CS 522—Spring 2018
Mobile Systems and Applications
Assignment Seven—Simple Cloud Chat Chap

In this assignment, you will implement a simple cloud-based chat app, 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 5.1, API 22).

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 specified, to be saved in shared preferences.

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. For now, this helper class supports two operations:

- Register with the cloud chat service. This operation will only succeed if the requested client name has not already been registered by a different client.
- Send a message to the cloud chat service.

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

Registration should generate a unique client identifier (using the Java UUID class) to identify the app installation. Save this with the desired user name, and a sender identifier (a long integer, set to -1 initially) in a shared preferences file when the user performs registration. Sending of chat messages is not enabled until registration succeeds (at which point the client will have a non-negative server-assigned sender identifier). More use will be made of these identifiers in the next assignment.

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). In addition a message will contain location information about where the sending device was when the message was sent. When a message is generated, its sequence number is set to a non-zero value and the message added to the local database. When the message is uploaded to the chat server, the response includes the server-generated sequence number which is then used to update the message in the database. For now, the UI will just display a list of the messages sent by the client. In the next assignment, you will synchronize with a chat database stored on the server.

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: `RegisterRequest` and `PostMessageRequest`. 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 senderId;  // database PK, assigned by server
   public UUID clientID;  // installation id
   // App-specific HTTP request headers.
   public abstract Map<String,String> getRequestHeaders();
   // 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(RegisterRequest request) { … }
   public void perform(PostMessageRequest 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, such as `HttpClient`. The public API for this class has the form:

```
public class RestMethod {
   ... // See lectures.
   public Response perform(Register request) { … }
   public Response perform(PostMessage request) { … }
}
```

For now, we are just sending *fixed-length requests*. Such a 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. The server expects these HTTP request headers:

1. "`X-Client-Id`", (a UUID identifying the installation).

2. "`X-Timestamp`" (a long integer timestamp for the request).
3. "`X-Latitude`" and "`X-Longitude`" (two double GPS coordinates).

You can just define fixed constants for the GPS coordinates. The forms of requests are:

1. For a registration request, you can supply the chat name for the client as a query parameter `chat-name`, and the remaining parameters as application-specific HTTP request headers (see above).
2. For a message posting request, you will want to include the posted message in a JSON entity body. The message should be described by the following JSON object:
   a. Label "`chatroom`": A chat room identifier (use "`_default`" for now).
   b. Label "`text`": The text of the message.

   Use the `JsonWriter` class to write the uploaded message. The other metadata associated with the request is sent as request headers, as with registration.

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 –H 'X-Client-Id: ...' ... \
      'http://host-name:8080/chat?chat-name=Joe'
```

This sends a POST request to the specified (registration) URI, and places the response headers in a file called `headers`. The desired user name is passed as a query parameter `chat-name`, and the UUID for the new registration is passed as a request header (using the `-H` option for curl). The contents of the response header file will be of the form:

```
HTTP/1.1 201 Created
Location: http://host-name:8080/chat/1
Content-Length: 0
```

When a client has been registered, the Location response header will record the URI for that client. The last segment will contain the identifier for the client, that you should use in the URI of all subsequent client requests to post and synchronize messages.

The following command will post a message to the server:

```
curl -X POST -H "Content-Type: application/json" \
    -d @message.json –H 'X-Latitude: ...' ... -D headers \
    'http://host-name:8080/chat/id/messages'
```

The file `message.json` should contain a single message as a JSON object. For example:

```
   {
```

```
    "chatroom" : "_default",
    "text" : "hello"
}
```

The contents of the response header file will be of the form:

```
HTTP/1.1 201 Created
Location: http://host-name:8080/chat/1/messages/17
Content-Length: 0
```

The last segment of the URI is the message identifier of the message, that should be used when inserting the message in the content provider.

The server contains some debug commands that allow you to interrogate it. For example, you can test if a client is registered as follows:

```
curl -X GET –H 'X-Client-Id: 123e4567-e89b-12d3-a456-426655440000' \
      'http://host-name:8080/chat'
```

You can query for the messages that have been uploaded as follows:

```
curl -X GET 'http://host-name:8080/chat/messages'
```

Finally you can query for the log at the server as follows:

```
curl -X GET 'http://host-name:8080/chat/log'
```

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. 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 mpeg or Quicktime videos of demonstrations 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*. Make sure that the output on the server is visible in the video (The server app will display messages as they are received).

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 a report in the root folder, called `README.pdf`, that contains a report on your solution, as well as videos demonstrating the working of your assignment. The report should describe how to test your app (running on an AVD) against the server, as well as

any parts of the spec that you were unable to complete. Use the debug commands to verify registration and get a list of all messages at the server, and to display the long at the end of your testing.