CMR Institute of Technology, Bangalore
DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING
III – INTERNAL ASSESSMENT

Semester: 6-CBCS 2017 /2018 Date: 02 Aug 2021
Subject: MOBILE APPLICATION DEVELOPMENT (18CS651/17CS661/15CS661)
Faculty: Mr Sudhakar K N
Time: 01:00 PM – 02:30 PM Max Marks: 50

Scheme & Solution
ANSWER_ANY 5 Question(s)

Marks CO BT/CL

1a. Define Content Provider. Explain with an example sharing the data between application using Content Provider.

[7.0] 1 [2]

Scheme: Definition 3M + Process 4M

Solution

A ContentProvider is a component that interacts with a repository. The app doesn't need to know where or how the data is stored, formatted, or accessed.

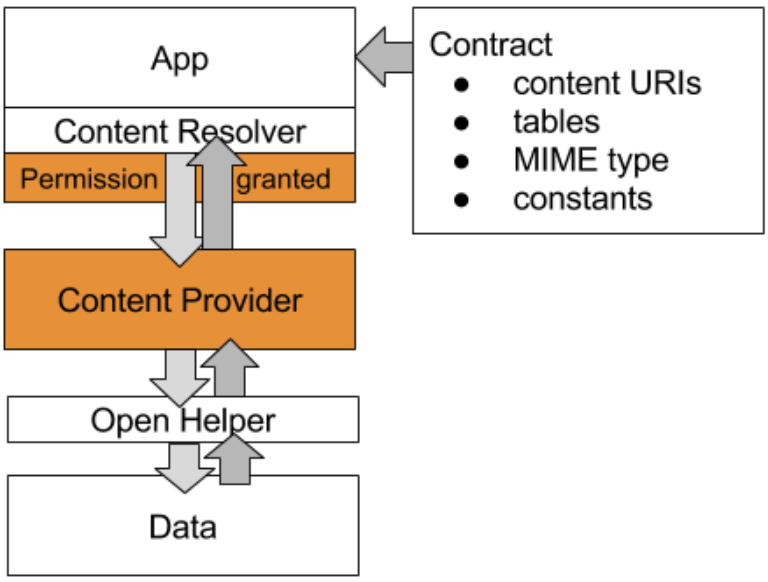
A content provider:

Separates data from the app interface code

Provides a standard way of accessing the data

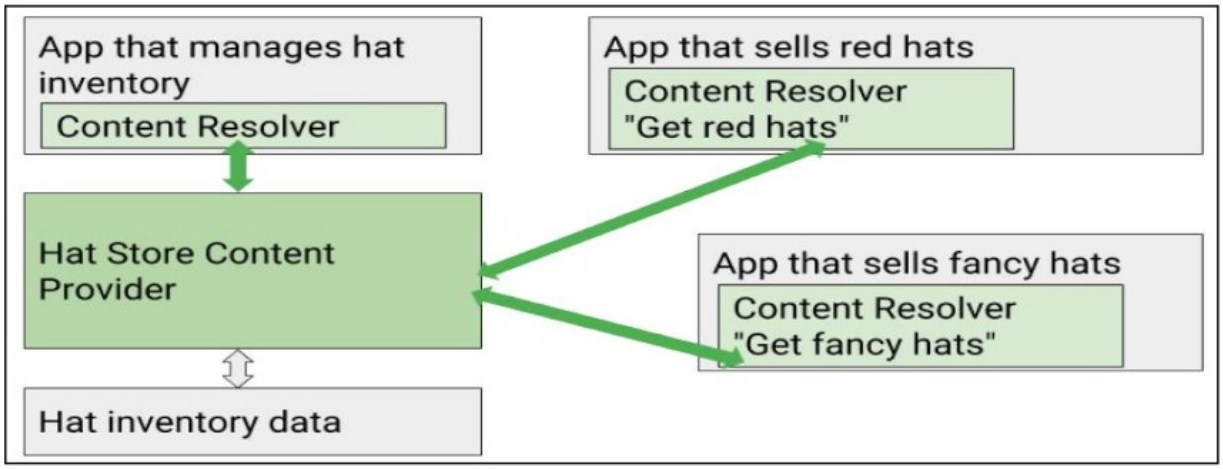
Makes it possible for apps to share data with other apps

Is agnostic to the repository, which could by a database, a file system, or the cloud.



Example of an app sharing data using a Content Provider:

Consider an app that keeps an inventory of hats and makes it available to other apps that want to sell hats. The app that owns the data manages the inventory, but does not have a customer- facing



interface. Two apps, one that sells red hats and one that sells fancy hats, access the inventory repository, and each fetch data relevant for their shopping apps.

1b. List the advantages of Content Provider.

[3.0] 1 [1]

Scheme: Advantages 3M

Solution

Advantages:

- Securely make data available to other Apps
- Manage access permissions to App database
- Store data or develop backend independently from UI
- Standardized way of accessing data
- Required to work with cursorLoaders

2a. List the various options provided by the android system to store the application data persistently. Explain each of these based on usage.

