

# HƯỚNG DẪN SỬ DỤNG (DÀNH CHO DEVELOPER)

Môn: Lập trình trên thiết bị di động

Đề tài: Tạo ứng dụng nhắn tin trực tuyến

Nhóm: 7

## Mục lục

1. Hướng dẫn hướng dẫn sử dụng backend .....	2
1. 1. Hướng dẫn chạy dự án trên local .....	2
a) Hướng dẫn cấu hình bên phía Server:.....	2
b) Hướng dẫn cấu hình bên phía client .....	3
c) Kết luận .....	5
1. 2. Hướng dẫn deploy dự án lên server .....	6
a) Khởi tạo kubernetes Cluster.....	6
b) Deploy mysql .....	10
c) Deploy Springboot Server.....	11
d) Hướng dẫn dưới đây sử dụng docker trên Linux:.....	12
e) Deploy socket server .....	13
f) Thiết lập ip cho client giao tiếp với server.....	14
2. Hướng dẫn dev .....	17
2. 1. Tổng quan về các API bên springboot server.....	17
a) Account Controller.....	17
b) User Controller.....	19
c) ChatRoom Controller .....	21
d) Message Controller .....	24
2. 2. Tổng quan các hàm của socket server.....	25
a) Lớp ClientHandleService.....	25
b) Lớp RequestService .....	26
c) Lớp ServerService.....	28
2. 3. Tổng quan các hàm của client .....	33
a) Gói AsyncTask .....	33
b) Các lớp xử lý response .....	37
c) Các Activity và Fragment .....	42
d) Service.....	47

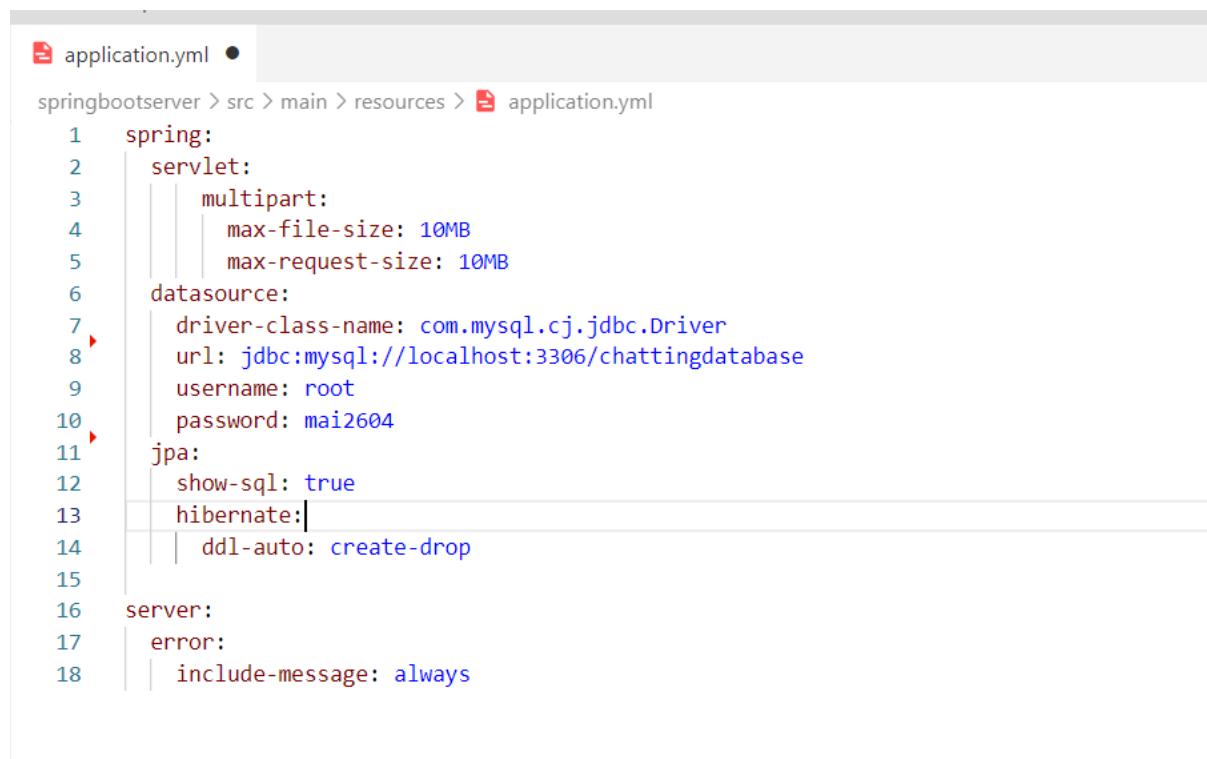
## 1. Hướng dẫn hướng dẫn sử dụng backend

### 1. 1. Hướng dẫn chạy dự án trên local

Lưu ý đầu tiên kết nối chỉ thành công khi mà các thiết bị có cùng 1 kết nối mạng wifi, nếu khác mạng sẽ không thể chạy được.

a) Hướng dẫn cấu hình bên phía Server:

**Bước 1:** Vào thư mục project springbootserver và truy cập theo đường dẫn `springbootserver\src\main\resources` truy cập file `application.yml`. Hệ thống sẽ sử dụng hệ quản trị cơ sở dữ liệu MySQL nên lưu ý phải có hệ quản trị này trên máy, tiếp theo cần cấu hình lại các trường kết nối gồm username và password là tài khoản của MySQL.



```
application.yml
```

```
springbootserver > src > main > resources > application.yml
```

```
1  spring:
2    servlet:
3      multipart:
4        max-file-size: 10MB
5        max-request-size: 10MB
6    datasource:
7      driver-class-name: com.mysql.cj.jdbc.Driver
8      url: jdbc:mysql://localhost:3306/chattingdatabase
9      username: root
10     password: mai2604
11   jpa:
12     show-sql: true
13     hibernate:
14       ddl-auto: create-drop
15
16   server:
17     error:
18       include-message: always
```

**Bước 2:** Trong cùng thư mục project đó, tìm và truy cập vào file `SpringbootserverApplication.java` để chạy spring boot server.

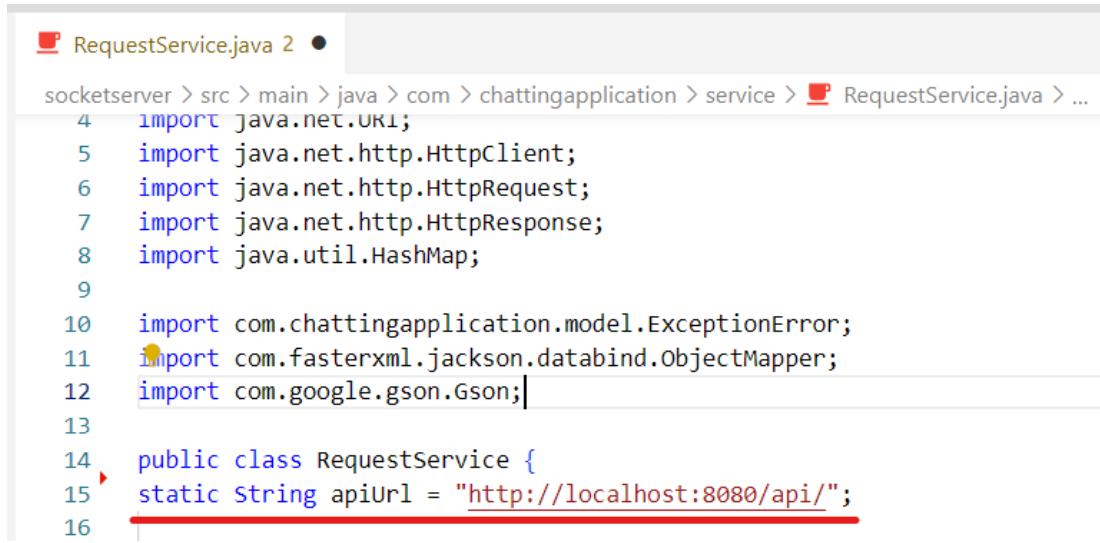


```
application.yml SpringbootserverApplication.java
```

```
springbootserver > src > main > java > com > chattingapplication > springbootserver > SpringbootserverApplication.java > ...
```

```
1 package com.chattingapplication.springbootserver;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6@SpringBootApplication()
7 public class SpringbootserverApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(SpringbootserverApplication.class, args);
11     }
12 }
```

**Bước 3:** Đến thư mục project socketserver tìm package service và truy cập vào file RequestService.java đảm bảo đã tạo chuỗi kết nối đến với spring boot server như sau:



```
RequestService.java 2

socketserver > src > main > java > com > chattingapplication > service > RequestService.java > ...
4 import java.net.URI;
5 import java.net.http.HttpClient;
6 import java.net.http.HttpRequest;
7 import java.net.http.HttpResponse;
8 import java.util.HashMap;
9
10 import com.chattingapplication.model.ExceptionError;
11 import com.fasterxml.jackson.databind.ObjectMapper;
12 import com.google.gson.Gson;
13
14 public class RequestService {
15     static String apiUrl = "http://localhost:8080/api/";
16 }
```

