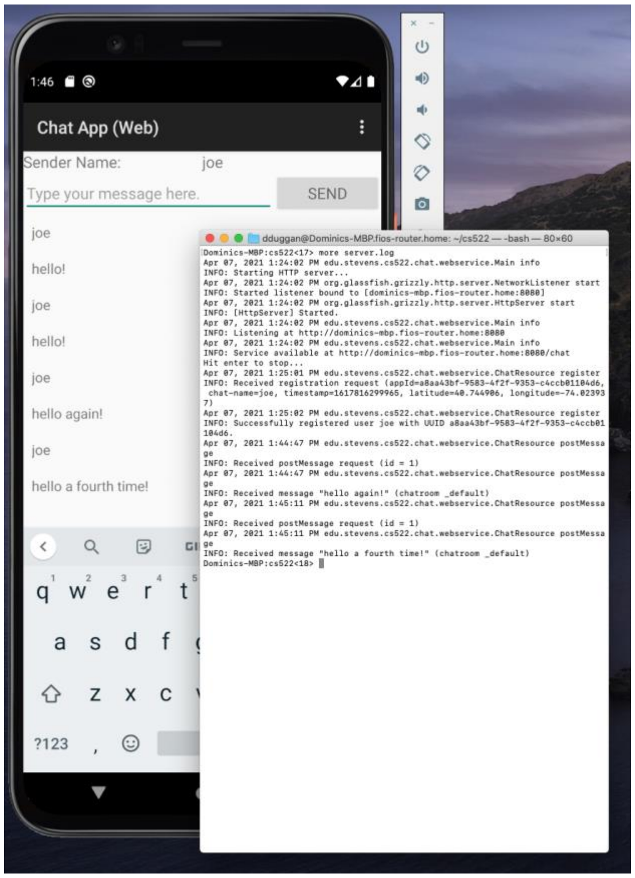
CS 522—Spring 2024
Mobile Systems and Applications
Assignment Nine—Simple Cloud Chat Chap

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 will use the `retrofit2` library to implement your Web service client, following the software architecture described in class.


1:46
Chat App (Web)
Sender Name:
joe
Type your message here.
SEND
ddugan@Dominics-MBP.fios-router.home: ~/cs522 — -bash — 80×60

joe
hello!
joe
hello!
joe
hello again!
joe
hello a fourth time!

```
Dominics-MBP:cs522<17> more server.log
Apr 07, 2021 1:24:02 PM edu.stevens.cs522.chat.webservice.Main info
INFO: Starting HTTP server...
Apr 07, 2021 1:24:02 PM org.glassfish.grizzly.http.server.NetworkListener start
INFO: Started listener bound to [dominics-mbp.fios-router.home:8080]
Apr 07, 2021 1:24:02 PM org.glassfish.grizzly.http.server.HttpServer start
INFO: [HttpServer] Started.
Apr 07, 2021 1:24:02 PM edu.stevens.cs522.chat.webservice.Main info
INFO: Listening at http://dominics-mbp.fios-router.home:8080
Apr 07, 2021 1:24:02 PM edu.stevens.cs522.chat.webservice.Main info
INFO: Service available at http://dominics-mbp.fios-router.home:8080/chat
Hit enter to stop...
Apr 07, 2021 1:25:01 PM edu.stevens.cs522.chat.webservice.ChatResource register
INFO: Received registration request (appId=a8aa43bf-9583-4f2f-9353-c4ccb01104d6,
 chat-name=joe, timestamp=1617816299965, latitude=40.744906, longitude=-74.02393
7)
Apr 07, 2021 1:25:02 PM edu.stevens.cs522.chat.webservice.ChatResource register
INFO: Successfully registered user joe with UUID a8aa43bf-9583-4f2f-9353-c4ccb01
104d6.
Apr 07, 2021 1:44:47 PM edu.stevens.cs522.chat.webservice.ChatResource postMessa
ge
INFO: Received postMessage request (id = 1)
Apr 07, 2021 1:44:47 PM edu.stevens.cs522.chat.webservice.ChatResource postMessa
ge
INFO: Received message "hello again!" (chatroom _default)
Apr 07, 2021 1:45:11 PM edu.stevens.cs522.chat.webservice.ChatResource postMessa
ge
INFO: Received postMessage request (id = 1)
Apr 07, 2021 1:45:11 PM edu.stevens.cs522.chat.webservice.ChatResource postMessa
ge
INFO: Received message "hello a fourth time!" (chatroom _default)
Dominics-MBP:cs522<18>
```

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, is 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 in a shared preferences file when the user performs registration. Sending of chat messages is not enabled until registration succeeds.

