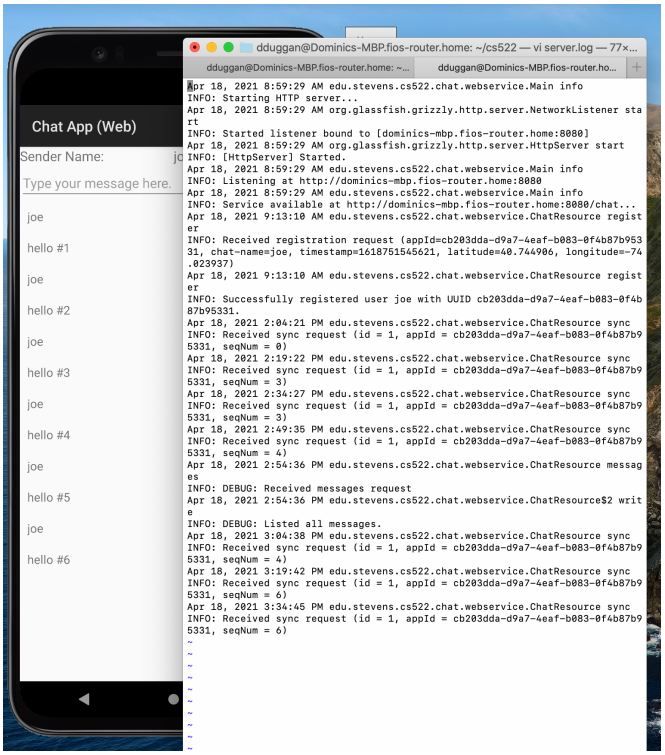
CS 522—Spring 2024
Mobile Systems and Applications
Assignment Ten—Cloud Chat Chap

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 will use the `retrofit2` library to implement your Web service client, following the software architecture described in class. You will use the same code base as in the previous assignment, but you should enable synchronization with the chat server by setting this flag in Settings to true:

```
public static final boolean SYNC = true;
```



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, as well as the name under which you are registered at the server. 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 several 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.
- Start and stop message synchronization: For this assignment, these operations will enable and disable background synchronization of the chat database with the server. They are invoked by the lifecycle methods in the main activity.

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

When a message is generated, it is added to the database 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 database immediately, without communicating with the server.

The database also stores a list of the other clients registered with the chat service. This list, the list of chatrooms 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 database 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 database 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 sender 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 all the chatrooms, 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 and chatrooms with the lists received from the server, inserting new records and updating existing client records with client metadata[1]. Since we have been using sender chat names as foreign key

[1] So all peers, chatrooms and messages downloaded from the server are “upserted” into the local database.

references for senders of messages, you will be able to maintain the correct relationships between clients and messages in your database.

As before, the service helper class should use the WorkManager Lite API 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.

Request Processing

There are three forms of request messages: `RegisterRequest`, `PostMessageRequest` and `SynchronizeRequest`. Two of these messages are spawned by the UI using the service helper, as before. The third is periodically spawned by a background service. 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 ChatServiceRequest implements Parcelable {
   public UUID appID;  // installation id
   public Date timestamp;
   public double latitude;
   public double longitude;
   // Application-defined HTTP request headers.
   public abstract Map<String,String> getRequestHeaders();;
   // Define your own Response class, including HTTP response code.
   public ChatServiceResponse
          getResponse(retrofit2.Response<?> response);
}
```

The time and location information is used to record the last known location of the client at the server.

The business logic for processing these requests is defined in a class called `RequestProcessor`. This is again a POJO 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. This classes uses `retrofit2` to wrap a type-safe API around an `okHttp` client stub, using the Gson library to marshall

and unmarshall between POJO obects (actually entity objects) and JSON. 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) { … }
}
```

The server expects these HTTP request headers:

1. "`X-App-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).

These headers are added by `RequestProcessor.process(`*`request`*`)`, which then uses the visitor pattern to dispatch to the logic for processing the request.

