CS 522—Spring 2015
Mobile Systems and Applications
Assignment Six—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 5.0.1).

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 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 provided 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 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. Sending of chat messages is not enabled until registration succeeds (at which point the client will have a server-assigned client 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). 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. 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: `Register` and `PostMessage`. 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(PostMessage 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. For a registration request, you can supply all of the request parameters

as (URL-encoded) query string parameters appended to the original service URI[1]. For a message posting request, you will want to include the posted message in a JSON entity body, along with the metadata associated with that message. You should provide the following metadata with a message:

1. A chat room identifier (use "`_default`" for now).
2. A timestamp for the message, as a long integer.
3. The text of the message.

Use the `JsonWriter` class to write the uploaded message.

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-latitude: 40.7439905' \
      -H 'X-longitude: -74.0323626' \
      '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. This also includes two application-specific request headers, `X-latitude` and `X-longitude`, that we will be using in future assignments. 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 post a message to the server:

```
curl -X POST -H "Content-Type: application/json" \
      -H 'X-latitude: 40.7439905' \
      -H 'X-longitude: -74.0323626' \
      -d @message.json -D headers \
      'http://localhost:8080/chat/1?regid=...'
```

[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.

The query string parameters include the registration id (UUID). The file `message.json` should contain a single message as a JSON object. For example:

```
{
   "chatroom" : "_default",
   "timestamp" : 12345678,
   "text" : "hello"
}
```

The output will be a JSON object `{"id":17}`. This is the message identifier of the message, that should be used to update the message identifier in the content provider.

Submitting Your Assignment

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

1. Export your Android Studio project to the file system.
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 project for your app.
4. 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*. Make sure that the output on the server is visible in the video (The server app will display messages as they are received). 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 Canvas, 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 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 (unless the videos were uploaded to Google Drive instead).