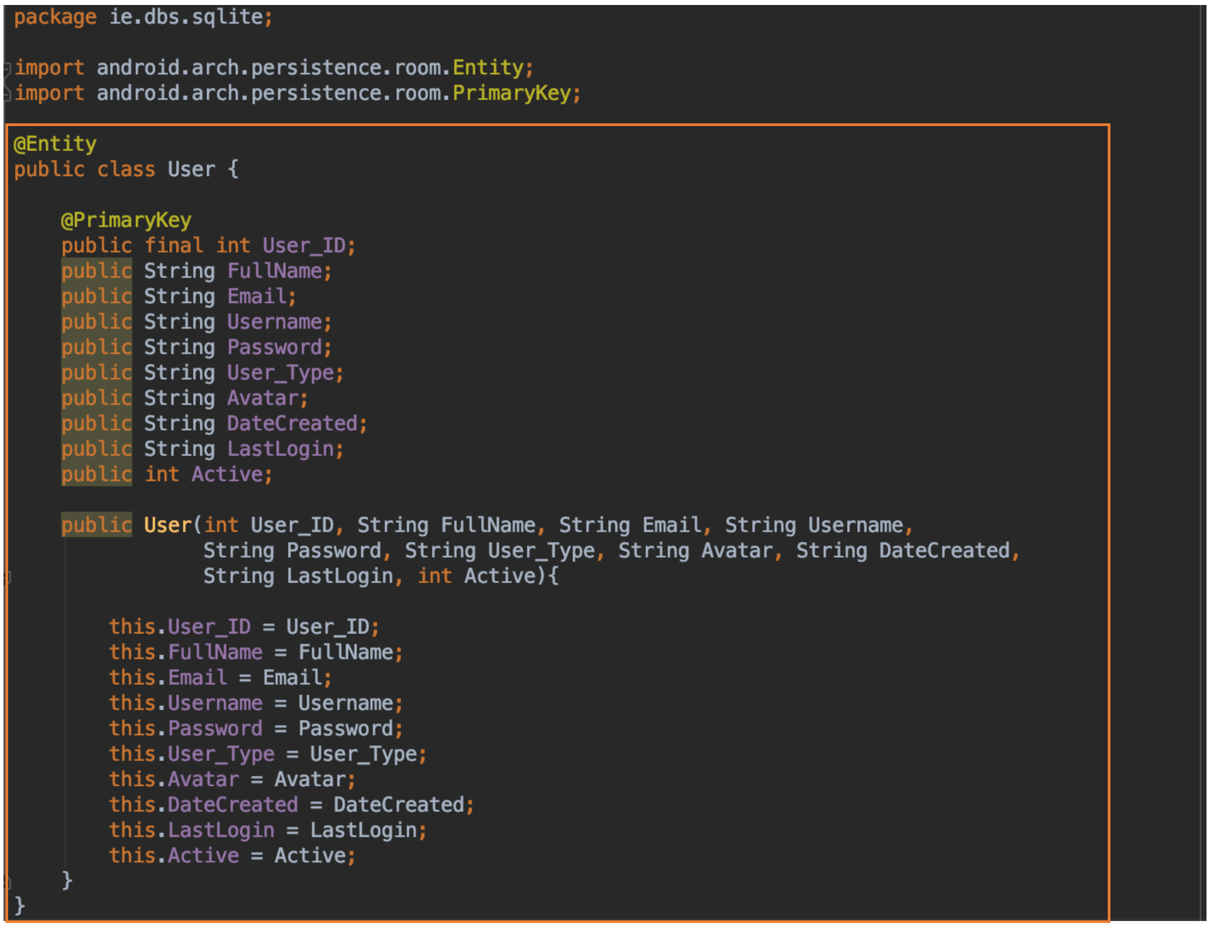
So we, will start by creating a new Java Class file, let’s call this User.



```
minSdkVersion 15
targetSdkVersion 28
versionCode 1
versionName "1.0"
testInstrumentationRunner "android.support.te
ProguardFile('pro
```

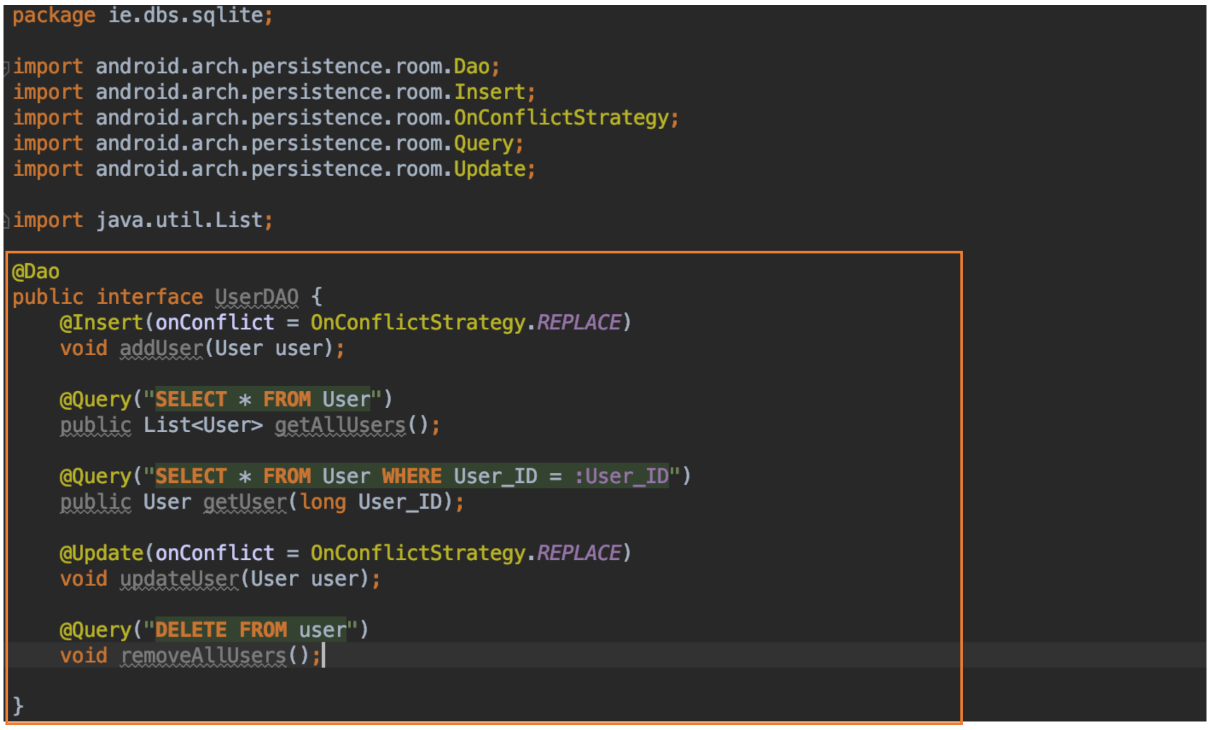
As mentioned earlier we will annotate this class with the @Entity to create a DB object for this. And the key element should be annotated with @PrimaryKey.

