CS 522—Fall 2022
Mobile Systems and Applications
Assignment Three—Databases and Content Providers

In this assignment, you will extend the chat server app from the previous assignment. The previous app just saves messages received in an array in the activity UI. If the user navigates away from the activity, and then returns to it, there is a good chance that the messages already received will have been lost, due to the way that Android manages resources in the activity life cycle. For example, in the example of the "treasure hunt" from the previous assignment, if a team leader navigates away from the chat app and the low memory killer kills the app before the team leader navigates back to it, all the messages received so far are lost. We will in this assignment persist both messages received, and information about people that xhave sent us messages, to a persistent database.

Another motivation is that we are so far violating one of the most basic tenets of Android programming, that no blocking operations should be performed on the main UI thread. In this assignment, we will continue to do network communication on the main thread, as well as insert records into the database on the main thread. However, we will use cursor loaders and loader managers to perform querying of the database on a background thread. In the next assignment, we will see how to do this using the Room database programming framework, and the lifecycle-aware observer pattern.

Your app should target Android API Level 33. You should follow these guidelines for your implementation.

First, define a subpackage of your application, called `contracts`. Define classes in this package called `MessageContract` and `PeerContract`, that define content URIs and content paths, as well as content types. They should also define (as final static `String` constants) the names of the columns in your database. For each operation, define a type-specific read operation that reads the value of the column from a cursor, and a type-specific write operation that adds the value in a column to a `ContentValues` object. For example:

```
public static final String MESSAGE_TEXT = "message-text";

public static String getMessageText(Cursor cursor) {
   return cursor.getString(cursor.getColumnIndexOrThrow(MESSAGE_TEXT));
}

public static void putMessageText(ContentValues values, String text) {
   values.put(MESSAGE_TEXT, text);
}
```

The base class `BaseContract` defines some operations common to all contract classes.

Second, extend the `Message` and `Peer` entity classes that you defined in the previous assignment, with a constructor that initializes the fields from an input cursor, and a method that writes the fields of the entity to a `ContentValues` object (for insertion into a database):

```
public interface Persistable {
   public void writeToProvider(ContentValues out);
}

public class Message implements Parcelable, Persistable {
   ...
   public Message(Cursor cursor) {
      this.messageText = MessageContract.getMessageText(cursor);
      ...
   }
   ...
   public void writeToProvider(ContentValues values) {
      MessageContract.putMessageText(values);
      ...
   }
}
```

Third, in your main activity where you display the messages received in a `ListView`, use the `SimpleCursorAdapter` class to connect the cursor resulting from a query to the list view. There are two constructors for `SimpleCursorAdapter`. You should use the form of the constructor that accepts flags for configuring the requerying behavior of the adapter. If you leave this as zero, then by default the adapter will not try to requery the database on the main thread:

```
   public SimpleCursorAdapter(Context context, int layout, Cursor c,
                              String[] from, int [] to, int flags);
```

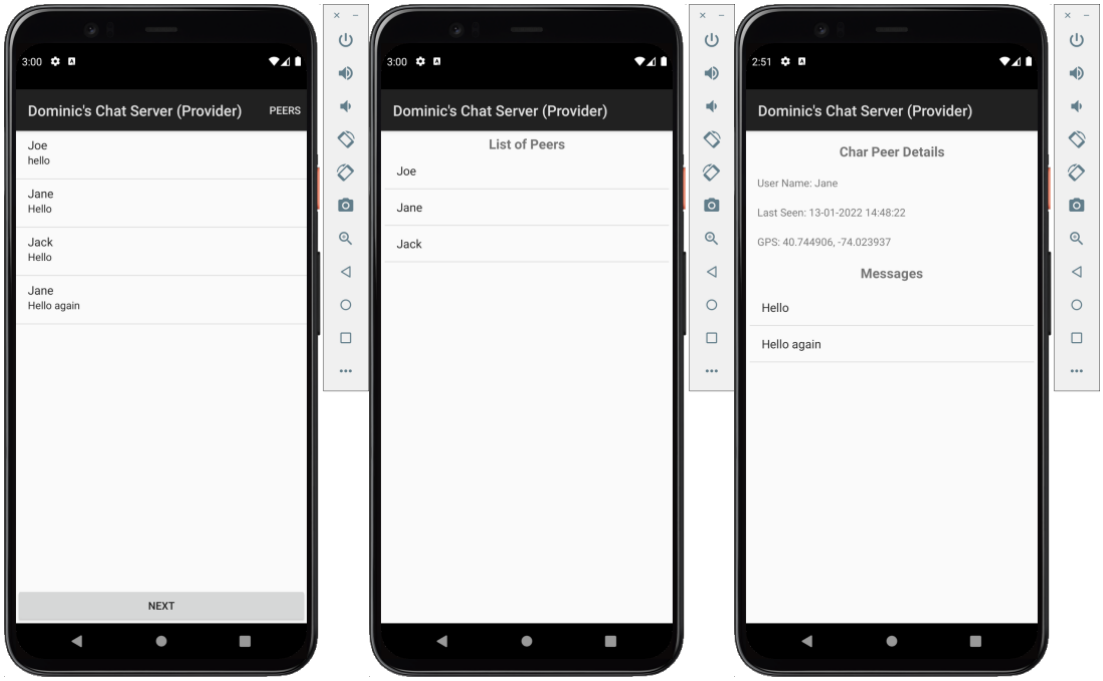
You should use the layout resource `R.layout.message` as the layout resource for each row in your list view. This defines a layout of two text views, one above the other, for each row. Display the title and authors of each book[1]:

```
   String[] from = new String[] { MessageContract.SENDER,
                                  MessageContract.MESSAGE_TEXT };
   int[] to = new int[] { R.id.sender,
                          R.id.message };
```

