Mobile Application Development

Produced by

David Drohan (ddrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology
http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Android Persistence using SQLite

Agenda & Goals

- Be aware of the different approaches to data persistence in Android Development
- Be able to work with the **`SQLiteOpenHelper`** & **`SQLiteDatabase`** classes to implement an SQLite database on an Android device (to manage our Coffees)
- Be able to work with **`Realm`** to implement a noSQL database on an Android device (again, to manage our Coffees)
- Be able to work with **`SharedPreferences`** to manage, for example, basic Login & Register screens

Data Storage Solutions *

- **Shared Preferences**
  - Store private primitive data in key-value pairs.
- Internal Storage
  - Store private data on the device memory.
- External Storage
  - Store public data on the shared external storage.
- **SQLite Databases**
  - Store structured data in a private database.
- Network Connection
  - Store data on the web with your own network server.

Data Storage Solutions *

❑ Bundle Class

- A mapping from String values to various `Parcelable` types and functionally equivalent to a standard `Map`.
- Does not handle Back button scenario. App restarts from scratch with no saved data in that case.

❑ File

- Use `java.io.*` to read/write data on the device's internal storage.

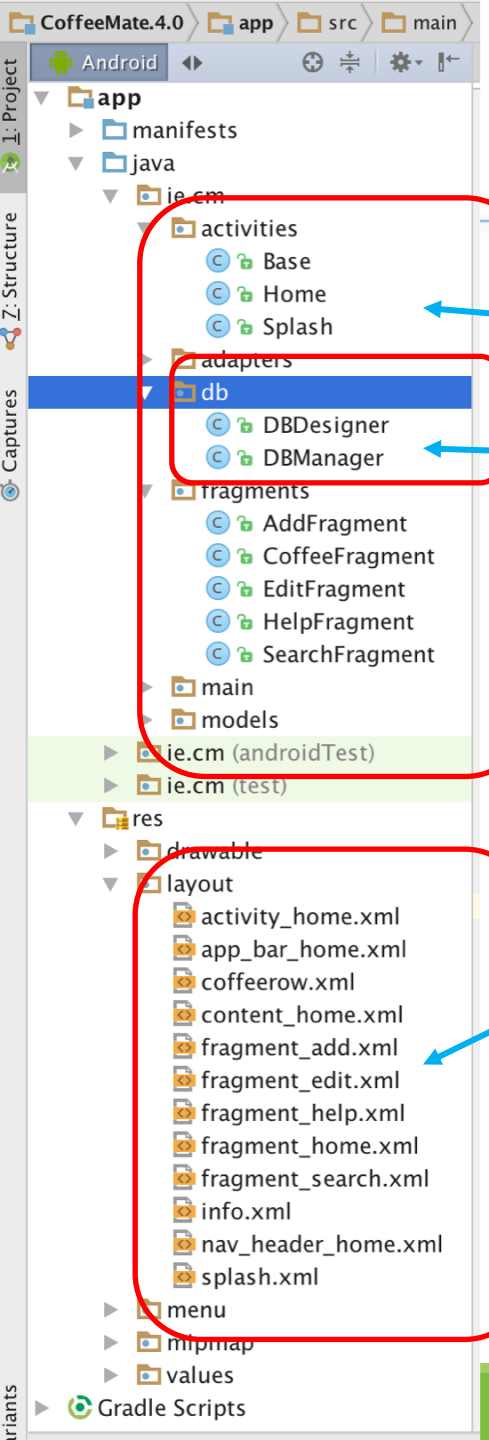
❑ Realm Databases

- Store non-structured data in a private database.

CoffeeMate.4.0

Using an SQLite Database

CoffeeMate 4.0 – Project Structure



- 15 java source files in total
  - Our Database classes
- xml layouts
- xml menu
- xml files for resources
- xml ‘configuration’ file

Idea

- Goal
  - Enhance **CoffeeMate.3.0** by managing the Coffees in an SQLite Database and improving the UI/UX with a Nav Drawer
- Approach
  - Implement/extend specific classes to add the database functionality to the app – Practical Lab 5