[6.0] 1 [2]

Scheme: Options 6M

Solution

Android provides several options for you to save persistent application data. Your data storage options are the following:

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.
- Cloud Backup—Backing up app and user data in the cloud.
- Content providers—Store data privately and make them available publicly.
- Firebase realtime database—Store and sync data with a NoSQL cloud database. Data

is synced across all clients in real time, and remains available when your app goes offline.

2b. Differentiate between the two file storage areas.

[4.0] 1 [2]

Scheme: Differences 4M

Solution

Internal storage	External storage
Always available.	Not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device.
Only your app can access files. Specifically, your app's internal storage directory is specified by your app's package name in a special location of the Android file system. Other apps cannot browse your internal directories and do not have read or write access unless you explicitly set the files to be readable or writable.	World-readable. Any app can read.
When the user uninstalls your app, the system removes all your app's files from internal storage.	When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from getExternalFilesDir().
Internal storage is best when you want to be sure that neither the user nor other apps can access your files.	External storage is the best place for files that don't require access restrictions and for files that you want to share with other apps or allow the user to access with a computer.

3. Demonstrate with code snippet writing and reading data from the file of Internal Storage of an android device.

[10.0] 1 [3]

Scheme: Writing 5M + Reading 5M

Solution

Write File:

```
private void writeToFile(String data,Context context) {
   try {
      OutputStreamWriter outputStreamWriter = new
OutputStreamWriter(context.openFileOutput("config.txt", Context.MODE_PRIVATE));
      outputStreamWriter.write(data);
      outputStreamWriter.close();
   }
   catch (IOException e) {
      Log.e("Exception", "File write failed: " + e.toString());
   }
}
```

Read File:

```
private String readFromFile(Context context) {

   String ret = "";
```

```
    try {
        InputStream inputStream = context.openFileInput("config.txt");

        if ( inputStream != null ) {
            InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
            BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
            String receiveString = "";
            StringBuilder stringBuilder = new StringBuilder();

            while ( (receiveString = bufferedReader.readLine()) != null ) {
                stringBuilder.append("\n").append(receiveString);
            }

            inputStream.close();
            ret = stringBuilder.toString();
        }
    }
    catch (FileNotFoundException e) {
        Log.e("login activity", "File not found: " + e.toString());
    } catch (IOException e) {
        Log.e("login activity", "Can not read file: " + e.toString());
    }

    return ret;
}
```

4a. Define Shared Preferences. Differentiate between Shared Preferences and Saved Instance State.

[5.0] 1 [2]

Scheme: Definition 2M + Differences 3M

Solution

Shared preferences

Using shared preferences is a way to read and write key-value pairs of information persistently to and from a file.

Files:

Android uses a file system that's similar to disk-based file systems on other platforms such as

Linux. File-based operations the java.io package. All Android devices have two file storage areas: "internal" and "external" storage. These names come from the early days of Android, when most devices offered built- in non-volatile memory (internal storage), plus a removable storage medium such as a micro SD card (external storage).

Shared preferences Vs Saved instance state.

Shared Preferences	Saved instance state
Persists across user sessions, even if your app is killed and restarted, or the device is rebooted.	Preserves state data across activity instances in the same user session.
Data that should be remembered across sessions, such as a user's preferred settings or their game score.	Data that should not be remembered across sessions, such as the currently selected tab, or any current state of an activity.
Small number of key/value pairs.	Small number of key/value pairs.
Data is private to the application.	Data is private to the application.
Common use is to store user preferences.	Common us is to recreate state after the device has been rotated.

4b. Demonstrate with Code Snippet the usage of Shared Preferences.

[5.0] 1 [3]

Scheme: Code + explanation: 5M

Solution

Internal storage

You don't need any permissions to save files on the internal storage. Your application always has permission to read and write files in its internal storage directory.
You can create files in two different directories:

- Permanent storage: getFilesDir()
- Temporary storage: getCacheDir() .

Recommended for small, temporary files totalling less than 1MB. Note that the system may delete temporary files if it runs low on memory. To create a new file in one of these directories, you can use the File() constructor, passing the File provided by one of the above methods that specifies your internal storage directory.

For example:

File file = new File(context.getFilesDir(), filename);

Alternatively, you can call openFileOutput() to get a FileOutputStream that writes to a file in your internal directory. For example, here's how to write some text to a file:

String filename = "myfile";

```
        String string = "Hello world!";
FileOutputStream outputStream;
try {
 outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
 outputStream.write(string.getBytes());
 outputSteam.close();
}catch (Exception e)
{
e.printStackTrace();
}
```

5a. Define and List the Characteristics of SQLite database. Justify why SQLite database is better option of Storing the data over other Databases.