We will then need to add the rest of the properties and a public constructor to set these properties as per below:



```java
package ie.dbs.sqlite;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.PrimaryKey;

@Entity
public class User {

    @PrimaryKey
    public final int User_ID;
    public String FullName;
    public String Email;
    public String Username;
    public String Password;
    public String User_Type;
    public String Avatar;
    public String DateCreated;
    public String LastLogin;
    public int Active;

    public User(int User_ID, String FullName, String Email, String Username,
             String Password, String User_Type, String Avatar, String DateCreated,
             String LastLogin, int Active){

        this.User_ID = User_ID;
        this.FullName = FullName;
        this.Email = Email;
        this.Username = Username;
        this.Password = Password;
        this.User_Type = User_Type;
        this.Avatar = Avatar;
        this.DateCreated = DateCreated;
        this.LastLogin = LastLogin;
        this.Active = Active;
    }
}
```

Once, we have this the next step as previously mentioned should be to add a DAO. So, to do that, we will need to add a Java interface. We can call this UserDAO.



```
package ie.dbs.sqlite;

import android.arch.persistence.room.Dao;
import android.arch.persistence.room.Insert;
import android.arch.persistence.room.OnConflictStrategy;
import android.arch.persistence.room.Query;
import android.arch.persistence.room.Update;

import java.util.List;

@Dao
public interface UserDAO {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void addUser(User user);

    @Query("SELECT * FROM User")
    public List<User> getAllUsers();

    @Query("SELECT * FROM User WHERE User_ID = :User_ID")
    public User getUser(long User_ID);

    @Update(onConflict = OnConflictStrategy.REPLACE)
    void updateUser(User user);

    @Query("DELETE FROM user")
    void removeAllUsers();

}
```

So, first we have started with an interface with annotation of @Dao.

Then we have added the @Insert annotation and specified what action should be taken when there's a conflicting User_ID, and what method should be called to insert a User.

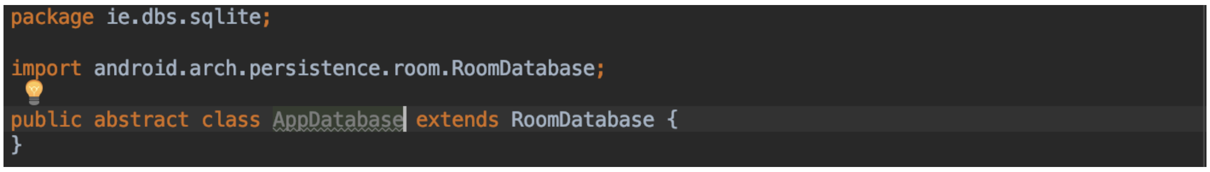
We have then two different @Query annotations, one to getAllUsers and what query should be executed on this action, second for getting a User for a specific User_ID and what query should be executed to get this.

The we have @Update strategy, and the method used to update a User.

Lastly, we have the removeUsers method, which specifies the DELETE query to be executed when this method is called.

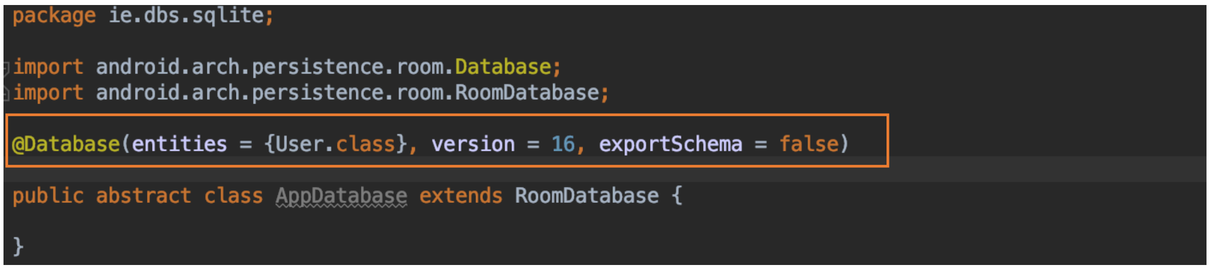
So, we have created the DAO, an Entity, we need a Database next to finish the Database Setup.

Let's finish the setup by adding the Database to our application, let's call this AppDatabase.



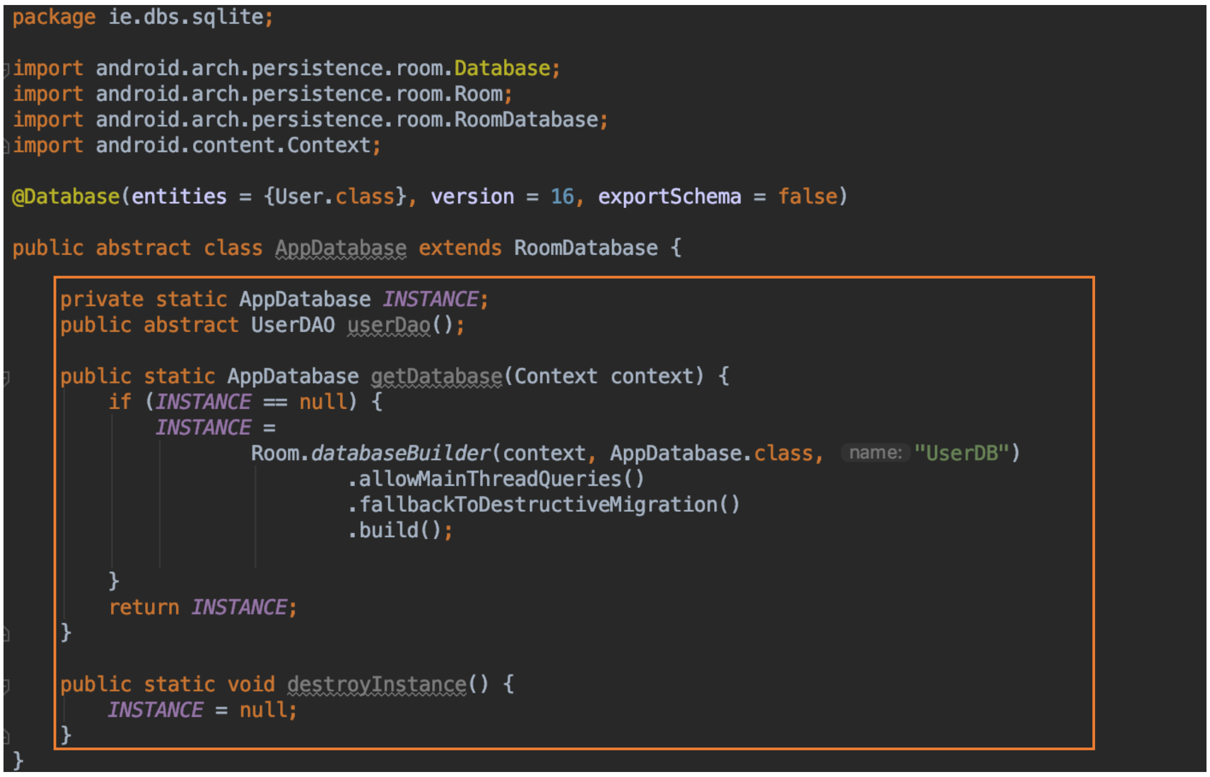
```
package ie.dbs.sqlite;

import android.arch.persistence.room.RoomDatabase;

public abstract class AppDatabase extends RoomDatabase {
}
```