Database Programming in Android *

- Android provides full support for `SQLite` databases. Any databases you create will be accessible by name to any class in the application, but not outside the application.
- The recommended method to create a new SQLite database is to create a subclass of `SQLiteOpenHelper` and override the `onCreate()` method, in which you can execute a SQLite command to create tables in the database. For example:

```
public class DictionaryOpenHelper extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 2;
    private static final String DICTIONARY_TABLE_NAME = "dictionary";
    private static final String DICTIONARY_TABLE_CREATE =
                "CREATE TABLE " + DICTIONARY_TABLE_NAME + " (" +
                KEY_WORD + " TEXT, " +
                KEY_DEFINITION + " TEXT);";

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

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(DICTIONARY_TABLE_CREATE);
    }
}
```

Database Programming in Android *

- You can then get an instance of your `SQLiteOpenHelper` implementation using the constructor you've defined. To write to and read from the database, call `getWritableDatabase()` and `getReadableDatabase()`, respectively. These both return a `SQLiteDatabase` object that represents the database and provides methods for `SQLite` operations.
- You can execute `SQLite` queries using the `SQLiteDatabase query()` methods, which accept various query parameters, such as the table to query, the projection, selection, columns, grouping, and others. For complex queries, such as those that require column aliases, you should use `SQLiteQueryBuilder`, which provides several convenient methods for building queries.
- Every SQLite query will return a `Cursor` that points to all the rows found by the query. The Cursor is always the mechanism with which you can navigate results from a database query and read rows and columns.

Database Programming in Android

- With `SQLite`, the database is a simple disk file. All of the data structures making up a relational database - tables, views, indexes, etc. - are within this file
- RDBMS is provided through the api classes so it becomes part of your app
- You can use the SQL you learned in a database module
- You should use DB best practices
  - Normalize data
  - Encapsulate database info in helper or wrapper classes
  - Don’t store files (e.g. images or audio), Instead just store the path string

Step 1 - Define a Schema (and Contract)

- One of the main principles of SQL databases is the schema: a formal declaration of how the database is organized. The schema is reflected in the SQL statements that you use to create your database.
- You may find it helpful to create a companion class, known as a *contract* class, which explicitly specifies the layout of your schema in a systematic and self-documenting way.
- A contract class is a container for constants that define names for URIs, tables, and columns. The contract class allows you to use the same constants across all the other classes in the same package. This lets you change a column name in one place and have it propagate throughout your code.

CoffeeMate - DBDesigner

```
public class DBDesigner extends SQLiteOpenHelper
{
    public static final String TABLE_COFFEE = "table_coffee";
    public static final String COLUMN_ID = "coffeeid";
    public static final String COLUMN_NAME = "coffeename";
    public static final String COLUMN_SHOP = "shop";
    public static final String COLUMN_RATING = "rating";
    public static final String COLUMN_PRICE = "price";
    public static final String COLUMN_FAV = "isfavourite";

    private static final String DATABASE_NAME = "coffeemate.db";
    private static final int DATABASE_VERSION = 1;

    // Database creation sql statement
    private static final String DATABASE_CREATE_TABLE_COFFEE = "create table "
            + TABLE_COFFEE + "( " + COLUMN_ID + " integer primary key autoincrement, "
            + COLUMN_NAME + " text not null,"
            + COLUMN_SHOP + " text not null,"
            + COLUMN_PRICE + " double not null,"
            + COLUMN_RATING + " double not null,"
            + COLUMN_FAV + " integer not null);"; //SQLite doesn't support boolean types

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

    @Override
    public void onCreate(SQLiteDatabase database) { database.execSQL(DATABASE_CREATE_TABLE_COFFEE); }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        Log.w(DBDesigner.class.getName(),
                "Upgrading database from version " + oldVersion + " to "
                        + newVersion + ", which will destroy all old data");
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_COFFEE);
        onCreate(db);
    }
}
```

Our Table & Column names (for SQL)

Creating the Table (or Tables)

Drop the Table (if we change the schema)

Step 2 - Define a Schema (and Contract)

- Once you have defined how your database looks, you should implement methods that create and maintain the database and your general **CRUD functionality**.
- From a design perspective, it can be a good idea to place this functionality in a separate *manager* class, to allow for reuse and SoC (Separation of Concerns)
- This class can act as an APi (of sorts) to allow access to the database from any other class in your application.
- You can refactor your *manager* class internally, without it affecting how other classes call/use the *manager* methods.