**Bước 4:** Vào file Main.java để khởi động socket server.



```
Main.java x

socketserver > src > main > java > com > chattingapplication > Main.java > ...
1 package com.chattingapplication;
2
3 import java.io.IOException;
4
5 import com.chattingapplication.controller.ServerController;
6
7 public class Main {
8     Run|Debug
9     public static void main(String[] args) throws IOException, InterruptedException {
10         ServerController.handleConnect();
11     }
12 }
```

b) Hướng dẫn cấu hình bên phía client

**Bước 1:** Khởi động IDE Android Studio và mở thư mục project chattingclient.

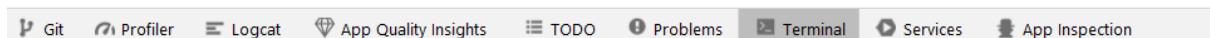
**Bước 2:** Trước tiên cần gõ lệnh setupip trong terminal và sử dụng cmd (lưu ý sử dụng powershell sẽ không chạy được), lệnh này được viết bằng batch script có nhiệm vụ sẽ tự động lấy địa chỉ Ipv4 trên máy tính được chọn làm server hiện tại và lưu vào file my\_ipv4.json nằm trong folder assets của project.

Kết quả khi chạy batch script setupip.bat in ra địa chỉ Ipv4:

```
Terminal: Local × Command Prompt × + ▾
Microsoft Windows [Version 10.0.22621.1702]
(c) Microsoft Corporation. All rights reserved.

C:\DieuHuynh\Projects\ChatApplication\chattingclient>setupip
192.168.101.32

C:\DieuHuynh\Projects\ChatApplication\chattingclient>
```



Địa chỉ IPv4 sẽ được lưu lại vào file my\_ipv4.json như sau:



**Bước 3:** Tiếp đến để project có thể chạy được cần phải đọc chuỗi json có địa chỉ IPv4 này và gán vào biến String để tạo 1 chuỗi kết nối đơn giản với spring boot server và socket server, chúng ta sẽ tiến hành tạo chuỗi kết nối server bằng cách truy cập vào file LoadActivity.java, trong hàm onCreate(...) đảm bảo đã có 2 dòng lệnh tạo chuỗi kết nối như sau:

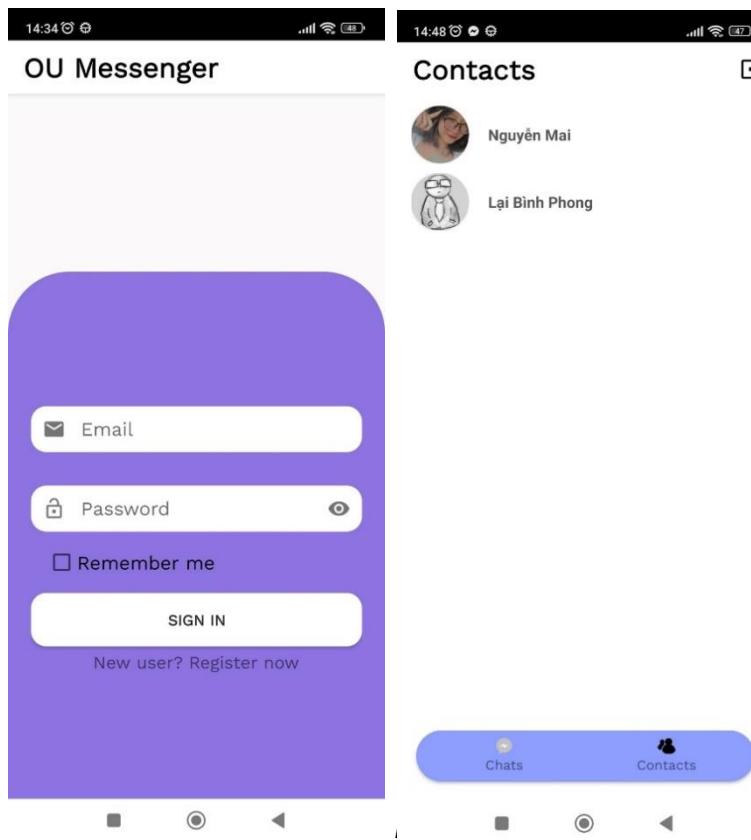
```

my_ipv4.json × LoadActivity.java ×
no usages
37     public static List<Long> idList = new ArrayList<>();
38
39     ↗ PlalsMe +1
40     @Override
41     protected void onCreate(Bundle savedInstanceState) {
42         super.onCreate(savedInstanceState);
43         setContentView(R.layout.activity_load);
44
45         preferencesCheck = getSharedPreferences( name: "check", MODE_PRIVATE);
46         preferencesAccount = getSharedPreferences( name: "account", MODE_PRIVATE);
47         currentContext = this;
48
49         IP = Utils.getIpV4( mainActivity: this, fileName: "my_ipv4.json");
50         apiUrl = String.format("http://%s:8080/api/", IP);
51
52         init();
53         progressBar.setVisibility(View.VISIBLE);
54         ConnectTask connectTask = new ConnectTask( activity: this);
55         connectTask.execute();
}

```

### c) Kết luận

Sau khi đã đảm bảo thực hiện các bước như trên chúng ta đã có thể tiến hành build và run project android studio. Kết quả của việc chạy project thành công sẽ thấy được giao diện đăng nhập và không bị gián đoạn kết nối với server.



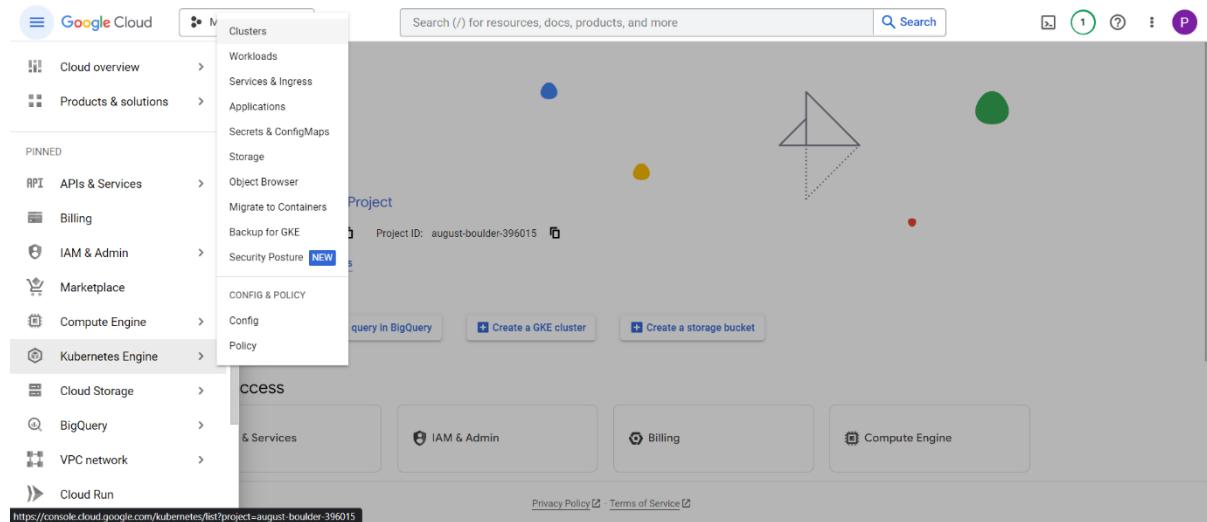
## 1. 2. Hướng dẫn deploy dự án lên server