We can now declare all the en tities for this Database



```java
package ie.dbs.sqlite;

import android.arch.persistence.room.Database;
import android.arch.persistence.room.RoomDatabase;

@Database(entities = {User.class}, version = 16, exportSchema = false)

public abstract class AppDatabase extends RoomDatabase {

}
```

Now, we can add the properties that will be used in this class to create DB objects



```java
package ie.dbs.sqlite;

import android.arch.persistence.room.Database;
import android.arch.persistence.room.Room;
import android.arch.persistence.room.RoomDatabase;
import android.content.Context;

@Database(entities = {User.class}, version = 16, exportSchema = false)

public abstract class AppDatabase extends RoomDatabase {

    private static AppDatabase INSTANCE;
    public abstract UserDAO userDao();

    public static AppDatabase getDatabase(Context context) {
        if (INSTANCE == null) {
            INSTANCE =
                    Room.databaseBuilder(context, AppDatabase.class, name: "UserDB")
                            .allowMainThreadQueries()
                            .fallbackToDestructiveMigration()
                            .build();

        }
        return INSTANCE;
    }

    public static void destroyInstance() {
        INSTANCE = null;
    }
}
```

So, first we have added an instance of the AppDatabase class which other app components can use to query the database or to insert to the database.

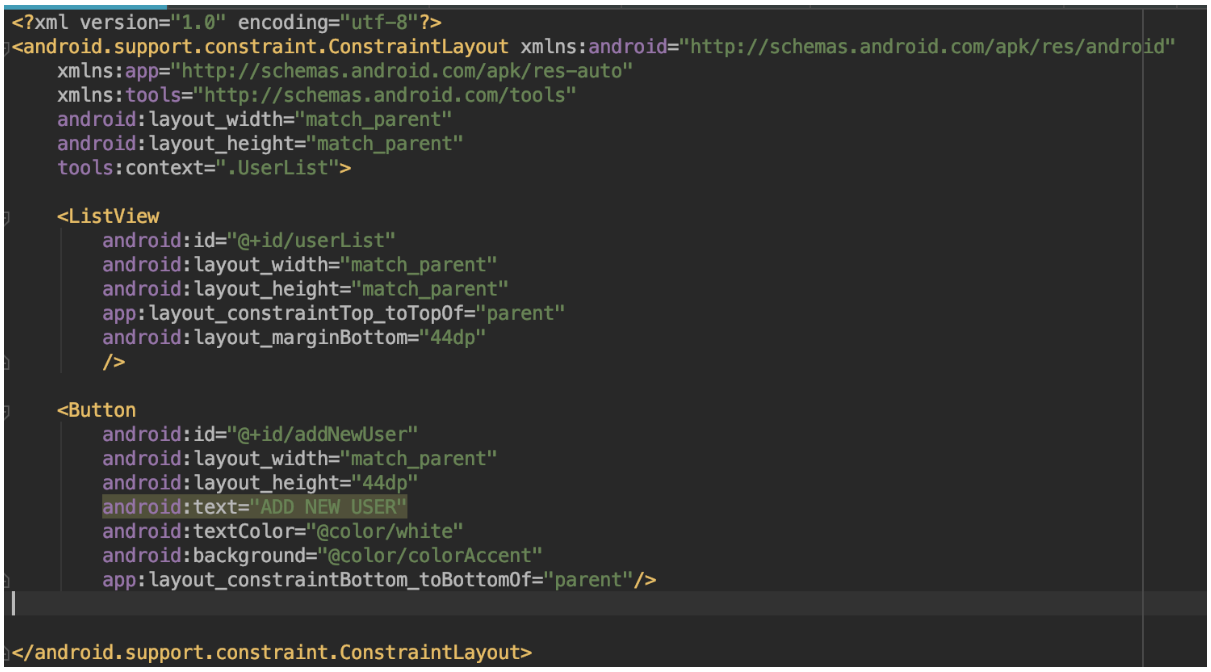
Then we have the UserDAO property, so that we can perform actions on the User entity.

We then have the static method that should return the AppDatabase instance to other components on request. Here we check if the instance is null, and initialize a new instance for the context, with database name UserDB.

I have added allowMainThreadQueries() for the purposes of this demo, we shouldn't be doing this on real life applications or on the CA submissions. As mentioned previously DB access requires access to the FileSystem which can be slow, hence this should be done asynchronously.

So, our database setup is complete with this, and now we should be able to start using this database in our activities, services or other app components we have implemented. We will now start by updating our UserList Activity to display a list of Users on the runtime, and this list will be obtained from the DB.

Let's update the activity_user_list.xml to below, so that we can use it to display the list of users.