CoffeeMate - DBManager

```
public class DBManager {

    private SQLiteDatabase database;
    private DBDesigner dbHelper;

    public DBManager(Context context) { dbHelper = new DBDesigner(context); }

    public void open() throws SQLException {
        database = dbHelper.getWritableDatabase();
    }

    public void close() { database.close(); }

    public void insert(Coffee c) {
        ContentValues values = new ContentValues();
        values.put(DBDesigner.COLUMN_NAME, c.name);
        values.put(DBDesigner.COLUMN_SHOP, c.shop);
        values.put(DBDesigner.COLUMN_PRICE, c.price);
        values.put(DBDesigner.COLUMN_RATING, c.rating);
        values.put(DBDesigner.COLUMN_FAV, (c.favourite == true) ? 1 : 0);

        long insertId = database.insert(DBDesigner.TABLE_COFFEE, null,
                values);
    }

    public void delete(int id) {...}

    public void update(Coffee c) {...}
```

Our database reference

Returns a reference to the database created from our SQL string

ContentValues are key/value pairs that are used when inserting/updating databases. Each ContentValue object corresponds to one row in a table

CoffeeMate – DBManager *

```
    public void delete(int id) {...}

    public void update(Coffee c) {...}

    public List<Coffee> getAll() {
        List<Coffee> coffees = new ArrayList<>();
        Cursor cursor = database.rawQuery("SELECT * FROM "
                + DBDesigner.TABLE_COFFEE, null);
        cursor.moveToFirst();
        while (!cursor.isAfterLast()) {
            Coffee pojo = toCoffee(cursor);
            coffees.add(pojo);
            cursor.moveToNext();
        }
        // Make sure to close the cursor
        cursor.close();
        return coffees;
    }

    public Coffee get(int id) {...}

    public List<Coffee> getFavourites() {...}

    private Coffee toCoffee(Cursor cursor) {
        Coffee pojo = new Coffee();
        pojo.coffeeId = cursor.getInt(0);
        pojo.name = cursor.getString(1);
        pojo.shop = cursor.getString(2);
        pojo.price = cursor.getDouble(3);
        pojo.rating = cursor.getDouble(4);
        pojo.favourite = (cursor.getInt(5) == 1) ? true : false;

        return pojo;
    }

    public void setupList() {...}
}
```

This method 'converts' a Cursor object into a Coffee Object

A Cursor provides random read-write access to the resultset returned by a database query

Other Cursor Functions

- moveToPrevious
- getCount
- getColumnIndexOrThrow
- getColumnName
- getColumnNames
- moveToPosition
- getPosition

Questions?


Thanks!

Appendix

- ❑ Files
- ❑ Content Providers
- ❑ And a bit on Bundles…

Using Files

File Access (Internal & External)

- Store data to file
- Use java.io.* to read/write file
- Only local file can be visited
  - Advantages: can store large amounts of data
  - Disadvantages: file format changes and/or updates may result in significant programming/refactoring
- Very similar to file handling in java desktop applications
- Generally though, not recommended

Read from a file

- Open a File for input
  - Context.openFileInput(String name)
  - If failure then throw a FileNotFoundException

```
public Map<String,String> readFromFile(Context context){
     Map<String,String> temp = null;

     try{
       inByteStream = context.openFileInput(FILENAME);
            OIStream = new ObjectInputStream(inByteStream);

       temp = (Map<String,String>)OIStream.readObject();

       inByteStream.close();
       OIStream.close();
     }
      catch(Exception e){...}

     return temp;
     }
```

Write to file

- Open a File for output
  - Context.openFileOutput(String name,int mode)
  - If failure then a new File is created
  - Append mode: to add data to file

```
public void writeToFile(Map<String,String> times, Context context){

  try{
      outByteStream = context.openFileOutput(FILENAME, Context.MODE_PRIVATE);
      OOStream = new ObjectOutputStream(outByteStream);
      OOStream.writeObject(times);
      outByteStream.close();
      OOStream.close();
     }
  catch(Exception e){...}

}
```

Write file to SDCard

- To get permission for SDCard r/w in AndroidManifest.xml:

