CS 522—Fall 2020
Mobile Systems and Applications
Assignment Nine—Cloud Chat Chap

In this assignment, you will complete a cloud-based chat app that you started on the previous assignment, where clients exchange chat messages through a Web service. You are given a simple chat server that apps will communicate with via HTTP. You should use the `HttpURLConnection` class to implement your Web service client, following the software architecture described in class.

The main user interface for your app presents a screen with a text box for entering a message to be sent, and a "send" button. The rest of the screen is a list view that for now will just show the messages posted by this app. Provide a settings screen where the server URI and this chat instance's client name can be viewed. The user chat name can only be set when registering with the server.

The interface to the cloud service, for the frontend activities, should be a *service helper* class. This is a POJO (plain old Java object) that is created by an activity upon its own creation, and encapsulates the logic for performing Web service calls. There are two operations that this helper class supports:

- Register with the cloud chat service.
- Post a message: Add a chat message to a request queue, to be uploaded to the server. Unlike the previous assignment, this upload will not be done immediately. Instead the message will be added to the local chat message database, and this database periodically synchronized with the server in the background. This synchronization should also refresh the list of peers registered with the chat service.

All of these operations are asynchronous, since you cannot block on the main thread.

When a message is generated, it is added to the content provider with its message sequence number set to zero. The sequence number is finally set to a non-zero value when the message is eventually uploaded to the chat server; see the protocol below. The flag is always non-zero for messages downloaded from the chat server. Note that you cannot use this message sequence number as a primary key in your database, because its value is set by the chat server, but you will have to add local messages to the content provider immediately, without communicating with the server.

The content provider also stores a list of the other clients registered with the chat service. This list, and the list of chat messages, are periodically refreshed by synchronizing with the chat service. For simplicity, you can assume that a complete list of chat clients is downloaded on each request. However you should be more intelligent with downloading of new chat messages. Assuming that the chat service assigns a unique sequence number to each chat message it receives, the app can retrieve the sequence number of the most recent chat message that it has received from the content provider, and provide this to the chat server. The chat server will respond with all chat messages that it has received since that last chat message seen by the client. This synchronization should be done at the

same time that the client is uploading messages to the server. So the protocol for synchronizing with the chat server, once the client is registered, is as follows:

1. The client uploads all messages stored in its content provider that have not yet been uploaded to the server. It also provides the sequence number of the last message it has received (along with its own UUID and service id to identify itself)
2. The server adds these messages to its own database, assigning each message a unique sequence number.
3. The server responds with a list of all of the registered clients, and a list of the messages that it has received since it last synchronized with the client. The client updates the messages it has just uploaded with their sequence numbers, and inserts the new messages (from other clients) received from the chat server. It also updates the list of chat clients with the list received from the server, inserting new client records and updating existing client records with client metadata[1]. Assuming you are using server-assigned sender identifiers as primary keys for clients in your content provider, you will be able to maintain the correct relationships between clients and messages in your content provider.

As before, the service helper class should use an intent service (`RequestService`, subclassing `IntentService`) to submit request messages to the chat server. This ensures that communication with the chat server is done on a background thread. Single-threaded execution of requests will be sufficient, and greatly simplify things. There are three forms of request messages: `RegisterRequest`, `PostMessageRequest` and `SynchronizeRequest`. Two of these messages are sent by the UI using the service helper, as before. The third is triggered by a repeating alarm. Define three concrete subclasses of an abstract base class, `Request`, for each of these cases. The basic interface for the `Request` class is as before:

```
public abstract class Request implements Parcelable {
   public long senderId
   public UUID appID;
   public Date timestamp;
   public Double latitude;
   public Double longitude;
   ...
}
```

The time and location information is used to record the last known location of the client at the server. For the registration and message posting requests, the service helper will create a request object and attach it to the intent that fires the request service (hence the necessity to implement the `Parcelable` interface).