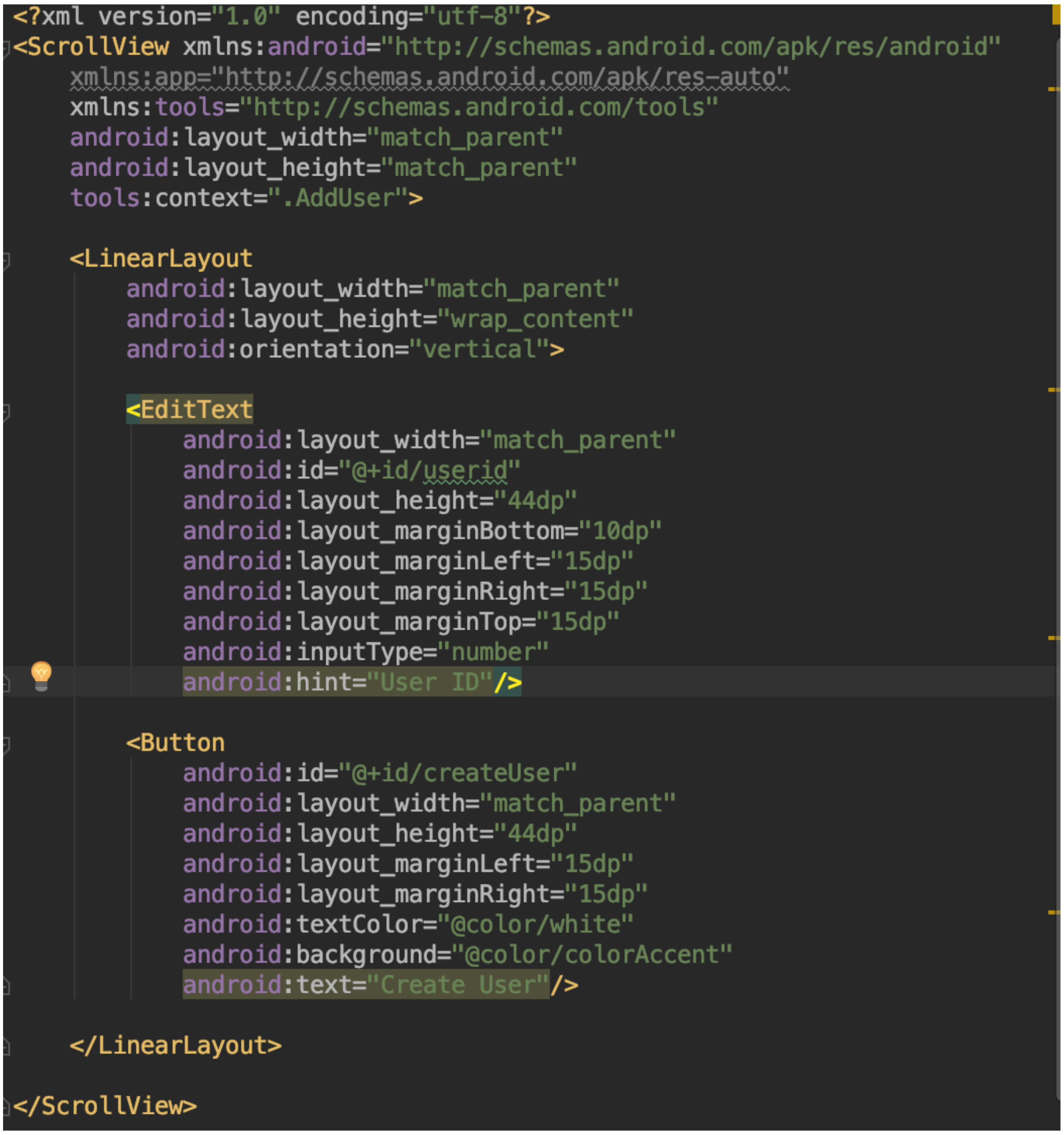


```xml
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".UserList">

    <ListView
        android:id="@+id/userList"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_constraintTop_toTopOf="parent"
        android:layout_marginBottom="44dp"
        />

    <Button
        android:id="@+id/addNewUser"
        android:layout_width="match_parent"
        android:layout_height="44dp"
        android:text="ADD NEW USER"
        android:textColor="@color/white"
        android:background="@color/colorAccent"
        app:layout_constraintBottom_toBottomOf="parent"/>


</android.support.constraint.ConstraintLayout>
```

We have a button that allows us to add new Users to our DB. This should be taking us to another activity which has the fields to populate and we should then be saving this to our DB.

So, let's add a new activity called AddUser to our project.

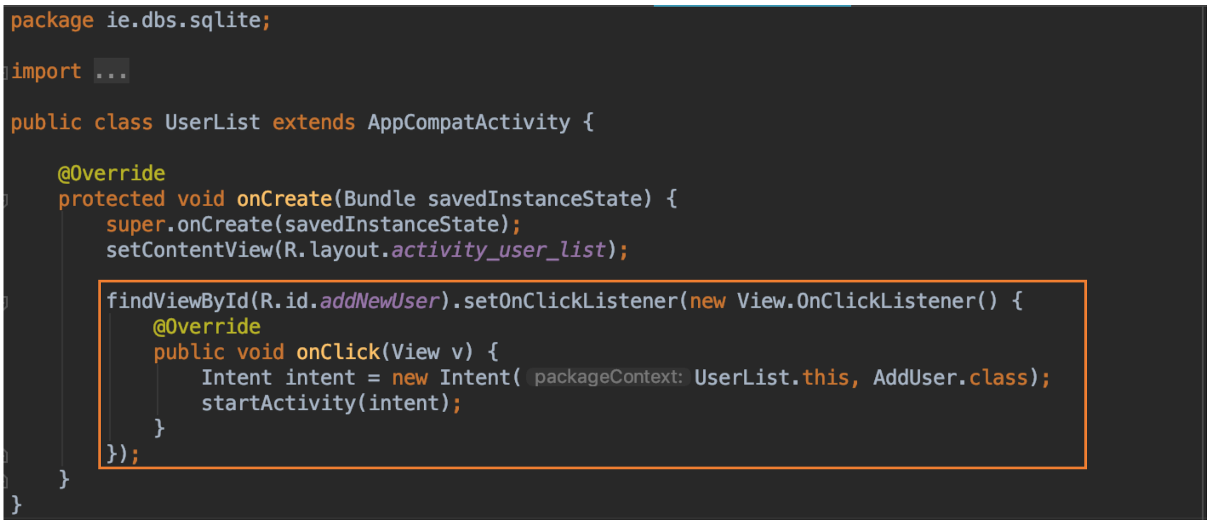
We will, update the xml for this new activity to the following:



```xml
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".AddUser">

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

        <EditText
            android:layout_width="match_parent"
            android:id="@+id/userid"
            android:layout_height="44dp"
            android:layout_marginBottom="10dp"
            android:layout_marginLeft="15dp"
            android:layout_marginRight="15dp"
            android:layout_marginTop="15dp"
            android:inputType="number"
            android:hint="User ID"/>

        <Button
            android:id="@+id/createUser"
            android:layout_width="match_parent"
            android:layout_height="44dp"
            android:layout_marginLeft="15dp"
            android:layout_marginRight="15dp"
            android:textColor="@color/white"
            android:background="@color/colorAccent"
            android:text="Create User"/>

    </LinearLayout>

</ScrollView>
```