[6.0] 1 [2]

Scheme: Characteristics 3M + Justification 3M

Solution

SQLite:

SQLite is a software library that implements SQL database engine that is:

- self-contained (requires no other components)
- serverless (requires no server backend)
- zero-configuration (does not need to be configured for your application)
- transactional (changes within a single transaction in SQLite either occur completely or not at all)

SQLite Database

Of the many storage options discussed, using a SQLite database is one of the most versatile, and straightforward to implement.

An SQLite database is a good storage solution when you have structured data that you need to store persistently and access, search, and change frequently.

You can use the database as the primary storage for user or app data, or you can use it to cache and make available data fetched from the cloud.

If you can represent your data as rows and columns, consider a SQLite database.

Content providers, which will be introduced in a later chapter, work excellently with SQLite databases

When you use an SQLite database, represented as an SQLiteDatabase object, all interactions with the database are through an instance of the SQLiteOpenHelper class which executes your requests and manages your database for you. Your app should only interact with the SQLiteOpenHelper.

5b. With an Example explain the queries for Android SQLite.

[4.0] 1 [1]

Scheme: Elaboration 4M

Solution

Query language

You use a special SQL query language to interact with the database. Queries can be very complex, but the basic operations are

inserting rows
deleting rows
updating values in rows

Android, the database object provides convenient methods for inserting, deleting, and updating the database. You only need to understand SQL for retrieving data.

Query structure

A SQL query is highly structured and contains the following basic parts:
 SELECT word, description FROM WORD_LIST_TABLE WHERE word="alpha"

Generic version of sample query:

- SELECT columns FROM table WHERE column="value"

  Parts:

- SELECT columns—select the columns to return. Use * to return all columns.
- FROM table—specify the table from which to get results.
- WHERE—keyword for conditions that have to be met.
- column="value"—the condition that has to be met. common operators: =, LIKE, <, >
- AND, OR—connect multiple conditions with logic operators.
- ORDER BY—omit for default order, or specify ASC for ascending, DESC for retrieving rows that meet given criteria descending. LIMIT is a very useful keyword if you want to only get a limited number of results.

Sample queries

1	SELECT * FROM WORD_LIST_TABLE	Get the whole table.
2	SELECT word, definition FROM WORD_LIST_TABLE WHERE _id > 2	Returns `[["alpha", "particle"]]`
3	SELECT _id FROM WORD_LIST_TABLE WHERE word="alpha" AND definition LIKE "%art%"	Return the id of the word alpha with the substring "art" in the definition. `[["3"]]`
4	SELECT * FROM WORD_LIST_TABLE ORDER BY word DESC LIMIT 1	Sort in reverse and get the first item. This gives you the last item per sort order. Sorting is by the first column, in this case, the _id. `[["3", "alpha", "particle"]]`
5	SELECT * FROM WORD_LIST_TABLE LIMIT 2,1	Returns 1 item starting at position 2. Position counting starts at 1 (not zero!). Returns `[["2", "beta", "second letter"]]`

6. What is a Cursors? Explain processing of Cursors with an example.

[10.0] 1 [2]

Scheme: Description 3M + Explanation 7M

Solution

Cursors:

Queries always return a Cursor object. A Cursor is an object interface that provides random read-write access to the result set returned by a database query. It points to the first element in the result of the query. A cursor is a pointer into a row of structured data. You can think of it as a pointer to table rows.

The Cursor class provides methods for moving the cursor through that structure, and methods to get the data from the columns of each row. When a method returns a Cursor object, you iterate over the result, extract the data, do something with the data, and finally close the cursor to release the memory.

The Cursor class has a number of subclasses that implement cursors for specific types of data.

- SQLiteCursor exposes results from a query on a SQLiteDatabase. SQLiteCursor is not internally synchronized, so code using a SQLiteCursor from multiple threads should perform its own synchronization when using the SQLiteCursor.
- MatrixCursor is an all-rounder, a mutable cursor implementation backed by an array of objects that automatically expands internal capacity as needed. Some common operations on cursor are:
- getCount() returns the number of rows in the cursor.
- getColumnNames() returns a string array holding the names of all of the columns in the result set in the order in which they were listed in the result.
- getPosition() returns the current position of the cursor in the row set.

- Getters are available for specific data types, such as getString(int column) and getInt(int column).
- Operations such as moveToFirst() and moveToNext() move the cursor.
- close() releases all resources and makes the cursor completely invalid. Remember to call close to free resources!

Processing cursors:

When a method call returns a cursor, you iterate over the result, extract the data, do something with the data, and finally, you must close the cursor to release the memory. Failing to do so can crash your app when it runs out of memory.

The cursor starts before the first result row, so on the first iteration you move the cursor to the first result if it exists. If the cursor is empty, or the last row has already been processed, then the loop exits. Don't forget to close the cursor once you're done with it. (This cannot be repeated too often.)

```
// Perform a query and store the result in a Cursor Cursor cursor = db.rawQuery(...);
try {
while (cursor.moveToNext()) {

// Do something with the data }
} finally {
cursor.close();}
```