---

[1] Rather than querying the content provider to see if a record is present (for a message or peer), it is better to have the content provider insert operation actually perform an "upsert" (insert or update). See here for more suggestions on how to implement this in the content provider. A variant on this idea is to do a "bulk insert" of the downloaded messages, which has the benefit of encompassing their insertion in a transaction, but this requires building a list of the insertion records in memory, which we are trying to avoid by streaming the data. The fundamental problem here is that Android's content provider API has no support for bracketing transactions outside the content provider.

The business logic for processing these requests in the app should be defined in a class called `RequestProcessor`. This is again a POJO class, created by the service class, which then invokes the business logic as represented by three methods, one for each form of request:

```
public class RequestProcessor {
   public Response perform(RegisterRequest request) { … }
   public Response perform(PostMessageRequest request) { … }
   public Response perform(SynchronizeRequest request) { … }
}
```

The request processor in turn will use an implementation class, `RestMethod`, that encapsulates the logic for performing Web service requests. See the lecture materials for examples of code that you can use for this class. This class should use `HttpURLConnection` for all Web service requests. You should not rely on any third-party libraries for managing your Web service requests. The public API for this class has the form[2]:

```
public class RestMethod {
   ... // See lectures.
   public Response perform(RegisterRequest request) { … }
   public StreamingResponse perform(SynchronizeRequest request,
                                    StreamingOutput out) { … }
}
```

There are two forms of requests:

1. A *fixed-length request* sends the request data in HTTP request headers, as part of the URI (e.g. as query parameters) and/or in a JSON output entity body. Use this form of request for the registration request. The format of the request should be as for the previous assignment.
2. A *streaming request* is more appropriate for the case where there is potentially a lot of data to be uploaded or downloaded, so building a JSON string in memory is not advisable. This is the case for the synchronization request. Stream both the chat messages to be uploaded to the service, and the lists of clients and messages to be downloaded. Use the `JsonWriter` class to write the uploaded messages, and use `JsonReader` to read the streaming response from the server.

For streaming requests, the streaming is done in the request processor, **not** in the REST implementation (`RestMethod`). The latter just handles the mechanics of managing the network connection with the server. Therefore the input to the synchronization request has a second argument of this type:

```
public interface StreamingOutput {
    public void write(OutputStream out);
```

---

[2] There is no case for `PostMessageRequest` in `RestMethod` in this assignment, because the message is logged in the content provider and asynchronously uploaded by an alarm-driven service. You can leave the case in the API for `PostMessageRequest`, for the sake of making this assignment upward-compatible with the previous assignment, even though `RequestProcessor` will never execute it.

```
}
```

and the response from the synchronization request has this type:

```
public class StreamingResponse {
   public HttpURLConnection connection;
   public Response response;
}
```

Unlike the other `RestMethod` operations, that perform all necessary I/O on the network connection, and then close this connection before returning to the request processor, the streaming request operation (for synchronization) executes the request with HTTP request headers and URI alone set, and then returns the open connection to the request processor. The latter can send data to the server by writing to the connection output stream (layering a `JsonWriter` stream over this) and receive data by reading from the connection input stream (layering a `JsonReader` over this). It is the responsibility of the request processor in this case to close the connection when done!

For synchronization, your service should transparently handle the case where communication is not currently possible with the server, either because the device is not currently connected to the network or because communication with the server times out. The client-side information that needs to be persisted is already in the content provider: Those messages that have a sequence number of zero, indicating that they have not yet been uploaded, and the maximum sequence number for the messages so far downloaded from the server. The latter can be obtained by querying the local database.

One way to upload messages is to rely on the user to do it manually, but this is obviously unsatisfactory. If the server is not available, the client will have to manually retry later. When you send an email, your email client should not rely on you to force resending if the network is not available when the email is initially sent. Therefore use an alarm, and the `AlarmManager` service, to periodically synchronize messages (and clients) with the server. The alarm handler should perform synchronization, provided registration has succeeded, in order to refresh the app state with state on the server. You are provided with a helper class, `ServiceManager`, that contains code for managing this alarm. The alarm is enabled and disabled in the `onResume` and `onPause` methods of `ChatActivity`.