### a) Khởi tạo kubernetes Cluster

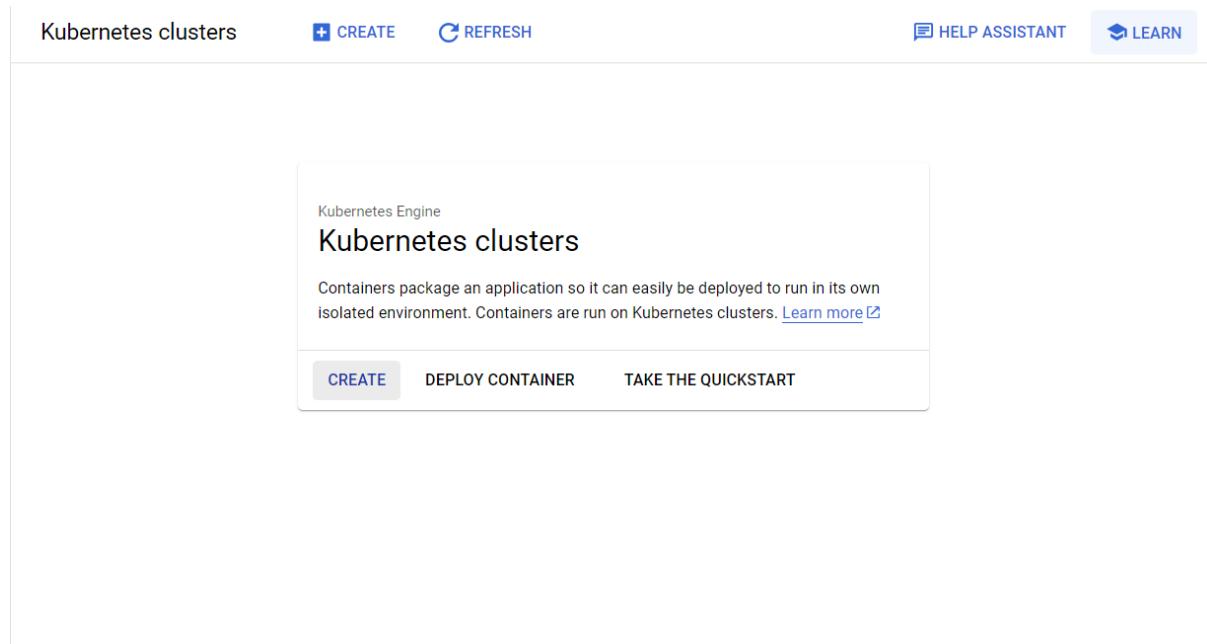
Dự án sử dụng kubernetes để deploy vì được code theo mô hình microservice gồm 2 server là springboot server và socket server.

Kubernetes hiện nay có mặt trên nhiều nền tảng cloud, trong dự án này sử dụng dịch vụ kubernetes engine của Google Cloud Platforms để deploy.

**Bước 1:** Trong trang chủ Google Cloud Platforms, chọn dịch vụ Kubernetes Engine, sau đó bấm vào Clusters.



**Bước 2:** Nhấn vào Create để tạo 1 cluster mới



**Bước 3:** Nhập tên Cluster ghi nhớ vùng (Region) đã chọn, sau đó nhấn Create

## Cluster basics

Create an Autopilot cluster by specifying a name and region. After the cluster is created, you can deploy your workload through Kubernetes and we'll take care of the rest, including:

- ✓ **Nodes:** Automated node provisioning, scaling, and maintenance
- ✓ **Networking:** VPC-native traffic routing for public or private clusters
- ✓ **Security:** Shielded GKE Nodes and Workload Identity
- ✓ **Telemetry:** Cloud Operations logging and monitoring

Name

Cluster names must start with a lowercase letter followed by up to 39 lowercase letters, numbers, or hyphens. They can't end with a hyphen. You cannot change the cluster's name once it's created.

Region

The regional location in which your cluster's control plane and nodes are located. You cannot change the cluster's region once it's created.

[NEXT: NETWORKING](#) [RESET SETTINGS](#)

---

[CREATE](#) [CANCEL](#) Equivalent [REST](#) or [COMMAND LINE](#)

**Bước 4:** Trở lại trang chủ, chọn dịch vụ Disks của Computer Engine để tạo nơi lưu trữ database.

The screenshot shows the Google Cloud Platform dashboard. The left sidebar has a pinned menu with Compute Engine selected. The main content area is titled "DISKS" and shows a list of disks under the heading "STORAGE". One disk, "chatting-disk", is listed with details: "Size: 100 GB", "Status: healthy", and "Created: 2023-01-15". Below the disk list are sections for "INSTANCE GROUPS" and "VM MANAGER". A modal window is open over the content, showing a list of virtual machines with "chatting-vm" selected. The modal also displays the "Location: asia-southeast2" and "Mode: Autopilot".

Bấm vào Create Disk.

The screenshot shows the Google Cloud Storage console's Disks page. At the top, there are buttons for CREATE DISK, REFRESH, and DELETE. To the right are links for OPERATIONS, HELP ASSISTANT, and LEARN. Below the header is a filter bar with a search input field labeled "Enter property name or value". A table below the filter shows one row: "local-storage". The columns in the table are: Status, Name (with an upward arrow), Type, Size, Architecture, Zone(s), In use by, Snapshot schedule, and Actions. A message at the bottom says "No rows to display".

Sau đó nhập tên disk và chọn vùng reagion TRÙNG VỚI vùng của kubernetes cluster vừa tạo sau đó nhấn Create.

Name \*

local-storage



Name is permanent

Description

### Location

Single zone

Regional

Create a failover replica in the same region for high availability. Storage and data replication is provided between both zones. [Learn more](#)

Region \*

asia-southeast2 (Jakarta)



Zone \*

asia-southeast2-a



### Source

Create a blank disk, apply a bootable disk image, or restore a snapshot of another disk in this project.

Disk source type \*

Blank disk

**CREATE**

CANCEL

EQUIVALENT COMMAND LINE



**Bước 5:** Sau khi kubernetes cluster đã được tạo xong, bấm vào connect.

Sau đó bấm vào connect.

Clusters

chatting-cluster

DETAILS STORAGE OBSERVABILITY LOGS APP ERRORS

Starting from September 2023, all GKE Autopilot clusters (minimum version to be determined in August 2023) will migrate to Cloud DNS as the default DNS provider (in replacement of kube-dns), at no extra charge.

However, if you plan to deploy or migrate GKE Autopilot clusters to an [Assured Workloads](#) folder with an [Impact Level 4 \(IL4\)](#) compliance regime, please reach out to your account manager to get more details about your migration path.

[Learn more about Cloud DNS for GKE](#)

Copy đoạn mã.

Connect to the cluster

You can connect to your cluster via command-line or using a dashboard.

**Command-line access**

Configure [kubectl](#) command line access by running the following command:

```
$ gcloud container clusters get-credentials chatting-cluster --region asia-southeast2 --project august-boulder-396015
```

[RUN IN CLOUD SHELL](#) [Copy to clipboard](#)

**Cloud Console dashboard**

You can view the workloads running in your cluster in the Cloud Console [Workloads dashboard](#).

[OPEN WORKLOADS DASHBOARD](#)

OK

**Bước 6:** Vào Google Cloud SDK Shell gõ gcloud auth login sau đó đăng nhập bằng tài khoản google đã dùng trên cloud sau đó dán đoạn mã trên và enter.

```
You are now logged in as [phonghuyen08092002@gmail.com].  
Your current project is [dark-bit-389500]. You can change this setting by running:  
$ gcloud config set project PROJECT_ID  
C:\Users\PHONG\AppData\Local\Google\Cloud SDK>gcloud auth login  
C:\Users\PHONG\AppData\Local\Google\Cloud SDK>gcloud container clusters get-credentials chatting-cluster --region asia-southeast2 --project august-boulder-396015  
Fetching cluster endpoint and auth data.  
kubeconfig entry generated for chatting-cluster.  
C:\Users\PHONG\AppData\Local\Google\Cloud SDK>
```

Để kiểm chứng kubernetes cluster đã có chưa gõ kubectl get all -o wide

```
C:\Users\PHONG\AppData\Local\Google\Cloud SDK>kubectl get all -o wide
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE     SELECTOR
service/kubernetes   ClusterIP  34.118.224.1 <none>       443/TCP   17m    <none>
```

b) Deploy mysql

**Bước 1:** Trong Google Cloud SDK Shell cd vào thư mục  
..\\ChatApplication\\k8s-configurations\\MysqlConfigurations

Trong thư mục chứa 5 file yaml trong đó:

- mysql-config.yaml: là file ConfigMap để định tuyến database nào được sử dụng với thuộc tính dbName trong dự án này là chattingdatabase.
- mysql-secret.yaml: chứa username và password của mysql được băm thành mã base64. Trong dự án này dùng username và password là phonglai.
- mysql-storage.yaml: dùng để config persistent volume và persistent volume claim, persistent volume là nơi chứa data còn persistent volume claim là request của pod được tạo từ deployment để sử dụng data đó, trong dự án này sử dụng google cloud platform nên phải định nghĩa disk cho persistent volume với pdName phải trùng với tên Disk vừa tạo ở Computer Engine.

```
gcePersistentDisk:  
  pdName: local-storage  
  fsType: ext4
```

- mysql-deployment.yaml: là file Deploy images của mysql được tải về từ Docker Hub sử dụng username và password của mysql từ mysql-secret.yaml và mount, claim database từ cấu hình file mysql-storage.yaml. Deploy này sẽ tạo ra 1 pod vì không định nghĩa replicas nên mặc định là 1.
- mysql-service.yaml: expose pod đã được mysql-deployment.yaml ra cổng 3306 và 1 địa chỉ nội bộ chỉ các pod khác trong kubernetes cluster này mới giao tiếp được.