Fourth, use cursor loaders and the loader manager to query the provider. The `ChatServerActivity` activity uses the loader manager to query the content provider for a list of all messages (Use `MessageContract.CONTENT_URI` to identify the content in the loader). `ViewPeersActivity` uses the loader manager to query the content provider for

[1] In the other activities, where you use `android.R.layout.simple_list_item_1` as the row layout in the listview, the single textview field has id `android.R.id.text1`.

a list of all chat peers messages (Use `PeerContract.CONTENT_URI` to identify the content in the loader). When it launches `ViewPeerActivity` to view the details for a chat peer, it should pass a peer entity object (as a `Parcelable` extra) to the subactivity; the latter will then need to use a loader manager to query the content provider for a list of all messages, filtered for that particular sender. *Define all callbacks for the loader manager as methods in your activities, not in separate callback objects.*



Note that your apps should **never** use the `startManagingCursor` or `managedQuery` operations, nor should they provide an initial cursor for the constructor for `SimpleCursorAdapter` (How would you get it?). However, it is all right in general to use the `SimpleCursorAdapter` class, using the second constructor that provides a flag that indicates that the query should not be managed by the activity. Just do not use the first, deprecated constructor that causes queries to be managed on the UI thread.

Fifth, implement the database in a content provider. Define a single content provider with two tables, visible to the app, and distinguished by their URIs that have the same authority but different content paths. One table stores the messages that have been received, while the second table stores information about the peers from whom we have received messages. Define a subpackage called `providers`. In this class, define the content provider class `ChatProvider`. This class defines:

1. Useful string literals such as such the SQL commands to create the database), `DATABASE_NAME` (the name of the file containing the database), `MESSAGE_TABLE` (the name of the table containing message information) and `PEER_TABLE` (the name of the table containing peer information), and `DATABASE_VERSION` (an integer that is used for versioning your database).
2. A private static inner class called `DatabaseHelper` that extends `SQLiteOpenHelper` with your logic for creating the database and upgrading

where necessary. The content provider, on startup, instantiates this helper class in order to obtain a reference to the database.

3. The content provider operations for inserting a message or upserting a chat peer, and for querying for all messages and all peers. The other cases for these operations, and the update and delete operations, are unnecessary for this application. When you receive a message, extract the name of the sender and other metadata on the message (the timestamp and GPS coordinates), and update the database record for this peer, or insert a record if this is the first time we have heard from this peer.
4. *Note that you will need, in the provider query operation for messages, to register the cursor as a content observer for changes in the content (messages), identified by a URI. When a message is inserted, in the provider insert operation, notify any observers of the content. This ensures that the list of messages in the UI is updated when a new message is received*. See the lecture material for more about this.

Your solution should consist of two apps, `Chat-Client-Oneway` and `Chat-Server-Oneway`. The client app is unchanged from the previous assignment. The chat server should follow the guidelines described above above: Use cursor loaders and the loader manager for querying all messages received so far, for querying all peers from whom we have received messages, and for querying messages received from a particular chat peer. It is okay to insert or "upsert" peer and message information into the database, when a message is retrieved, on the main thread.

Submitting Your Assignment

Once you have your code working, please follow these instructions for submitting your assignment:

1. You should submit two Android Studio projects: chat client and chat server.
2. Create a zip archive file, named after you, containing a directory with your name. E.g. if your name is Humphrey Bogart, then name the directory Humphrey_Bogart.
3. In that directory you should provide the Android Studio projects for your apps.
4. You should also provide APK files for your compiled projects.
5. Also include in the directory a completed rubric for your submission.

In addition, record short mpeg videos of demonstrations of your assignment working. Make sure that your name appears at the beginning of each video. For example, put your name in the user interface of the app. *Do not provide private information such as your email or cwid in the video*. The videos should demonstrate running the app, exiting the app, then rerunning the app and demonstrating that the saved state from the previous run (messages for the chat app) are still present when the app is restarted. Make sure that you send messages from several peers to the chat server. Be careful of any "free" apps that you download to do the recording, there are known cases of such apps containing Trojan horses, including key loggers.

Your solution should be uploaded via the Canvas classroom. Your solution should consist of a zip archive with one folder, identified by your name. Within that folder, you

should have two Android Studio projects, for the apps you have built. You should also provide videos demonstrating the working of your assignment. **Make sure that your video shows the server app being launched from the Android Studio project based on content providers and loaders, and that you show the data persisted even after you leave the app and return to it.**