CS 522—Spring 2015
Mobile Systems and Applications
Assignment Seven—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. All of your projects for this submission should target Lollipop (Android 5.0.1).

Please note that the server provided for this assignment has a different API from that for the previous assignment.

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 showing messages received and their senders. Provide a settings screen where the server URI and this chat instance's client name can be specified. There should also be, accessible from the options menu, a screen to see all other chat clients, as well as a screen to see all messages from a particular client.

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 four operations that this helper class supports:

- Register with the cloud chat service. This operation will only succeed provided the requested client name has not already been registered by a different client.
- Send a message to the cloud chat service.
- Refresh the list of messages received from the cloud chat service, as well as the list of peers registered with the chat service.

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

Registration should generate a unique identifier (use the Java UUID class) to identify the app installation. Save this with the desired user name, and a registration client identifier (a long integer, set to 0 initially,) in a shared preferences file when the user performs registration. The client identifier is updated when registration is confirmed by the Web service. Use notifications to inform the user if registration succeeds, or fails because that user name is already taken. In the latter case, the user will have to try to register again, with a different user name. Sending of chat messages is not enabled until registration succeeds (at which point the client will have a server-assigned client identifier, that can be used as the primary key for the client record in the clients database, stored on the server and replicated on each app installation).

Chat messages should be stored in a content provider for the app. In addition to the message text, store the timestamp of the message (a Java Date value, stored in the

database as a long integer), a unique sequence number for that message (a long) set by the chat server (see below) and the identity of the sender (another long, provided by the chat server). We will be adding other metadata in future assignments. 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 store the sequence number of the most recent chat message that it has received in 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 client identifier 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.
4. The client deletes the messages it has just uploaded, and inserts the messages received from the chat server. It also replaces the list of chat clients with the list received from the server. Assuming you are using server-assigned client 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.

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 two forms of request messages: `Register` and `Synchronize`. Define two concrete subclasses of an abstract base class, `Request`, for each of these cases. The basic interface for the `Request` class is as follows:

```
public abstract class Request implements Parcelable {
   public long clientID;
   public UUID registrationID;  // sanity check
   // App-specific HTTP request headers.
```

```
   public abstract Map<String,String> getRequestHeaders();
   // Chat service URI with parameters e.g. query string parameters.
   public abstract Uri getRequestUri();
   // JSON body (if not null) for request data not passed in headers.
   public String getRequestEntity() throws IOException;
   // Define your own Response class, including HTTP response code.
   public Response getResponse(HttpURLConnection connection,
                               JsonReader rd /* Null for streaming */);
}
```

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). 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 two methods, one for each form of request:

```
public class RequestProcessor {
   public void perform(Register request) { … }
   public void perform(Synchronize 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:

```
public class RestMethod {
   ... // See lectures.
   public Response perform(Register request) { … }
   public StreamingResponse perform(Synchronize 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. You can supply all of the request parameters as (URL-encoded) query string parameters appended to the original service URI[1].
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

[1] For all requests except registration, the client id (returned from the registration request) should be provided as the last segment in the URI itself.

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 response from the synchronization request has this type:

```
public class StreamingResponse {
   public Response getResponse();
   public InputStream getInputStream();
   public void disconnect();
}
```

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, including performing the streaming upload of data if a POST request, and then returns the open download stream to the request processor. The latter provides a callback operation to the REST method for streaming the output, and after the call, receives 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. You will need to persist the sequence number of the last message that you received from the service. The only other 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.

One way to retry requests is to rely on the user to do it manually, but this is obviously unsatisfactory. 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 retry a request that failed because the server was unavailable. The alarm handler should perform synchronization, provided registration has succeeded, in order to refresh the app state with state on the server.

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. It takes two optional command line arguments: The host name (default `localhost`) and the HTTP port (default 8080). 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. For example:

```
curl -X POST -D headers \
      'http://localhost:8080/chat?username=joe&regid=...'
```

This sends a POST request to the specified (registration) URI, and places the response headers in a file called `headers`. The query parameters include the desired user name and the UUID for the new registration. The contents of header file will be of the form:

```
HTTP/1.1 201 Created
Content-Encoding: UTF-8
Content-Location: http://localhost:8080/chat/1
Content-Type: application/json
Date: Fri, 28 Feb 2014 14:57:25 GMT
Content-Length: 8
```

The output will be a JSON object `{"id":1}`. The identifier is used by the client to identify itself in subsequent requests. This identifier will be part of the URI that the client uses to specify the service. Provide the UUID as well, as a query string parameter, as a sanity check. The following command will synchronize messages with the server:

```
curl -X POST -H "Content-Type: application/json" \
      -d @messages.json -D headers \
      'http://localhost:8080/chat/1?regid=...&seqnum=0'
```

The query string parameters include the registration id (UUID) and 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" : 12345678,
      "text" : "hello"
   },
   {
      "chatroom" : "_default",
      "timestamp" : 12345678,
      "text" : "is there anybody out there?"
   }
]
```

This will produce a JSON output of the form:

```
{"clients":["joe"],"messages":[{"chatroom":"_default","timestamp":12345
678,"seqnum":1,"sender":"joe","text":"hello"},{"chatroom":"_default","t
imestamp":12345678,"seqnum":2,"sender":"joe","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).

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 Android app.
3. Also include in the directory a report of your submission. This report should be in PDF format. Do not provide a Word document.

In addition, record short flash, mpeg, avi 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*. You can upload this video to the folder you are provided with in Google Drive. The video must be uploaded on time for the assignment as a whole to be considered as on time. The time of submission is based on the latest of the time you submitted your code and report to Moodle, and the time you uploaded your videos to Google Drive.

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 Eclipse project, for the app you have built. You should also provide a report in the root folder, called README.pdf or README.rtf, that contains a report on your solution, as well as videos demonstrating the working of your assignment (unless the videos were uploaded to Google Drive instead).