**Bước 2:** Gõ các lệnh sau:

```
kubectl apply -f mysql-storage.yaml
```

```
kubectl apply -f mysql-config.yaml
```

```
kubectl apply -f mysql-secret.yaml
```

```
kubectl apply -f mysql-deployment.yaml
```

```
kubectl apply -f mysql-service.yaml
```

```
E:\>cd E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\MySQLConfigurations
E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\MySQLConfigurations>kubectl apply -f mysql-storage.yaml
persistentvolume/mysql-pv created
persistentvolumeclaim/mysql-pv-claim created

E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\MySQLConfigurations>kubectl apply -f mysql-config.yaml
configmap/mysql-config created

E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\MySQLConfigurations>kubectl apply -f mysql-secret.yaml
secret/mysql-secret created

E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\MySQLConfigurations>kubectl apply -f mysql-deployment.yaml
warning: autopilot-default-resources-mutator:Autopilot updated Deployment default/mysql: defaulted unspecified resources for containers [mysql] (see http://g.co/gke/autopilot-defaults)
deployment.apps/mysql created

E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\MySQLConfigurations>kubectl apply -f mysql-service.yaml
service/mysql-service created
```

**Bước 3:** Gõ kubectl get po nếu thấy STATUS là running thì đã deploy mysql thành công nếu chưa thì chờ 1 lúc rồi gõ lại.

```
E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\MySQLConfigurations>kubectl get po
NAME          READY   STATUS    RESTARTS   AGE
mysql-7bb47cddb9-s9jnz   1/1     Running   0          2m44s
```

### c) Deploy Springboot Server

**Bước 1:** Trong Google Cloud SDK Shell gõ kubectl get svc

```
E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\MySQLConfigurations>kubectl get svc
NAME        TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
kubernetes   ClusterIP   34.118.224.1   <none>           443/TCP     39m
mysql-service   ClusterIP   34.118.231.60   <none>           3306/TCP    5m28s
```

**Bước 2:** Copy Cluster-ip của mysql-service và dán vào file application.yml trong đường dẫn:

..\\ChatApplication\\springbootserver\\src\\main\\resources\\application.yml

**Bước 3:** Comments dòng code cấu hình database cho local như hình

```
driver-class-name: com.mysql.cj.jdbc.Driver
url: jdbc:mysql://34.118.231.60:3306/${DB_NAME}
username: ${DB_USERNAME}
password: ${DB_PASSWORD}
# url: jdbc:mysql://localhost:3306/chattingdatabase
# username: root
# # password: mai2604
# password: IamPhong89
```

**Bước 4:** Vào terminal cd vào Springboot Server

PROBLEMS 27 TEST RESULTS DEBUG CONSOLE TERMINAL OUTPUT

PS E:\HocTap\Nam\_3\Hoc\_ki\_3\LapTrinhMobile\ChatApplication> cd ..\springbootserver\

Gõ mvn clean install -DskipTests để xóa file jar trong target cũ và tạo file jar mới đồng thời bỏ qua test vì địa chỉ database này local trên máy không thể truy cập, màn hình kết quả sau khi thành công.

```

[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 11.896 s
[INFO] Finished at: 2023-08-21T22:54:24+07:00
[INFO] -----
PS E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\springbootserver>

```

Tiếp đến ta container hóa springboot server bằng Docker trên windows tải Docker Desktop.

d) Hướng dẫn dưới đây sử dụng docker trên Linux:

**Bước 1:** Vào terminal cd vào springbootserver gõ: docker build -t springbootserver:latest.

Màn hình kết quả sau khi build thành công.

```

phong@phong-VirtualBox:~/ChatApplication/springbootserver$ sudo docker build -t springbootserver:latest .
Sending build context to Docker daemon 48.75MB
Step 1/5 : FROM openjdk:17-jdk-alpine
--> 264c9bdce361
Step 2/5 : WORKDIR /springbootserver
--> Using cache
--> eb4cbe17af7a
Step 3/5 : EXPOSE 8080
--> Using cache
--> 8a877eacea4a
Step 4/5 : COPY ./target/*.jar springbootserver.jar
--> b9132a453740
Step 5/5 : CMD ["java", "-jar", "./springbootserver.jar"]
--> Running in cb192f5d8556
Removing intermediate container cb192f5d8556
--> 57903de6f077
Successfully built 57903de6f077
Successfully tagged springbootserver:latest

```

**Bước 2:** Tag image vào tên tài khoản docker của bạn, trong dự án này là phonglai0809, để tag image gõ: docker tag springbootserver:latest {tên tài khoản}/springbootserver:latest

**Bước 3:** Gõ: docker push phonglai0809/springbootserver:latest màn hình kết quả sau khi push thành công.

```

phong@phong-VirtualBox:~/ChatApplication/springbootserver$ sudo docker tag springbootserver:latest phonglai0809/springbootserver:latest
phong@phong-VirtualBox:~/ChatApplication/springbootserver$ sudo docker push phonglai0809/springbootserver:latest
The push refers to repository [docker.io/phonglai0809/springbootserver]
244b279e8b0d: Pushed
36dadfb7e3a0: Pushed
34f7184834b2: Layer already exists
5836ece05bfd: Layer already exists
72e830a4dff5: Layer already exists
latest: digest: sha256:b55f90119ee37452e833ddabd9607e7143e51ab0b77991cbf07b749353483b8b size: 1370

```

**Bước 4:** Vào Google Cloud SDK Shell cd vào ..\ChatApplication\k8s-configurations\ApplicationConfigurations sau đó gõ

kubectl apply -f springbootserver-deployment.yaml

kubectl apply -f springbootserver-service.yaml

Để kiểm tra deploy đã thành công chưa, gõ kubectl get po nếu status là running là đã deploy thành công.

```
E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\ApplicationConfigurations>kubectl get po
NAME                      READY   STATUS    RESTARTS   AGE
mysql-7bb47cddb9-s9jnz   1/1     Running   0          11h
springbootserver-745788576b-sn9sn  1/1     Running   0          100s
```

### Bước 5: Copy Name của pod mysql gõ kubectl exec -it {name vừa copy} – bash

Sau đó vào mysql bằng username và mật khẩu chưa được băm từ mysql secret kiểm tra database nếu đã gen hết các tables là đã thành công

```
E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\ApplicationConfigurations>kubectl exec -it mysql-7bb47cddb9-s9jnz -- bash
bash-4.4# mysql -uphonglai -pphonglai
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 18
Server version: 8.1.0 MySQL Community Server - GPL

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases
-> ;
+-----+
| Database      |
+-----+
| chattingdatabase |
| information_schema |
| performance_schema |
+-----+
3 rows in set (0.01 sec)

mysql> use chattingdatabase;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+-----+
| Tables_in_chattingdatabase |
+-----+
| account
| chat_room
| message
| user
| user_chat_room |
+-----+
5 rows in set (0.00 sec)
```

### e) Deploy socket server

#### Bước 1: Trong Google Cloud Shell gõ kubectl get svc

```
E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\ApplicationConfigurations>kubectl get svc
NAME           TYPE      CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
kubernetes     ClusterIP  34.118.224.1  <none>        443/TCP       12h
mysql-service  ClusterIP  34.118.231.60 <none>        3306/TCP       11h
springbootserver-service LoadBalancer 34.118.232.203 34.128.104.216  8080:30764/TCP  4m39s
```

Vì springbootserver-service.yaml được cấu hình expose ra bằng loại LoadBalancer nên sẽ có thêm địa chỉ external-ip nhưng socket server là service internal nên ta copy cluster-ip của springbootserver-service.

**Bước 2:** Vào file RequestService.java trong socket server thuộc đường dẫn: ..\ChatApplication\socketserver\src\main\java\com\chattingapplication\service\RequestService.java và comment dòng code: static String apiUrl = "http://localhost:8080/api/"; sau đó dán cluster IP vào dòng trên:

```
public class RequestService {
    static String apiUrl = "http://34.118.232.203:8080/api/";
    // static String apiUrl = "http://localhost:8080/api/";
```

**Bước 3:** Thực hiện clean install cho socket server bằng mvn clean install tương tự như springboot server nhưng không cần skip Test vì socket server không có cấu hình test.

