Project 4 Hints

NOTE: These are only general hints – they should not be regarded as complete directions on how to implement the State Capitals Quiz app.

1. Design your database

Design a database schema to hold all quiz questions (one per state), and quizzes given to the user, each including the date of the quiz, the 6 questions that were asked, and the quiz result. You will need at least 2 tables: one for the questions and one for the quizzes. If you want, you may store user's answer to each of the questions, as well, but this will likely require a bit more involved database schema (discussed below).

1.1. Design a table to store the quiz questions, one per row. The table should include columns for the question identifier (auto-incremented primary key), and for the names of the state, capital city, and the two additional cities.

Additionally, you may include the year when the state achieved the US statehood, the year when the city became the capital of the state, and its rank in the state, in terms of its population (CSCI 6060 will have to do this). This additional information may add some "extra" information when asking a question to the user, but giving the population ran may. However, if you are a CSCI 4060 student team, this additional data is not necessary for this project and you may simply skip it.

1.2. Design a table to store quizzes. A simple solution would be to create a table with columns for the quiz identifier (auto-incremented primary key), the quiz date, and 6 columns for the questions, each being a foreign key to the question table. Of course, the questions should be randomly selected for each quiz. One more column should be there to store the quiz result (i.e., the number of correct answers). You may include 6 additional columns to store individual user scores, for each state-question in the quiz, and perhaps one more, to record how many answers a user has given so far, in order to restart a quiz.

A much better solution would be to have a table for questions (as above) and one for the quizzes, but only for the quiz identifier (auto-incremented primary key), the quiz date, and the quiz result. Here, you would also need to design a relationship table (similar to the isMemberOf association between the Person and the Club, as discussed in the class lecture). This relationship table would connect the question table to the quiz table (using two foreign keys). The advantage of this solution is that you could easily change the number of questions in a quiz, and also you would be able to store the user's answer to each quiz question (in this table). Additionally, you could store here how many answers a user has given so far, in order to restart a quiz.

1.3. You may use one of the stand-alone programs to manage SQLite database, such as SQLiteStudio (www.sqlitestudio.pl), DB Browser for SQLite (www.sqlitebrowser.org), or SQLiteManager (www.sqlabs.com/sqlitemanager.php), or any other similar program. These are point-and-click systems, which you can use to define SQLite database tables, populate them with initial data, and test some queries that you will later need to implement your app.

1.4. Whether you use an SQLite database manager program or not, you should have a very good idea what tables you need to create and how to store and retrieve data for the quiz questions and quizzes.

2. Start a new Android Studio project for your app

2.1. Start with a simple main activity, but you may forgo its details until later.

2.2. Create a class extending the SQLiteOpenHelper. Define the necessary constants for the table create statements and override the onCreate and onUpgrade methods to create and upgrade the tables. You may base your implementation on the example presented in class, which is available on eLC.

2.3. You may want to create Java classes to represent questions and quizzes. You may also want to create a class similar to JobLeadsData from the example on eLC, which would be a convenient way to open and close your database, and to create and store quizzes, or perhaps their partial progress. However, this class is not essential and it is possible to implement your app without it.

2.4. Implement the initial population of the questions table with data from the CSV file. You may base your solution on an example of a CSV-reading app, available on eLC.

Alternatively, you may create the database, create the necessary tables, and populate the database using an external SQLite manager program (one of the mentioned above), or even a stand-alone program. Once you create the database this way, you should copy the database file (it is just one file) to your app's assets folder in Android Studio. However, if you follow this route, you will need to write code in your app to copy your database from the assets folder to the databases folder at the start of your app (just for the first time only!). You will have to learn how to do this, but it is not a difficult task (make sure that you use methods `context.getAssets()` and `context.getDatabasePath()` and *not* the absolute pathnames to assure app's portability!).

2.5. Create a new activity to start a quiz (it should be started somehow from the main activity). In the Java controller for this activity create an instance of your SQLite helper class (or the Data class, if you have it).

2.6. In most of you classes, you should include logging of debugging messages to trace your app later.

2.7. Run you app and check if the database has been created. If you have included debugging messages, check you logcat to see if all of the expected events actually happened.

2.8. Go back to your start the quiz activity and add the code to actually create a new quiz and save it in the database.

2.9. All of the database interactions should be done asynchronously. A good way to do this is by extending the `AsyncTask` class and overriding the `doInBackground` and `onPostExecute` methods. You may to have such a class for each of your activities that needs to save/restore data from the database. The JobTracker app, which is available on eLC. In addition, the initial population of the database (unless you decided to pre-load the database ahead of time) should also be performed asynchronously.

3. Implement the rest of your app

3.1. Depending on your app design, you should implement the actual running of a quiz. You should implement swiping. You do not have to implement blocking the user from going back to the already answered questions. However, you will have to implement accepting and storing the user's answer. Test your app by running a couple of complete quizzes.

3.2. Implement the activity for viewing past quiz results. Test your app again.

3.3. Make sure that you have the code necessary to restart a partial quiz (you will need to save the partial quiz to the database and then restore it at a suitable point in your app's lifecycle). Test save/restore operation by suspending a quiz, switching to a different app in the emulator (e.g., the Chrome browser), and then switching back the suspended quiz.

4. Thoroughly test your app and correct any problems and submit your project