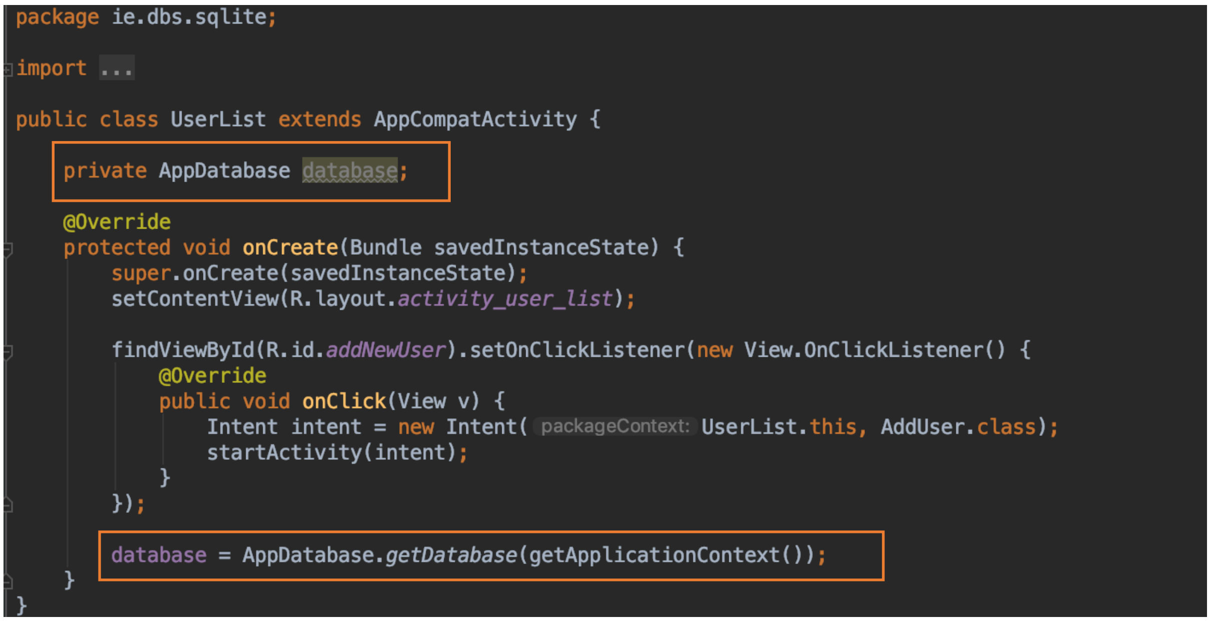
We should now, add the EditText for the rest of the fields within the DB object. I won't be adding the code to this tutorial for all of them, so you can copy the code and assign appropriate ids for all the fields.

We should now launch this activity when someone clicks on the Add New User button on the UserList activity. So, let's add an onClickListener for this.



```java
package ie.dbs.sqlite;

import ...

public class UserList extends AppCompatActivity {

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

        findViewById(R.id.addNewUser).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent( packageContext: UserList.this, AddUser.class);
                startActivity(intent);
            }
        });
    }
}
```

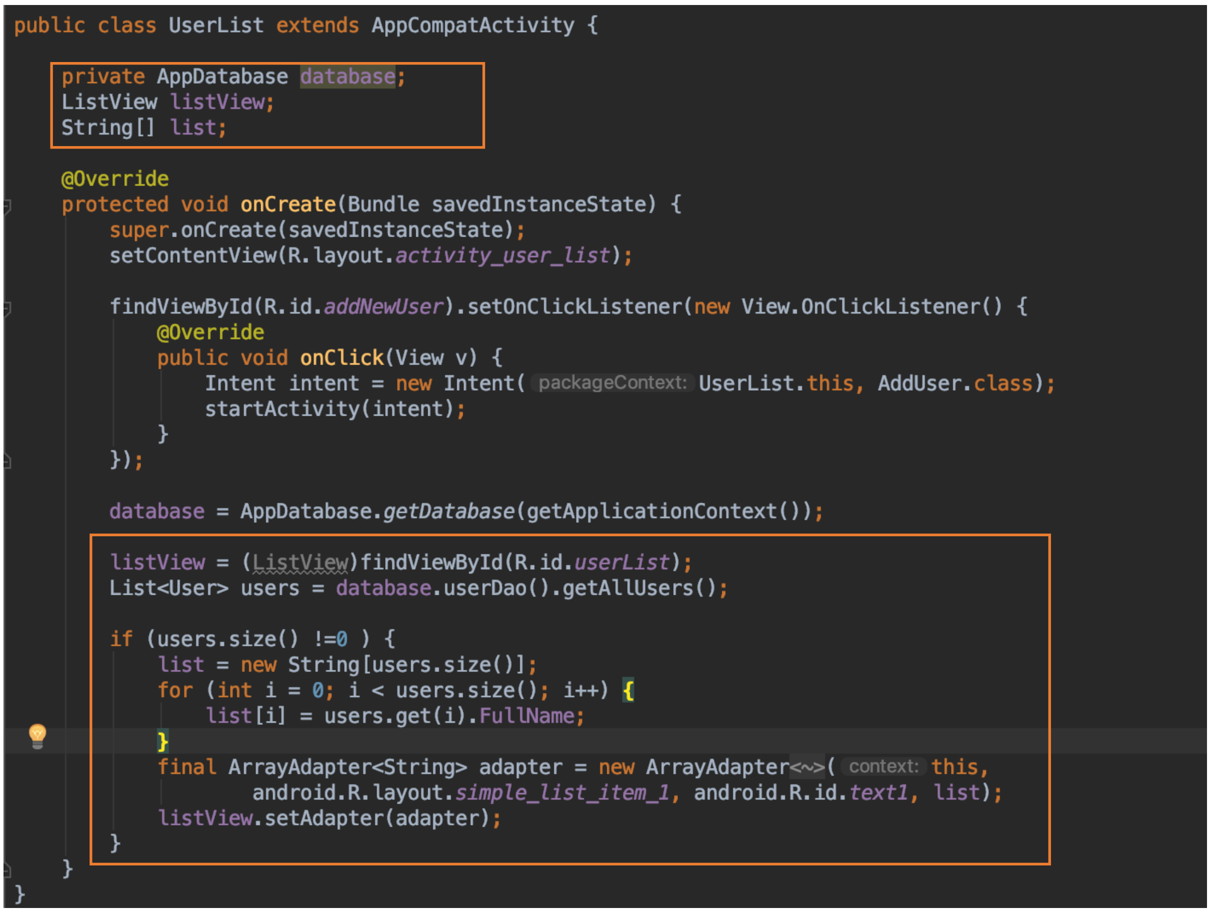
We should now add the adapter for the listView to obtain data from the database and display the users within the listview.



```java
package ie.dbs.sqlite;

import ...

public class UserList extends AppCompatActivity {

    private AppDatabase database;

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

        findViewById(R.id.addNewUser).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent( packageContext: UserList.this, AddUser.class);
                startActivity(intent);
            }
        });

        database = AppDatabase.getDatabase(getApplicationContext());
    }
}
```