Synchronization with the Server

The background synchronization of messages with the server is done using the WorkManager Lite API, invoked in `ChatHelper.startMessageSync` (which is called in `ChatActivity.onCreate`). Define a periodic request by instantiating this class:

```
public class PeriodicWorkRequest extends WorkRequest {
    public <T extends Worker> PeriodicWorkRequest (Class<T> _class,
                                                   Bundle data,
                                                   int interval);
}
```

The request specifies the class of the worker and the interval between executions of the periodic request (in minutes):

```
PeriodicWorkRequest syncRequest =
    new PeriodicWorkRequest(SynchronizeWorker.class, ...);

WorkManager.getInstance(context)
           .enqueuePeriodicUniqueWork(syncRequest);
```

The `SynchronizeWorker` object instantiates the request processor and calls it with a `SynchronizeRequest` message. The worker object is invoked with a period specified as one of the arguments to the `PeriodicWorkRequest` constructor. The reason that we use

---

[2] We have no use for the case for `PostMessageRequest` in `RestMethod` in this assignment, because the message is logged in the database and asynchronously uploaded by the WorkManager Lite 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.

our own home-brewed WorkManager Lite API is that WorkManager requires this period to be no less than fifteen minutes!

Syncronization is disabled in `ChatHelper.stopMessageSync`, which should be invoked in `ChatActivity.onStop`, where it is important that the original synchronization request object be passed to the work manager operation for cancelling that operation:

```
WorkManager.getInstance(context)
           .cancelPeriodicUniqueWork(syncRequest);
```

As before, the logic for synchronizing with the server is defined in a worker object (whose class is provided in the `PeriodicWorkRequest` object enqueued with Work Manager Lite):

```
public class SynchronizeWorker extends Worker {
    ...
}
```

The worker just instantiates the request processor and invokes the logic for performing synchronization. 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 in the REST method has a second argument of this type:

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

and the response from the synchronization request has this type:

```
public class StreamingResponse {
   public InputStream getInputStream();
   public ChatServiceResponse getResponse();
   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 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 database: 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.

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 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).
- `--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").

To post a message to the server, the app posts to a URI of the form:

`http://`*`host-name`*`:8080/chat/<`*`chat-name`*`>/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 payload for the request should contain a list of all chatrooms and a list of messages to be uploaded, in JSON format. For example:

```
{
   "chatrooms" : [ { "name" : "_default" } ],
   "messages" : [
      {
         "id":1,"seqNum":0,"chatroom":"_default","messageText" : "hello"
         "timestamp":...,"latitude":...,"longitude":...,
         "sender":"joe","senderId":1
      },
      {
         "id":2,"seqNum":0,"chatroom":"_default",,"messageText":"..."
         "timestamp":...,"latitude":...,"longitude":...,
         "sender":"joe","senderId":1
      }
   ]
}
```

The response from the server will include JSON output of the form:

```
{
  "peers" : [
    {"id":1,"name":"joe","timestamp":0,"latitude":0.0,"longitude":0.0}
  ],
  "chatrooms" : [{"name ":"_default"}],
```

```
  "messages" : [
    {"id":1,"seqNum":1,"chatroom":"_default","messageText":"hello",
     "timestamp":...,"latitude":...,"longitude":...,"sender":"joe"},
    {"id":2,"seqNum":2,"chatroom":"_default","messageText":"goodbye",
     "timestamp":...,"latitude":...,"longitude":...,"sender":"joe"}
  ]
}
```

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). The peer database should be updated with the client information downloaded from the server, by "upserting" the downloaded client records (updating existing client records with metadata). Using sender names as foreign keys to peer records in the local database will ensure that downloaded messages remain correctly linked to their sender records.

As before, you can query for the messages that have been uploaded by sending a GET command (e.g., with a web browser) to the URI:

`http://ip-address:8080/chat/messages`

Finally, you can query for the log at the server by sending a GET command to the URI:

`http://ip-address: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. In addition, record mpeg 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 a running server in EC2 that you will be provided with. Make sure that this is defined as the base URI for the server when you are registering. 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. Do this for both devices. Your final submission should include a demonstration of your app running against the chat server that will be provided running in the cloud.