Sau khi clean install xong, lặp lại các bước containerize socket server như spring boot server bằng các lệnh:

```
docker build -t socketserver:latest .
```

```
docker tag socketserver:latest phonglai0809/socketserver:latest
```

```
docker push phonglai0809/socketserver:latest
```

Hình ảnh khi thành công sử dụng Docker Desktop:

```
PS E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\socketserver> docker build -t socketserver:latest .
[+] Building 3.4s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 216B
=> [internal] load .dockerrcignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/openjdk:17-jdk-alpine
=> [auth] library/openjdk:pull token for registry-1.docker.io
=> [1/3] FROM docker.io/library/openjdk:17-jdk-alpine@sha256:4b6abae565492dbe9e7a894137c966a7485154238902f2f25e9dbd9784383d81
=> [internal] load build context
=> => transferring context: 4.61MB
=> CACHED [2/3] WORKDIR /socketserver
=> [3/3] COPY ./target/*.jar socketserver.jar
=> exporting to image
=> => exporting layers
=> => writing image sha256:a3e39bd5ef9aaa09299df8d1116c89d98961653a51a3fbe1d546a0135dec833a
=> => naming to docker.io/library/socketserver:latest
PS E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\socketserver> docker tag socketserver:latest phonglai0809/socketserver:latest
PS E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\socketserver> docker push phonglai0809/socketserver:latest
The push refers to repository [docker.io/phonglai0809/socketserver]
1bb8a1ca014f: Pushed
d6c5457e6a16: Layer already exists
34f7184834b2: Layer already exists
5836ece05bfd: Layer already exists
72e830aa4dff5: Layer already exists
latest: digest: sha256:df12a7a12959e31e9efea53bcd53de24e9169eced05d13630ea5e1e34f1d234c size: 1369
```

**Bước 5:** Vào Google Cloud SDK Shell cd vào ..\ChatApplication\k8s-configurations\ApplicationConfigurations sau đó gõ

```
kubectl apply -f socketserver-deployment.yaml
```

```
kubectl apply -f socketserver-service.yaml
```

**Bước 6:** Chờ đến khi deploy hoàn tất bằng cách gõ kubectl get po kiểm tra tới khi STATUS là Running

```
E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\ApplicationConfigurations>kubectl get po
NAME          READY   STATUS    RESTARTS   AGE
mysql-7bb47cdbb9-s9jnz   1/1     Running   0          12h
socketserver-6f77b446f6-fh85k   1/1     Running   0          3m12s
springbootserver-745788576b-qmvh9  1/1     Running   0          2m7s
```

f) Thiết lập ip cho client giao tiếp với server

**Bước 1:** Trong Google Cloud Shell gõ kubectl get svc

```
E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\ApplicationConfigurations>kubectl get svc
NAME           TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)        AGE
kubernetes     ClusterIP  34.118.224.1    <none>          443/TCP       12h
mysql-service  ClusterIP  34.118.231.60   <none>          3306/TCP      12h
socketserver-service LoadBalancer  34.118.229.29  34.101.218.243  8081:31236/TCP  6m30s
springbootserver-service LoadBalancer  34.118.232.203  34.128.104.216  8080:30764/TCP  28m
```

**Bước 2:** Vào LoadActivity trong client comment 2 dòng code:

```

IP = Utils.getIpV4(this, "my_ipv4.json");
apiUrl = String.format("http://%s:8080/api/", IP);

```

Sau đó Uncomment 2 dòng dưới đó và dán địa chỉ EXTERNAL-IP của socketserver-service vào IP, dán địa chỉ EXTERNAL-IP của springbootserver-service vào apiUrl như sau:

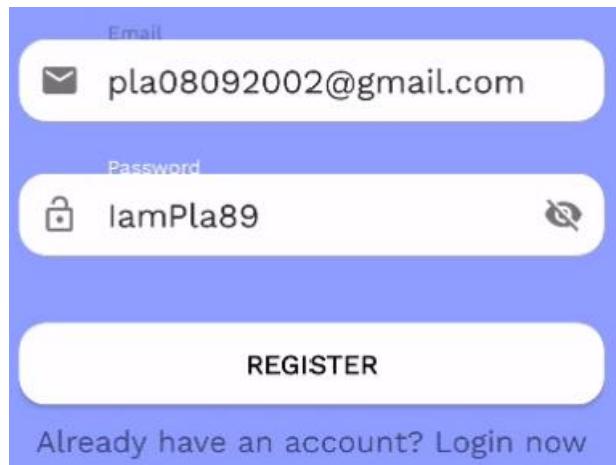
```

//          IP = Utils.getIpV4(this, "my_ipv4.json");
//          apiUrl = String.format("http://%s:8080/api/", IP);
IP = "34.101.218.243";
apiUrl = String.format("http://%s:8080/api/", "34.128.104.216");

```

### Bước 3: Chạy app và kiểm chứng

Phía Client



### Bước 4: Sau khi nhấn register vào Google Cloud SDK Shell gõ kubectl get po

```

E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\ApplicationConfigurations>kubectl get po
NAME                      READY   STATUS    RESTARTS   AGE
mysql-7bb47cbb9-s9jnz     1/1     Running   0          12h
socketserver-6f77b446f6-fh85k   1/1     Running   0          23m
springbootserver-745788576b-qmvh9 1/1     Running   0          22m

```

Sau đó copy NAME của socket server đó gõ kubectl logs {NAME vừa copy}

```

E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\ApplicationConfigurations>kubectl logs socketserver-6f77b446f6-fh85k
RECEIVED: {"requestFunction": "updateRequest", "requestParam": {"email": "\pla08092002@gmail.com", "id": 1, "password": "IamPla89", "user": {"chatRooms": [], "firstName": "Phong", "gender": "Male", "id": 1, "isOnline": false, "lastName": "Lai"}}, "token": null}
DEBUG: Account{id=1, email=pla08092002@gmail.com, password=IamPla89, user=User{id=1, lastName=Lai, firstName=Phong, dob=null, avatar=null, gender=Male, chatRooms=[]})
SENT: {"responseFunction": "updateResponse", "responseParam": {"id": 1, "email": "\pla08092002@gmail.com", "password": "IamPla89", "user": {"id": 1, "lastName": "Lai", "firstName": "Phong", "gender": "Male", "chatRooms": []}}}
SENT: {"responseFunction": "onlineUsers", "responseParam": []}
[]
```

Dữ liệu bên socket server đã nhận được thành công

Sau đó copy NAME của springboot server gõ kubectl logs {NAME vừa copy}

```

2023-08-22T04:01:58.959Z INFO 1 --- [           main] c.c.s.SpringbootserverApplication      : Started SpringbootserverApplication in 60.3 seconds (process running for 65.05)
2023-08-22T04:20:33.263Z INFO 1 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2023-08-22T04:20:33.266Z INFO 1 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2023-08-22T04:20:33.276Z INFO 1 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 8 ms
Hibernate: select a1_0.id,a1_0.created_at,a1_0.email,a1_0.password,a1_0.updated_at,a1_0.user_id from account a1_0 where a1_0.email=?
Hibernate: insert into user (avatar,dob,first_name,gender,last_name) values (?,?, ?,?, ?)
Hibernate: insert into account (created_at,email,password,updated_at,user_id) values (?, ?, ?, ?, ?)
Hibernate: select a1_0.id,a1_0.created_at,a1_0.email,a1_0.password,a1_0.updated_at,a1_0.user_id from account a1_0 where a1_0.email=?
Hibernate: select u1_0.id,u1_0.avatar,u1_0.dob,u1_0.first_name,u1_0.gender,u1_0.last_name from user u1_0 where u1_0.id=?
```

Dữ liệu đã được thêm vào bảng thành công

**Bước 5:** Gõ kubectl exec -it {NAME của mysql} -- bash và thực hiện các câu truy vấn query kiểm tra database đã được thêm vào hay chưa.

```
E:\HocTap\Nam_3\Hoc_ki_3\LapTrinhMobile\ChatApplication\k8s-configurations\ApplicationConfigurations>kubectl exec -it mysql-7bb47cddb9-s9jnz -- bash
bash-4.4# mysql -uphonglai -pphonglai
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 29
Server version: 8.1.0 MySQL Community Server - GPL

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use chattingdatabase;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

mysql> select * from account;
+-----+-----+-----+-----+
| created_at | id | password | updated_at | user_id | email |
+-----+-----+-----+-----+
| 2023-08-22 04:20:35.951059 | 1 | IamPla89 | 2023-08-22 04:20:35.951074 | 1 | pla08092002@gmail.com |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

## 2. Hướng dẫn dev

### 2. 1. Tổng quan về các API bên springboot server

#### a) Account Controller

Đối với controller này có cấu trúc api như sau bắt đầu bằng /api/account/{endpoint}, trong đó các endpoint sẽ được định nghĩa bằng các annotation mapping và đính trên một phương thức tương ứng đóng vai trò nhận và xử lý request. Tôi sẽ bắt đầu giải thích từng phương thức và endpoint đi kèm với nó đầu tiên sẽ là API lấy toàn bộ accounts GET:/api/account, phương thức getAllAccounts() sẽ trả toàn bộ tài khoản được lấy ra từ database:

```
@GetMapping  
public List<Account> getAllAccounts() {  
    return accountService.getAllAccounts();  
}
```

API dùng để gửi một yêu cầu từ hệ thống để lấy nội dung của một tài khoản GET:/api/account/{accountId} trong đó biến đường dẫn (path variable) lấy từ url phải là một số nguyên chỉ định id của một tài khoản:

```
@GetMapping(path = "{accountId}")  
public Account getAccountById(@PathVariable Long accountId) throws Exception{  
    return accountService.getAccountById(accountId);  
}
```

Hệ thống sẽ cho phép người dùng đăng ký tài khoản cho nên sẽ có API phục vụ cho mục đích đăng ký đăng nhập trên hệ thống, POST:/api/account/signup phương thức createAccount có hai tham số trong đó account là nội dung request body hệ thống sẽ yêu cầu bên phía client xử lý giao diện để gửi một request bằng phương thức POST có nội dung json như sau, trong đó không một trường value nào được phép rỗng hoặc null:

```
{  
    "email": {value},  
    "password": {value}  
}  
  
@PostMapping(path = "/signup")  
public Account createAccount(@Valid @RequestBody Account account,  
    BindingResult result) throws Exception {  
    if (result.hasErrors()) {  
        throw new Exception(result.getAllErrors().get(0).getDefaultMessage());  
    }  
    return accountService.createAccount(account);  
}
```

Hệ thống cho phép người dùng đăng nhập, sử dụng API là POST:/api/account/signin phương thức loginAccount có tham số tương tự với createAccount nhưng nội dung của request body sẽ khác yêu cầu nội dung json cung cấp đủ nội dung để đăng nhập như sau:

```
{  
    "email": {value},  
    "password": {value}  
}
```

```
@PostMapping(path = "/signin")  
public Account loginAccount(@Valid @RequestBody Account account,  
                            BindingResult result) throws Exception {  
    if (result.hasErrors()) {  
        throw new Exception(result.getAllErrors().get(0).getDefaultMessage());  
    }  
    return accountService.loginAccount(account);  
}
```

Hệ thống cho phép xóa đi một tài khoản người dùng trong cơ sở dữ liệu, sử dụng API là DELETE:/api/account/{accountId}, biến đường dẫn cần là một số nguyên chỉ định id của một tài khoản.

```
@DeleteMapping(path = "{accountId}")  
public void deleteAccount(@PathVariable("accountId") Long accountId) throws Exception {  
    accountService.deleteAccount(accountId);  
}
```

Hệ thống cho phép cập nhật thông tin của một tài khoản trong cơ sở dữ liệu PUT:/api/account/{accountId}, phương thức updateAccount có ba tham số trong đó biến môi trường {accountId} là một số nguyên chỉ định id của tài khoản, nội dung của request body là có thể bất kỳ trường nào của Entity Account, API này dùng để xử lý thay đổi mật khẩu:

```
{  
    "password": {value}  
}  
  
@PutMapping(path = "{accountId}")  
public Account updateAccount(@PathVariable("accountId") Long accountId,  
                            @Valid @RequestBody Account account,  
                            BindingResult result) throws Exception {  
    if (result.hasErrors()) {  
        throw new Exception(result.getAllErrors().get(0).getDefaultMessage());  
    }  
    return accountService.updateAccount(accountId, account);  
}
```

### b) User Controller

Đối với controller này có cấu trúc api như sau bắt đầu bằng /api/user/{endpoint}, trong đó các endpoint sẽ được định nghĩa bằng các annotation mapping và đính trên một phương thức tương ứng đóng vai trò nhận và xử lý request. Tôi sẽ bắt đầu giải thích từng phương thức và endpoint đi kèm với nó đầu tiên sẽ là API lấy toàn bộ user GET:/api/user, phương thức getAllUsers() sẽ trả toàn bộ thông tin người dùng được lấy ra từ database:

```
@GetMapping  
public List<User> getAllUsers() {  
    return userService.getAllUsers();  
}
```

API dùng để gửi một yêu cầu từ hệ thống để lấy nội dung thông tin người dùng GET:/api/user/{userId} trong đó biến đường dẫn (path variable) lấy từ url phải là một số nguyên chỉ định id của một người dùng:

```
@GetMapping(path = "{userId}")  
public User getUserById(@PathVariable Long userId) throws Exception{  
    return userService.getUserById(userId);  
}
```

Hệ thống cần xử lý khi người dùng đăng ký trên ứng dụng sẽ cần tạo Account Entity và một User Entity rỗng để sau khi người dùng nhập email và password, sẽ có một User Entity chưa có nội dung để nhập thông tin và lưu trên đó cho nên tạo ra một POST:/api/user để gọi đến server tạo một user entity rỗng đó, chính vì vậy ở phía client sau khi đã gọi POST:/api/account/signup cũng cần xử lý gọi chính API này.

```
@PostMapping  
public User createUser(@Valid @RequestBody User user,  
        BindingResult result) throws Exception {  
    if (result.hasErrors()){  
        throw new Exception(result.getAllErrors().get(0).getDefaultMessage());  
    }  
    return userService.createUser(user);  
}
```

Hệ thống cho phép xóa đi thông tin của người dùng DELETE:/api/user/{userId}, trong đó biến đường dẫn sẽ là một số nguyên chỉ định id của một người dùng cho nên annotation @PathVariable được đính vào biến Long userId.

```

@DeleteMapping(path = "{userId}")
public void deleteAccount(@PathVariable("userId") Long userId) throws Exception {
    userService.deleteUser(userId);
}

```

Hệ thống cho phép cập nhật thông tin người dùng, cụ thể hơn ở phía client sau khi đã xử lý cho phép người dùng nhập thông tin email và password thì trên hệ thống đã tồn tại một User Entity chưa có nội dung gì và chờ người dùng nhập tin cụ thể, lập trình viên ở phía client cần thiết kế giao diện cho người dùng nhập các thông tin đó và gọi vào PATCH:/api/user/{userId}, phương thức updateUser có ba tham số trong đó biến đường dẫn userId được định vào Long userId chỉ định id của một người dùng và nội dung của request body được định vào User user. Nội dung của một json gửi lên server sẽ bao gồm các trường {value} không được phép rỗng hay null như sau:

```

{
    "firstName": {value},
    "lastName": {value},
    "dob": {value},
    "gender": {value}
}

```

```

@PatchMapping(path = "{userId}")
public User updateUser(@PathVariable("userId") Long userId,
    @Valid @RequestBody User user, BindingResult result) throws Exception {
    if (result.hasErrors()) {
        throw new Exception(result.getAllErrors().get(0).getDefaultMessage());
    }
    return userService.updateUser(userId, user);
}

```

Hệ thống cho phép người dùng được đăng ảnh đại diện lên ứng dụng bằng POST:/api/user/upload\_avatar/{userId} api này yêu cầu content type là multipart/form-data để xử lý nhận một file từ bên client, phương thức uploadAvatar có hai tham số trong đó biến đường dẫn userId sẽ được định vào Long id là một số nguyên chỉ định id của một người dùng và User user có thuộc tính MultipartFile uploadAvatar để nhận multipart file gửi từ request. Ở bên phía client cần xử lý cho người dùng chọn một file từ thiết bị và gửi request content type cần phải là multipart/form-data. API sẽ trả về một chuỗi secure URL là ảnh đại diện người dùng đã được lưu trên cloudinary.

```

@PostMapping(path = "upload_avatar/{userId}")
public ResponseEntity<String> uploadAvatar(@PathVariable(name = "userId")
    Long id, User user) throws IOException{
    System.out.println("=====USER" + user);
    return uploadFileService.uploadAvatar(id, user);
}

```

Hệ thống cho phép tìm kiếm người dùng theo từ khóa, GET:/api/user/search/{keyword} trong đó biến đường dẫn keyword là một chuỗi đính vào String keyword chỉ định từ khóa để tìm kiếm, api sẽ trả về một danh sách người dùng dựa trên từ khóa tìm kiếm được.