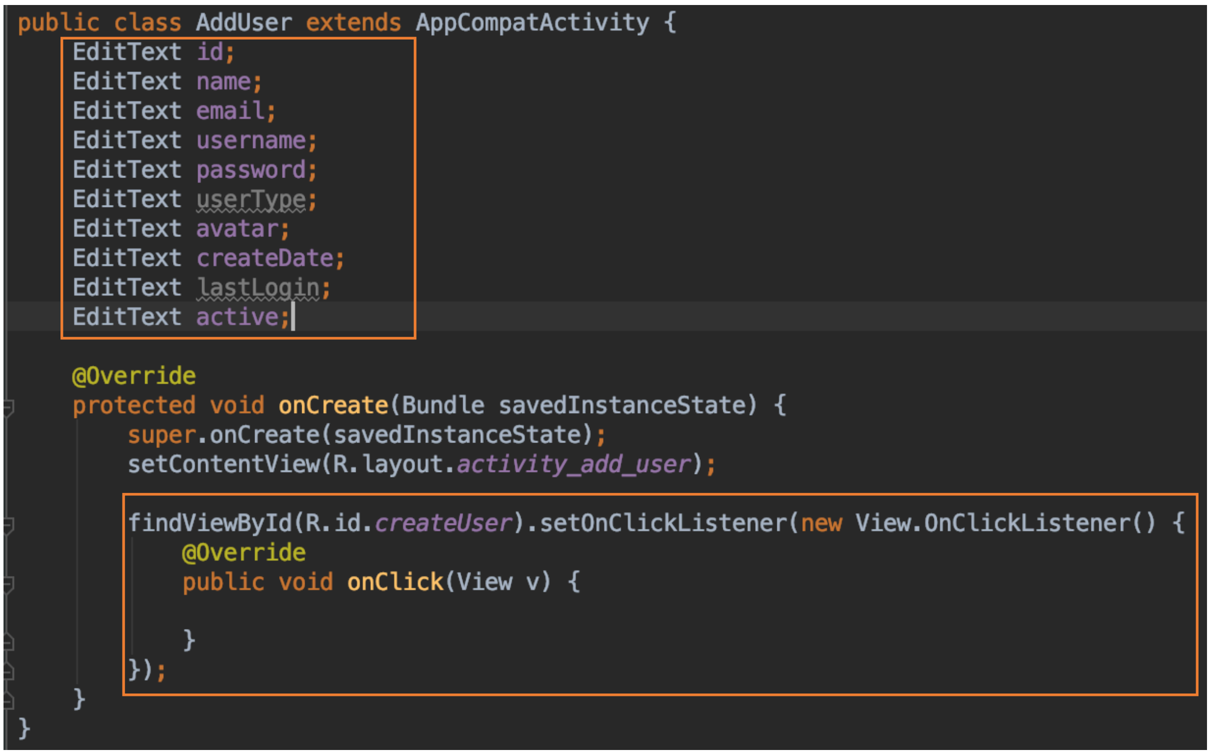
Then we obtain a list of users, check if there are any users in the list and assign them to an adapter and set the listview adapter.

This should however, only show an empty list, as we haven't yet added any users to our database.



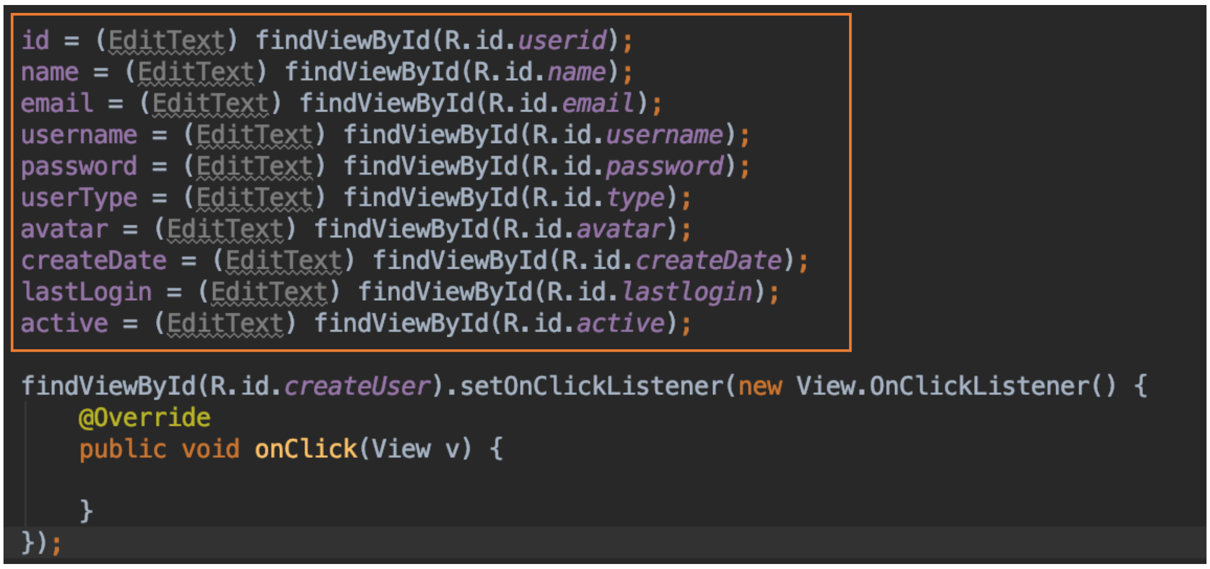
```java
public class UserList extends AppCompatActivity {

    private AppDatabase database;
    ListView listView;
    String[] list;

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

        findViewById(R.id.addNewUser).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent( packageContext: UserList.this, AddUser.class);
                startActivity(intent);
            }
        });

        database = AppDatabase.getDatabase(getApplicationContext());

        listView = (ListView)findViewById(R.id.userList);
        List<User> users = database.userDao().getAllUsers();

        if (users.size() !=0 ) {
            list = new String[users.size()];
            for (int i = 0; i < users.size(); i++) {
                list[i] = users.get(i).FullName;
            }
            final ArrayAdapter<String> adapter = new ArrayAdapter<~>( context: this,
                    android.R.layout.simple_list_item_1, android.R.id.text1, list);
            listView.setAdapter(adapter);
        }
    }
}
```

Let’s now add the user to our database, we will now need to implement the click listener for the create user button on our second activity.



