To connect an Android app to a **PostgreSQL database** hosted on a cloud service (such as **Heroku**, **Supabase**, or any other PostgreSQL provider), you would generally use **JDBC (Java Database Connectivity)** for database operations.

Below is the step-by-step guide on how to set up PostgreSQL connection and perform INSERT, SELECT, and DELETE operations from your Android app.

**1. Add JDBC Dependencies to Your build.gradle File**

In Android, you need to add the necessary dependencies for JDBC to work. Unfortunately, Android does not support full JDBC in the same way a typical Java application would. However, you can use a library like **PostgreSQL JDBC Driver** or use **JDBC for Android** via third-party solutions.

Add the following dependency to your app/build.gradle:

dependencies {

implementation 'org.postgresql:postgresql:42.5.0' // PostgreSQL JDBC driver

}

This dependency will allow your Android app to communicate with a PostgreSQL database.

**2. Permissions in AndroidManifest.xml**

To allow your app to access the internet, you need to add the INTERNET permission in your AndroidManifest.xml:

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

**3. Create a PostgreSQL Database Connection Class**

You’ll need to create a class to handle the connection to your PostgreSQL database using JDBC. Here’s an example of how to set this up.

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class DatabaseHelper {

// Replace with your database connection details

private static final String DB\_URL = "jdbc:postgresql://your\_database\_url:5432/your\_database\_name";

private static final String DB\_USER = "your\_username";

private static final String DB\_PASSWORD = "your\_password";

private Connection connection = null;

// Establish a connection to the PostgreSQL database

public Connection connect() throws SQLException {

if (connection == null || connection.isClosed()) {

try {

// Load the JDBC driver (this is generally not necessary with newer JDBC versions)

Class.forName("org.postgresql.Driver");

// Connect to the database

connection = DriverManager.getConnection(DB\_URL, DB\_USER, DB\_PASSWORD);

} catch (ClassNotFoundException | SQLException e) {

throw new SQLException("Connection failed.", e);

}

}

return connection;

}

// Close the connection to the database

public void disconnect() {

if (connection != null) {

try {

connection.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

}

**4. Insert Data into the Database**

To insert data into the database, you will use the INSERT INTO statement. Here’s how you can do it:

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.SQLException;

public class DatabaseHelper {

// Insert a new user into the "User" table

public void insertUser(String email, String name, int rating) {

String query = "INSERT INTO \"User\" (email, name, rating) VALUES (?, ?, ?)";

try (Connection connection = connect();

PreparedStatement statement = connection.prepareStatement(query)) {

statement.setString(1, email);

statement.setString(2, name);

statement.setInt(3, rating);

int rowsAffected = statement.executeUpdate();

if (rowsAffected > 0) {

System.out.println("User inserted successfully.");

}

} catch (SQLException e) {

e.printStackTrace();

}

}

}

* **Explanation**: The insertUser method prepares an SQL query with placeholders (?) and binds the actual values (email, name, rating) to them using setString and setInt. executeUpdate() is used for queries that modify the database (like INSERT, UPDATE, or DELETE).

**5. Select Data from the Database**

To select data, you can use the SELECT statement. Here’s how to retrieve and display user information:

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class DatabaseHelper {

// Select users from the "User" table

public void selectUsers() {

String query = "SELECT UID, email, name, rating FROM \"User\"";

try (Connection connection = connect();

PreparedStatement statement = connection.prepareStatement(query)) {

ResultSet resultSet = statement.executeQuery();

while (resultSet.next()) {

int uid = resultSet.getInt("UID");

String email = resultSet.getString("email");

String name = resultSet.getString("name");

int rating = resultSet.getInt("rating");

// Handle the results

System.out.println("User ID: " + uid + ", Email: " + email + ", Name: " + name + ", Rating: " + rating);

}

} catch (SQLException e) {

e.printStackTrace();

}

}

}

* **Explanation**: The selectUsers method executes a SELECT query to retrieve users from the User table. The results are stored in a ResultSet, and you can iterate over it to access each row.

**6. Delete Data from the Database**

To delete data, you can use the DELETE FROM statement:

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.SQLException;

public class DatabaseHelper {

// Delete a user from the "User" table

public void deleteUser(int uid) {

String query = "DELETE FROM \"User\" WHERE UID = ?";

try (Connection connection = connect();

PreparedStatement statement = connection.prepareStatement(query)) {

statement.setInt(1, uid);

int rowsAffected = statement.executeUpdate();

if (rowsAffected > 0) {

System.out.println("User deleted successfully.");

}

} catch (SQLException e) {

e.printStackTrace();

}

}

}

* **Explanation**: The deleteUser method executes a DELETE query where it uses the setInt method to bind the UID value to the query.

**7. Async Execution in Android (Optional, but Recommended)**

Since database operations can be time-consuming and may block the main UI thread, it's important to run database queries asynchronously, typically using **AsyncTask** or **Java’s ExecutorService** for better performance in Android.

For example, you can modify your DatabaseHelper methods to run inside an AsyncTask:

import android.os.AsyncTask;

public class DatabaseTask extends AsyncTask<Void, Void, Void> {

private DatabaseHelper dbHelper;

@Override

protected Void doInBackground(Void... params) {

dbHelper = new DatabaseHelper();

// Call the database operation here

dbHelper.insertUser("example@example.com", "John Doe", 5);

dbHelper.selectUsers();

dbHelper.deleteUser(1);

return null;

}

@Override

protected void onPostExecute(Void aVoid) {

super.onPostExecute(aVoid);

// Update the UI or notify the user

}

}

You can then execute it as follows:

new DatabaseTask().execute();

**8. Handling Cloud Database Credentials**

Ensure that your cloud database (e.g., Supabase, Heroku, etc.) allows external connections, and be mindful of exposing sensitive credentials in your Android app. Use environment variables or secure methods to keep credentials safe. For cloud-based databases, you should often find the necessary credentials in the service's dashboard.

**Final Notes:**

1. **Cloud Database Security**: Ensure that your cloud database provider supports **SSL** connections and that you're securely managing your credentials, as cloud databases should never be exposed directly.
2. **Asynchronous Operations**: Database operations should always be done in the background to avoid blocking the main UI thread.
3. **Error Handling**: Make sure to handle database connection errors, SQL exceptions, and network issues in your production app.
4. **ProGuard (or R8)**: If you are obfuscating your app with ProGuard or R8, make sure that the PostgreSQL JDBC classes are excluded from obfuscation.

Let me know if you need further clarification or additional examples!