Running the Server App

You are provided with a server app, written using Java JAX-RS (Jersey). You can use it just by executing the jar file[3]. It takes several optional command line arguments:

- `--host` *`host-name`*: The name of the host the server is running on (default is the result of executing `InetAddress.getLocalHost().getCanonicalHostName()`).
- `--port` *`port-number`*: The port the server is binding to (default 8080).

[3] The server will also be running on a department machine. You will need to be on the campus VPN in order to access it. You can find more information about VPN setup here.

- `--bg true|false`: Whether the server is running in the background (it does not read a line of input to terminate, if running in the background; default false).
- `--log log-file-name`: Where to write the server log (default "server.log").

If you want to see the behavior of the server, without relying on your own code, you can use the curl command-line tool to send Web requests to the server. The following command will synchronize messages with the server:

```
curl -X POST -H "Content-Type: application/json" \
   -d @messages.json –H 'X-Latitude: ...' ... \
   'http://host-name:8080/chat/sender-id/sync?last-seq-num=0'
```

The query string parameter specifies the sequence number of the last message received by the client (The first message has a sequence number of 1). The file `messages.json` should contain messages to be uploaded, in JSON format. For example:

```
[
   {
      "chatroom" : "_default",
      "timestamp":...,"latitude":...,"longitude":...,
      "text" : "hello"
   },
   {
      "chatroom" : "_default",
      "timestamp":...,"latitude":...,"longitude":...,
      "text" : "is there anybody out there?"
   }
]
```

This will produce a JSON output of the form:

```
{"clients": [{"id":1, username:"joe",timestamp:...,latitude:...,
longitude:...}],
 "messages":
     [{"chatroom":"_default",timestamp:...,latitude:..., longitude:...,
       "seqnum":1,"sender":1,"text":"hello"},
      {"chatroom":"_default",timestamp:...,latitude:..., longitude:...,
       "seqnum":2,"sender":1,"text":"is there anybody out there?"}]}
```

The response includes a list of all clients registered with the service, and a list of messages uploaded to the service since the last time this client synchronized (including messages just uploaded by the client itself). The messages database should be updated with the downloaded message, by "upserting" the downloaded messages (updating the sequence numbers for messages that were uploaded from the current device[4]). The peer database should be updated with the client information downloaded from the server, by

[4] You can identify messages from the current device by the sender id, which refers to the current sender. You can match these downloaded messages with records in the local database using the timestamp (make sure to define a secondary index on the timestamp field, otherwise SQLite will do a linear search on the database to find a record with a particular timestamp.

"upserting" the downloaded client records (updating existing client records with metadata). Using sender identifiers as foreign keys to peer records in the local database will ensure that downloaded messages remain correctly linked to their sender records.

For debugging purposes, you can query for the messages that have been uploaded as follows:

```
curl -X GET ... 'http://localhost:8080/chat/messages'
```

Submitting Your Assignment

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

1. 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.
2. In that directory you should provide the Android Studio project for your app.
3. In addition, record mpeg or Quicktime videos of a demonstration of your assignment working. Make sure that your name appears at the beginning of the video. For example, put your name in the title of the app. *Do not provide private information such as your email or cwid in the video*.
4. For this assignment, you should demonstrate your app working against the running server at http://shadow.srcit.stevens-tech.edu:8080/chat. You will need to be on the Stevens network, using VPN if necessary, in order to access this server. Make sure that this is defined as the base URI for the server in the string resources. Use the debugging commands to show the messages on the server, at the beginning and end of your demo videos.

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 a single Android Studio project, for the app you have built. You should also provide videos demonstrating the working of your assignment. Your testing should demonstrate at least two devices registered at the server, messages being added, and messages from one device becoming visible at the other device (for both devices).