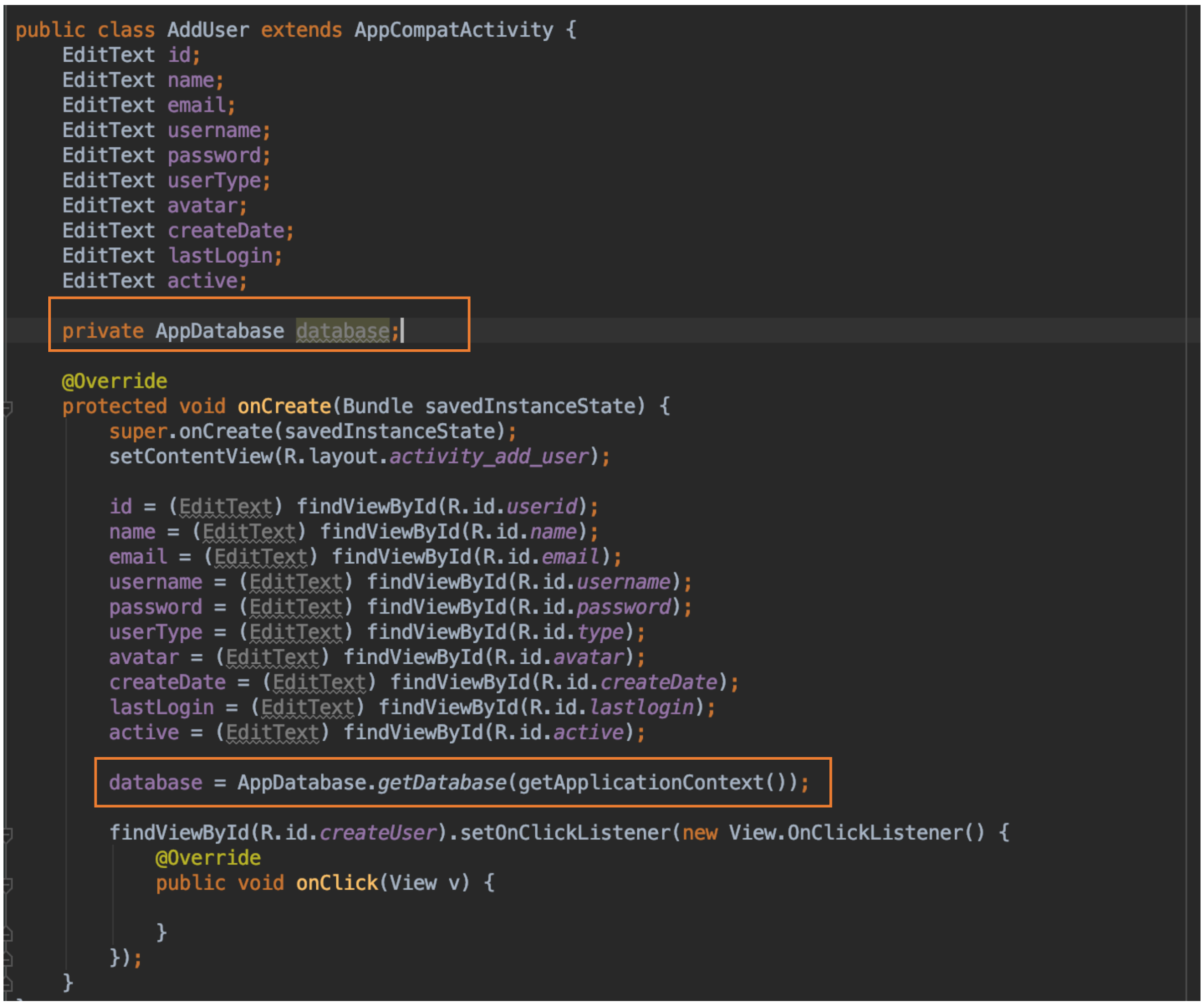
```java
public class AddUser extends AppCompatActivity {
    EditText id;
    EditText name;
    EditText email;
    EditText username;
    EditText password;
    EditText userType;
    EditText avatar;
    EditText createDate;
    EditText lastLogin;
    EditText active;

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

        findViewById(R.id.createUser).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {

            }
        });
    }
}
```

We should now, assign references to the properties we have defined earlier in this activity.

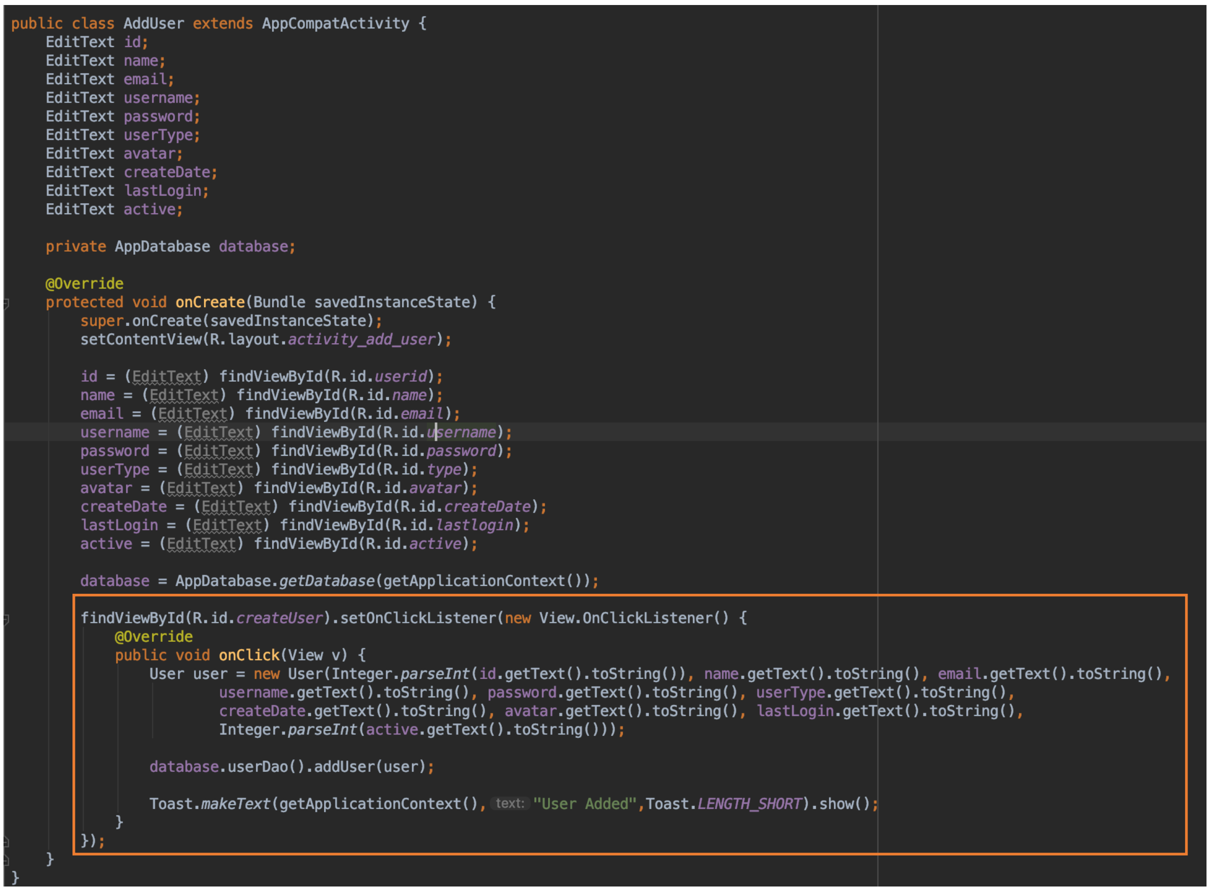


```
id = (EditText) findViewById(R.id.userid);
name = (EditText) findViewById(R.id.name);
email = (EditText) findViewById(R.id.email);
username = (EditText) findViewById(R.id.username);
password = (EditText) findViewById(R.id.password);
userType = (EditText) findViewById(R.id.type);
avatar = (EditText) findViewById(R.id.avatar);
createDate = (EditText) findViewById(R.id.createDate);
lastLogin = (EditText) findViewById(R.id.lastlogin);
active = (EditText) findViewById(R.id.active);

findViewById(R.id.createUser).setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {

    }
});
```