```
@GetMapping(path = "/search/{keyword}")
public List<User> searchUser(@PathVariable(name = "keyword")
|   String keyword) throws Exception {
    return userService.searchUser(keyword);
}
```

### c) ChatRoom Controller

Đối với controller này có cấu trúc api như sau bắt đầu bằng /api/chat\_room/{endpoint}, trong đó các endpoint sẽ được định nghĩa bằng các annotation mapping và đính trên một phương thức tương ứng đóng vai trò nhận và xử lý request. Tôi sẽ bắt đầu giải thích từng phương thức và endpoint đi kèm với nó đầu tiên sẽ là API lấy toàn bộ phòng nhắn tin GET:/api/chat\_room, phương thức getChatRooms () sẽ trả toàn bộ phòng nhắn tin được lấy ra từ database:

```
@GetMapping
public List<ChatRoom> getChatRooms() {
    return chatRoomService.getChatRooms();
}
```

Hệ thống sẽ cần tạo ra một phòng nhắn tin riêng cho từng đối tượng người dùng, POST:/api/chat\_room/create\_chat\_room ở phía người lập trình viên client cần xử lý khi nhắn vào một tài khoản trên danh sách người dùng cần phải tạo ra một phòng nhắn tin rõ ràng chưa tồn tại tài khoản nào. Phương thức createChatRoom có hai tham số trong đó nội dung của request body được gửi từ request sẽ tương ứng với biến ChatRoom chatRoom, json sẽ có cấu trúc như sau:

```
{
    "roomName": {value},
    "isPrivate": {value}
}
```

```
@PostMapping("/create_chat_room")
public ChatRoom createChatRoom(@Valid @RequestBody ChatRoom chatRoom,
|   BindingResult result) throws Exception {
    if (result.hasErrors()) {
        throw new Exception(result.getAllErrors().get(0).getDefaultMessage());
    }

    return chatRoomService.createChatRoom(chatRoom);
}
```

Hệ thống cho phép được thêm các người dùng vào trong phòng nhắn tin, khi một phòng nhắn tin rỗng được tạo ra người lập trình viên ở phía client cần phải gọi POST/api/chat\_room/{room\_id}/add\_users để thêm một danh sách người dùng vào phòng nhắn tin, phương thức addUsers gồm hai tham số trong đó biến đường dẫn là một số nguyên chỉ định id của một phòng và nội dung của request body sẽ là danh sách các người dùng, cho nên cấu trúc của json sẽ như sau:

```
{  
    [  
        {  
            {  
                "id": {value}  
            },  
            {  
                "id": {value}  
            }  
        ]  
    }  
}
```

```
@PostMapping(path = "{room_id}/add_users")  
public String addUsers(@PathVariable(name = "room_id") Long chatRoomId,  
    @RequestBody List<User> users)  
    throws Exception {  
    try {  
        return chatRoomService.addUsers(chatRoomId, users);  
    } catch (Exception e) {  
        throw new Exception(e.getMessage());  
    }  
}
```

Hệ thống cho phép tạo ra một phòng chat riêng tư chỉ bao gồm hai đối tượng người dùng, người lập trình viên ở client cần phải xử lý khi nhấn vào một tài khoản trong danh sách tài khoản và người dùng thực hiện gửi một tin nhắn bất kỳ mới bắt đầu tiên hành tạo ra loại phòng nhắn tin này bằng cách gọi POST:/api/chat\_room/create\_private\_room/{user\_id\_created}/{user\_id\_targed}.

Phương thức createPrivateRoom sẽ bao gồm bốn tham số trong đó hai biến đường dẫn lần lượt là id của người dùng hiện tại và id của đối tượng mà người dùng nhấn vào, nội dung của request body sẽ có cấu trúc json như sau:

```
{  
    "content": {value}  
}
```

```

@PostMapping(path = "/create_private_room/{user_id_created}/{user_id_targed}")
public ChatRoom createPrivateRoom(@PathVariable Long user_id_created,
    @PathVariable Long user_id_targed,
    @Valid @RequestBody Message message, BindingResult result) throws Exception {
    try {
        return chatRoomService.createPrivateRoom(user_id_created, user_id_targed, message);
    } catch (Exception e) {
        throw new Exception(e.getMessage());
    }
}

```

POST:/api/chat\_room/{room\_id}/add\_user API này dùng để thêm một người dùng vào một phòng nhắn tin, biến đường dẫn chatRoomId là một số nguyên chỉ định id của một phòng nhắn tin và nội dung của request body được đính với User user có cấu trúc json như sau:

```

{
    "id": {value}
}

```

```

@PostMapping(path = "{room_id}/add_user")
public User addUser(@PathVariable(name = "room_id") Long chatRoomId,
    @RequestBody User user) throws Exception {
    try {
        System.out.println("target user: " + user);
        return chatRoomService.addUser(chatRoomId, user);
    } catch (Exception e) {
        throw new Exception(e.getMessage());
    }
}

```

Hệ thống cho phép xóa đi phòng nhắn tin DELETE:/api/chat\_room/{room\_id} phương thức deleteChatRoom chỉ có một tham số là biến đường dẫn có kiểu số nguyên chỉ định id của một phòng nhắn tin.

```

@DeleteMapping(path = "{room_id}")
public void deleteChatRoom(@PathVariable(name = "room_id") Long chatRoomId)
    throws Exception {
    try {
        chatRoomService.deleteChatRoom(chatRoomId);
    } catch (Exception e) {
        throw new Exception(e.getMessage());
    }
}

```

Giao diện ở phía client cần phải có danh sách các phòng nhắn tin để người dùng có thể nhấp vào sử dụng và danh sách đó phải thuộc của riêng một người dùng, chính vì vậy cần gọi GET:/api/chat\_room/{user\_id} phương thức getChatRoomsByUserId có

một tham số duy nhất là biến đường dẫn userId có kiểu số nguyên chỉ định id của một người dùng và trả về một danh sách các phòng nhắn tin của đối tượng người dùng đó.

```
@GetMapping(path = "{user_id}")
public Set<ChatRoom> getChatRoomsByUserId(@PathVariable(name = "user_id") Long userId){
    return chatRoomService.getChatRoomsByUserId(userId);
}
```

Giao diện ở phía client cũng cần lấy ra thông tin của một phòng nhắn tin khi biết được id của người dùng và đối tượng của người dùng đó, GET:/api/chat\_room/{current\_user\_id}/{target\_user\_id}/{is\_private} phương thức getChatRoomByUsers bao gồm ba tham số trong đó lần lượt là các biến đường dẫn kiểu số nguyên chỉ định id của người dùng hiện, id của đối tượng người dùng và biến đường dẫn kiểu luận lý cho biết thuộc tính isPrivate của phòng nhắn tin.

```
@GetMapping(path = "{current_user_id}/{target_user_id}/{is_private}")
public ChatRoom getChatRoomByUsers(@PathVariable(name = "current_user_id")
    Long currentUser, @PathVariable(name = "target_user_id")
    Long targetUser, @PathVariable(name = "is_private")
    boolean isPrivate) throws Exception {
    try {
        return chatRoomService.getChatRoomByUsers(currentUser, targetUser, isPrivate);
    } catch (Exception e) {
        throw new Exception(e.getMessage());
    }
}
```

#### d) Message Controller

Đối với controller này có cấu trúc api như sau bắt đầu bằng /api/message/{endpoint}, trong đó các endpoint sẽ được định nghĩa bằng các annotation mapping và đính trên một phương thức tương ứng đóng vai trò nhận và xử lý request. Tôi sẽ bắt đầu giải thích từng phương thức và endpoint đi kèm với nó đầu tiên sẽ là API lấy toàn bộ phòng nhắn tin GET:/api/message/{room\_id}, phương thức getAllMessages nhận một tham số duy nhất là biến đường dẫn room\_id kiểu số nguyên chỉ định id của phòng nhắn tin sẽ trả toàn bộ tin nhắn được lấy ra từ database:

```
@GetMapping("{room_id}")
public List<Message> getAllMessages(@PathVariable(name = "room_id")
    Long chatRoomId) {
    return messageService.getAllMessages(chatRoomId);
}
```

Để hiện thực hóa được việc nhắn tin qua lại giữa các người dùng cần phải tạo ra api đáp ứng được lưu một tin nhắn vào cơ sở dữ liệu, gọi POST:/api/message/{room\_id}/{user\_id} phương thức createMessage có bốn tham số trong đó hai biến đường dẫn kiểu số nguyên lần lượt chỉ định id của phòng nhắn tin và

id của người dùng, nội dung của request body sẽ tương ứng với Message message có cấu trúc json như sau:

```
{  
    "content": {value}  
}
```

```
@PostMapping("/{room_id}/{user_id}")  
public Message createMessage(@PathVariable(name = "room_id") Long chatRoomId,  
    @PathVariable(name = "user_id") Long userId,  
    @Valid @RequestBody Message message, BindingResult result) throws Exception {  
    if (result.hasErrors()) {  
        throw new Exception(result.getAllErrors().get(0).getDefaultMessage());  
    }  
    return messageService.createMessage(chatRoomId, userId, message);  
}
```

## 2. 2. Tổng quan các hàm của socket server

### a) Lớp ClientHandleService

Là 1 lớp hiện thực lại interface Runnable để trở thành 1 Thread đại diện cho 1 client, khi 1 socket client được kết nối, constructor của lớp này sẽ được gọi để khởi tạo 1 Thread cho client đó. Phương thức khởi tạo gồm Socket của client và tạo ra 1 DataInputStream riêng cho client đó để nhận request từ socket tương ứng từ client, 1 DataOutputStream tương ứng để gửi request từ socket tương ứng từ client.