Chat messages are stored in a database for the app, as before. 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 and the message is updated with this 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.

Web Services

There are two forms of request messages that are sent to the chat server: `RegisterRequest` and `PostMessageRequest`. Two concrete subclasses of an abstract base class, `ChatServiceRequest`, are defined for each of these cases. The basic interface for the `ChatServiceRequest` class is as follows:

```
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);
}

public abstract class ChatServiceResponse implements Parcelable {
   public String responseMessage;  // Human readable
   public int httpResponseCode;
   public String httpResponseMessage;
   public abstract Boolean isValid();
}
```

The business logic for processing these requests in the app should be defined in a class called `RequestProcessor`. This 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. 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:

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

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).

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). The peer object should not be inserted into the database until the user has successfully registered.
2. For a message posting request, you will want to include the posted message in a JSON entity body. The message will be a JSON object corresponding to the `Message` entity class defined in earlier assignments, with a new field, `seqNum`, for the sequence number that will be assigned to the message in the next assignment. In the next assignment, we will see a more realistic way for exchanging messages with the server.

Running in the Background

Both the registration and message posting must be done asynchronously, on background threads. The registration is done while the user is waiting for confirmation, so we will do that in a *foreground service*[1], that uses a notification to inform the user of the progress of the registration. A foreground service attaches itself to the user interface with a notification, preventing the low memory killer from destroying the service and allowing

[1] https://developer.android.com/guide/components/foreground-services.

the service to update the user on the status of registration. The service attaches itself to the notification bar with the notification that should be displayed:
```
startForeground(notificationId, notification, type);
```

The service uses an executor service to perform the registration Web service on a background thread, while a handler is used to execute the reporting of the result in the UI back on the main thread:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Handler mainLoop = new Handler(Looper.getMainLooper());
...
executor.execute(() -> {

    response = processor.process(registerRequest);

    mainLoop.post(() -> {
        // Update the notification with the result
        stopForeground(Service.STOP_FOREGROUND_DETACH);
        stopSelf(startId);
    });

}
```

You should a button to the notification that allows the user to cancel registration if they wish. If registration succeeds or fails, the notification should be updated to reflect the result (Use `NotificationManager::notify` to do this). The notification will stay in the notification bar until the user explicitly removes it with a side swipe.

For posting messages to the server, in preparation for the next assignment where we synchronize messages with the chat server, the recommended practice is to use the WorkManager API. However, there is one aspect of this API that is pragmatically difficult: It requires at least fifteen minutes between periodic requests! So we will work with our own home-brewed version, that we will call WorkManager Lite, that removes this restriction. It does not have the power or resilience of the original (e.g., it does not ensure that requests are not lost if the low memory killer stops the process), but it does the work of scheduling jobs on background threads and acquiring partial wake locks. The WorkManager Lite API is similar enough to the original that it should be relatively simple for you to get your app working with WorkManager, if you wish, but it is not required for this assignment.

To use the WorkManager Lite API, you define the logic for a worker thread by subclassing this definition (the subclass must define a binary constructor with the same signature as that in this base class):

```
package edu.stevens.cs522.chat.base.work;

public abstract class Worker {
    protected Worker(Context context, Bundle data);
    public abstract boolean doWork();
}
```

The code base you are provided with includes a worker for posting a single message:

```
public class PostMessageWorker extends Worker {
    private final Message message;
    ...
 }
```

Define a one-time work request, e.g., to upload a message to the server, by instantiating the following class, where the bundle argument contains initial data for the execution of the work request (e.g., the message to be uploaded):

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

The request is enqueued to be executed on a background thread by the work manager itself:

```
public class WorkManager {
    public static WorkManager getInstance(Context context);
    public void enqueueUniqueWork(OneTimeWorkRequest request;
}
```

You should complete the logic in the `ChatHelper` object, to use WorkManager Lite to schedule the uploading of a chat message to the server in the background.

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[2]. It takes several optional command line arguments:

- `--host` *`host-name`*: The name of the host the server is running on (default[3] is `localhost`).
- `--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").

On the Android emulator, IP address `10.0.2.2` is the synonym for the loopback interface for the host machine on which the emulator is running. This should allow your app on an

[2] The server will also be running on host in Amazon EC2. Details will be provided through Canvas.

[3] Depending on your OS, you may have to explicitly specify the IP address of the machine on which you are running the server.

AVD to connect to the server running on the host machine, but it doesn't. On the other hand, targeting the actual IP address on your network of the host machine does appear to work. Another wrinkle is that by default Android does not allow cleartext Web traffic. Since we don't want to deal with keystores and truststores for this assignment, you can enable cleartext traffic[4] with this network configuration:

```
<network-security-config>
    <base-config cleartextTrafficPermitted="true">
        <trust-anchors>
            <certificates src="system" />
        </trust-anchors>
    </base-config>
</network-security-config>
```

This is specified as the configuration in the application manifest:

```
<application
   android:networkSecurityConfig="@xml/network_security_config" .../>
```

When the user registers with the server, they should provide a URI of the form:

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

The app will add a context root of `chat` to this URI, and then register with a POST request. It will provide the chat name as a query parameter in the HTTP request, and it will provide its unique application identifier (a UUID generated by the app during registration) as an application-specific HTTP request header `X-App-Id`, with other application-specific headers carrying metadata about the client (location, timestamp):

```
http://ip-address:8080/chat?chat-name=Joe
```

The response will be of the form:

```
HTTP/1.1 201 Created
Location: http://ip-address: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 server-side database key for the client record, that will be saved in shared preferences in the app. However later communications with the server will use the chat name and application UUID, passed in the `X-App-Id` request header, to identify the client on the server.

For development purposes, the chat server allows a chat name to be re-registered. However, we cannot have two different clients with the same chat name synchronizing at

[4] You should obviously never do this in a production system.

the same time with the server. Therefore, the re-registration of a chat name invalidates the previous registration, and the previous client will no longer be able to interact with the server. Different registrations with the same chat name are distinguished by the application id, that is provided with each Web service request. If you need to re-register with the server, you will have to uninstall and re-install the app, so re-registering with the server is effectively starting as a new client.

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

```
http://host-name:8080/chat/chat-name/messages
```

This time, the POST request includes a payload, a JSON object of the form:

```
   {  "id" : "...",
      "seqNum" : "0"
      "appID" : "123e4567-e89b-12d3-a456-426655440000",
      "chatroom" : "_default",
      ...
      "sender": "joe"
   }
```

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

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

The last segment of the URI is the global sequence number of the message, and the copy of the message in the local database should be updated with this once the message has been uploaded. You can think of it as the primary key for the message on the server database. This might be used for example to identify the message on the server if we wanted to allow users to delete individual messages.

The server contains some debug commands that allow you to interrogate it. For example, you can test if a client is registered by using a tool such as `curl` to send a GET query:

```
curl -X GET –H 'X-App-Id: 123e4567-e89b-12d3-a456-426655440000'
      'http://ip-address:8080/chat'
```

This curl command adds the `X-App-Id` header to the request. 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.

In addition, record mpeg videos of demonstrations of your assignment working. You should show the app launching from Android Studio, with completed parts in RequestProcessor visible, and running 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 should demonstrate your app working against a running server, either on your local machine or on a server in EC2 that you are provided with. You supply the base URI for the server as part of registration[5] (to facilitate testing on your local machine first of all). 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 a completed rubric, that contains a report on your solution, as well as videos demonstrating the working of your assignment. Use the debug commands for the server to verify registration and get a list of all messages at the server, and to display the log at the end of your testing.

[5] The base URI should **not** include the context root, `chat`.