Let’s now save this data to our database

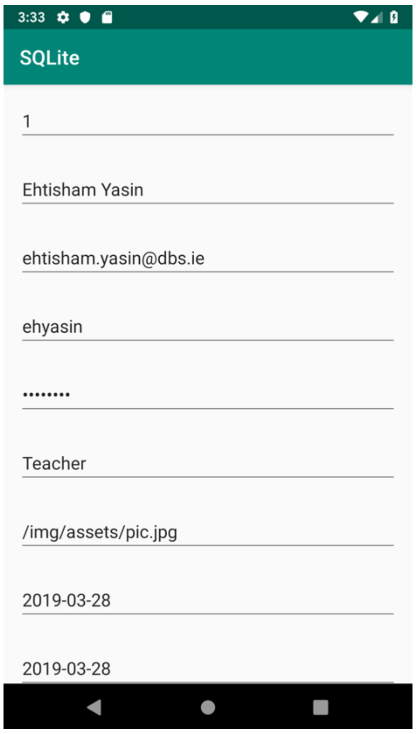


```
public class AddUser extends AppCompatActivity {
    EditText id;
    EditText name;
    EditText email;
    EditText username;
    EditText password;
    EditText userType;
    EditText avatar;
    EditText createDate;
    EditText lastLogin;
    EditText active;

    private AppDatabase database;

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

        id = (EditText) findViewById(R.id.userid);
        name = (EditText) findViewById(R.id.name);
        email = (EditText) findViewById(R.id.email);
        username = (EditText) findViewById(R.id.username);
        password = (EditText) findViewById(R.id.password);
        userType = (EditText) findViewById(R.id.type);
        avatar = (EditText) findViewById(R.id.avatar);
        createDate = (EditText) findViewById(R.id.createDate);
        lastLogin = (EditText) findViewById(R.id.lastlogin);
        active = (EditText) findViewById(R.id.active);

        database = AppDatabase.getDatabase(getApplicationContext());

        findViewById(R.id.createUser).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {

            }
        });
    }
```



```java
public class AddUser extends AppCompatActivity {
    EditText id;
    EditText name;
    EditText email;
    EditText username;
    EditText password;
    EditText userType;
    EditText avatar;
    EditText createDate;
    EditText lastLogin;
    EditText active;

    private AppDatabase database;

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

        id = (EditText) findViewById(R.id.userid);
        name = (EditText) findViewById(R.id.name);
        email = (EditText) findViewById(R.id.email);
        username = (EditText) findViewById(R.id.username);
        password = (EditText) findViewById(R.id.password);
        userType = (EditText) findViewById(R.id.type);
        avatar = (EditText) findViewById(R.id.avatar);
        createDate = (EditText) findViewById(R.id.createDate);
        lastLogin = (EditText) findViewById(R.id.lastlogin);
        active = (EditText) findViewById(R.id.active);

        database = AppDatabase.getDatabase(getApplicationContext());

        findViewById(R.id.createUser).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                User user = new User(Integer.parseInt(id.getText().toString()), name.getText().toString(), email.getText().toString(),
                        username.getText().toString(), password.getText().toString(), userType.getText().toString(),
                        createDate.getText().toString(), avatar.getText().toString(), lastLogin.getText().toString(),
                        Integer.parseInt(active.getText().toString()));

                database.userDao().addUser(user);

                Toast.makeText(getApplicationContext(), text: "User Added",Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

So, now you should be able to run this application to see an empty screen to start with. But if you go and add a new user on the user screen, quit the app and relaunch it the user should be in the list.



1

Ehtisham Yasin

ehtisham.yasin@dbs.ie

ehyasin

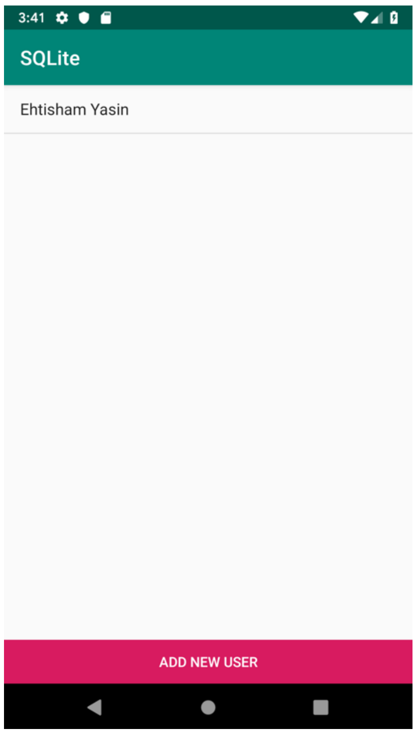
••••••••

Teacher

/img/assets/pic.jpg

2019-03-28

2019-03-28



Ehtisham Yasin

Now, you should be able to use this to store the login information of the last logged in user. For the next part, you should implement onResume, so that it refreshes the list as soon as the data is created and the user return to this screen instead of on relaunch of the application.