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

SDCard read/write

- Need a SD Card, (obviously ☺)

```
if(Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED))
{
  File sdCardDir = Environment.getExternalStorageDirectory();
  File saveFile = new File(sdCardDir, "stuff.txt");
  FileOutputStream outStream = new FileOutputStream(saveFile);

  // Same approach as before, once you have a FileOutputStream and/or
  // FileInputStream reference...
  ...

  outStream.close();
}
```

Using ContentProviders

Content Provider

- a content provider is a specialized type of datastore that exposes standardized ways to retrieve and manipulate the stored data.
- Apps can expose their data layer through a Content Provider, identified by a URI.
- Some native apps provide Content Providers
- Your apps can provide Content Providers

Using ContentProvider to share data

- Content Providers are the Android platforms way of sharing information between multiple applications through its ContentResolver interface.
- Each application has access to the SQLite database to maintain their information and this cannot be shared with another application.

```
public class PersonContentProvider extends ContentProvider{
   public boolean onCreate()
   public Uri insert(Uri uri, ContentValues values)
   public int delete(Uri uri, String selection, String[]
      selectionArgs)
   public int update(Uri uri, ContentValues values, String
    selection, String[] selectionArgs)
   public Cursor query(Uri uri, String[] projection, String
    selection, String[] selectionArgs, String
sortOrder)
   public String getType(Uri uri)}
```

Addition to the AndroidManifest.xml

- Add the following user permission tag

  <uses-permission android:name="android.permission.READ_CONTACTS" />
- To give your application access to the contacts information.

```
<manifest .... >
    <application android:icon="@drawable/icon"
    android:label="@string/app_name">
     <provider android:name=".PersonContentProvider"
    android:authorities="ie.wit.provider.personprovider"/>
    </application>
</manifest>
```

Using Bundles

The Bundle Class (Saving)

- Override onSaveInstanceState
  - And pass the Bundle to the superclass method

```
protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        outState.putBlah(someData);
}
```

- Called
  - When user rotates screen
  - When user changes language
  - When app is hidden and Android needs the memory
- Not called
  - When user hits Back button
- Note
  - Superclass method automatically stores state of GUI widgets (EditText data, CheckBox state, etc.)

Bundle : Restoring Data

- Override onRestoreInstanceState
  - Pass Bundle to superclass method
  - Look for data by name, check for null, use the data

```
protected void onRestoreInstanceState(Bundle savedInstanceState) {
      super.onRestoreInstanceState(savedInstanceState);
      SomeType data = savedInstanceState.getBlah(key);
      if (data != null) { doSomethingWith(data); }
}
```

- Called
  - Any time app is restarted after onSaveInstanceState
- Note
  - The same Bundle is passed to onCreate.
  - Superclass method automatically restores widget state

The Bundle Class: Details

- Putting data in a Bundle
  - putBoolean, putBooleanArray, putDouble, putDoubleArray, putString, putStringArray, etc.
    - These all take keys and values as arguments. The keys must be Strings. The values must be of the standard types (int, double, etc.) or array of them.
  - putSerializable, putParceleable
    - Lets you store custom objects. Note that ArrayList and most other builtin Java types are already Serializable
- Retrieving data from a Bundle
  - getBoolean, getBooleanArray, getDouble, getDoubleArray, getString, getStringArray, etc.
    - No typecast required on retrieval. Numbers are 0 if no match.
  - **getSerializable, getParceleable**
    - Typecast required on retrieval. Values are null if no match.

Bundle Summary

- Save data in onSaveInstanceState
  - Can put individual pieces of data in the Bundle, or can add a composite data structure.
  - Custom classes must implement Serializable or Parceleable
- Load data in onRestoreInstanceState or in onCreate
  - Look in Bundle for property of given name
  - For Object types, check for null
  - For number types, check for 0 (zero)

Note: Preventing Screen Rotations

- Issue
  - Screen rotations usually require a new layout
  - They also cause the app to be shutdown and restarted
    - Handling this is the topic of this lecture
- Problem
  - What if you do not have landscape layout?
  - Or have not yet handled shutdown and restart?
- Solution
  - Put an entry in AndroidManifest.xml saying that app runs only in portrait mode (or only in landscape mode).

```
<activity android:name=".YourActivity"
          android:label="@string/app_name"
          android:screenOrientation="portrait">
```