```
public ClientHandleService(Socket clientSocket) {  
    try {  
        this.clientSocket = clientSocket;  
        this.dIn = new DataInputStream(clientSocket.getInputStream());  
        this.dOut = new DataOutputStream(clientSocket.getOutputStream());  
    } catch (IOException e) {  
        ServerService.removeClient(this);  
    }  
}
```

Hàm run là phương thức trừu tượng của interface Runnable là hàm mà Thread sẽ thực thi, khi Thread ClientHandleService được tạo ra, hàm run sẽ chạy và liên tục đọc request từ socket client cho tới khi client đó ngắt kết nối, vòng lặp sẽ dừng và Thread sẽ tự hủy.

```

@Override
public void run() {
    while (clientSocket.isConnected()) {
        String buffer = ServerService.socketReceive(this);
        if (buffer == null) {
            ServerService.removeClient(this);
            break;
        } else {
            try {
                ServerService.handleRequest(this, buffer);
            } catch (NoSuchMethodException e) {
                e.printStackTrace();
            } catch (SecurityException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                e.printStackTrace();
            }
        }
    }
    ServerService.removeClient(this);
}

```

Nếu chuỗi buffer khác null gọi phương thức handleRequest từ lớp ServerService.

#### b) Lớp RequestService

Lớp này chứa các phương thức GET POST PUT DELETE để SocketServer giao tiếp với Springboot Server.

Hàm getRequest dùng để gọi phương thức GET từ Springboot Server để lấy dữ liệu từ database, parameter là đường dẫn để gọi API.

```

public static String getRequest(String path) throws IOException, InterruptedException {
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create(apiUrl + path))
        .build();
    HttpResponse<String> response = client.send(request,
        HttpResponse.BodyHandlers.ofString());
    if (response.statusCode() == 200) {
        return response.body();
    } else {
        return "Oops something went wrong!";
    }
}

```

Hàm postRequest dùng để gọi phương thức POST từ Springboot Server để thêm dữ liệu từ database, parameter là đường dẫn để gọi API và jsonString như 1 requestBody.

```

public static String postRequest(String path, String jsonString) throws IOException, InterruptedException {
    HashMap<String, Object> values = new Gson().fromJson(jsonString, classOfT:HashMap.class);

    ObjectMapper objectMapper = new ObjectMapper();
    String requestBody = objectMapper
        .writeValueAsString(values);

    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create(apiUrl + path))
        .setHeader(name:"Content-type", value:"application/json")
        .POST(HttpRequest.BodyPublishers.ofString(requestBody))
        .build();

    HttpResponse<String> response = client.send(request,
        HttpResponse.BodyHandlers.ofString());

    if (response.statusCode() == 200) {
        return response.body();
    } else {
        Gson gson = new Gson();
        ExceptionError exceptionError = gson.fromJson(response.body(), classOfT:ExceptionError.class);
        return exceptionError.getMessage();
    }
}

```

Hàm putRequest tương tự như postRequest nhưng dùng để gọi phương thức PUT từ Springboot Server để sửa đổi dữ liệu từ database.

```

public static String putRequest(String path, String jsonString) throws IOException, InterruptedException {
    HashMap<String, Object> values = new Gson().fromJson(jsonString, classOfT:HashMap.class);

    ObjectMapper objectMapper = new ObjectMapper();
    String requestBody = objectMapper
        .writeValueAsString(values);

    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create(apiUrl + path))
        .setHeader(name:"Content-type", value:"application/json")
        .PUT(HttpRequest.BodyPublishers.ofString(requestBody))
        .build();

    HttpResponse<String> response = client.send(request,
        HttpResponse.BodyHandlers.ofString());

    if (response.statusCode() == 200) {
        return response.body();
    } else {
        Gson gson = new Gson();
        ExceptionError exceptionError = gson.fromJson(response.body(), classOfT:ExceptionError.class);
        return exceptionError.getMessage();
    }
}

```

Hàm deleteRequest dùng để gọi phương thức DELETE từ Springboot Server để xóa dữ liệu từ database, parameter là đường dẫn để gọi API.

```

public static String deleteRequest(String path) throws IOException, InterruptedException {
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create(apiUrl + path))
        .DELETE()
        .build();
    HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
    if (response.statusCode() == 200) {
        return "Delete Success!";
    } else {
        com.chattingapplication.model.ExceptionError
        ExceptionError exceptionError = gson.fromJson(response.body(), classOfT:ExceptionError.class);
        return exceptionError.getMessage();
    }
}

```

### c) Lớp ServerService

Lớp này chứa các xử lý logic và các hàm dùng chung của các ClientHandleService, quản lý danh sách các ClientHandleService.

Hàm handleConnect dùng để khởi tạo vòng lặp cho tới khi tắt server, khi có socket client kết nối vào, socket client đó sẽ được khởi tạo thành 1 ClientHandleService và được thêm vào danh sách chứa tất cả ClientHandleService trên socket server.

```

public static void handleConnect(ServerSocket serverSocket) throws IOException {
    System.out.println("Server started!");
    while (!serverSocket.isClosed()) {
        Socket clientSocket = serverSocket.accept();
        ClientHandleService clientHandleService = new ClientHandleService(clientSocket);
        clientHandlers.add(clientHandleService);
        Thread thread = new Thread(clientHandleService);
        thread.start();
    }
}

```

Hàm handleRequest được gọi khi client gửi request là 1 JSON String gồm requestFunction và requestParam, hàm sử dụng thư viện gson để parse json string sang đối tượng Request để lấy 2 thuộc tính trên, sau đó gọi Method để lấy hàm có tên trùng với requestFunction và loại parameter được định nghĩa sẵn sẽ là ClientHandleService.class để truyền ClientHandleService của client gọi request, String.class để truyền requestParam từ JSON, sau đó gọi phương thức invoke để chạy hàm method vừa get được.

```

public static void handleRequest(ClientHandleService clientHandleService, String jsonRequest)
throws NoSuchMethodException, SecurityException, IllegalAccessException, InvocationTargetException {
    Gson gson = new Gson();
    Request request = gson.fromJson(jsonRequest, classOfT:Request.class);
    Method method = ServerService.class.getMethod(request.getRequestFunction(), ...parameterTypes:ClientHandleService.class ,String.class];
    method.invoke(obj:null, clientHandleService, request.getRequestParam());
}

```

Hàm chattingRequest được gọi khi client gửi tin nhắn hàm sẽ chuyển jsonString sang jsonObject để lấy các key content, roomId, userId từ JSON của phía client sau đó gọi hàm saveMessage.

```

public static void chattingRequest(ClientHandleService clientHandleService, String jsonString)
throws JSONException, IOException, InterruptedException {
    JSONObject requestObject = new JSONObject(jsonString);
    saveMessage(clientHandleService, requestObject.getString("content"), requestObject.getLong("roomId"), requestObject.getLong("userId"));
}

```

Hàm saveMessage dùng để lưu tin nhắn phía client đã gửi vào database bằng cách gọi postRequest từ RequestService, sau đó gọi hàm sendMessage.

```

public static void saveMessage(ClientHandleService clientHandleService, String content, Long roomId, Long userId)
throws IOException, InterruptedException {
    String message;
    try {
        message = new JSONObject()
            .put("content", content)
            .toString();
    } catch (JSONException e) {
        throw new RuntimeException(e);
    }
    String messageResponse = RequestService.postRequest(String.format("message/%d/%d", roomId, userId), message);
    sendMessage(clientHandleService, messageResponse);
}

```

Hàm sendMessage dùng để gửi tin nhắn cho client khác, sử dụng gson để parse messageJson được lấy từ response của API Springboot server sau đó gửi đến ClientHandleService nào có user có chatroom giống với chatroom của message vừa trả về bằng hàm socketSend.

```

public static void sendMessage(ClientHandleService clientHandleService, String messageJson) {
    Gson gson = new Gson();
    Message message = gson.fromJson(messageJson, classOfT:Message.class);
    clientHandlers.stream()
        .filter(c -> c.getClientSocket() != clientHandleService.getClientSocket() &&
            c.getClientAccount() != null && c.getClientAccount().getUser() != null &&
            c.getClientAccount().getUser().getChatRooms().contains(message.getChatRoom()))
        .forEach(client -> {
            socketSend(client, responseName:"chattingResponse", messageJson);
        });
}

```

Hàm socketSend gửi response tới socket client dưới dạng JSON String gồm 2 keys là responseFunction là tên function client sẽ thực hiện response và responseParam để client truyền vào parameter, như ở hàm sendMessage, phía client sẽ invoke hàm có tên chattingResponse và truyền messageJson vào parameter.

```

public static void socketSend(ClientHandleService clientHandleService, String responseName, String responseParam) {
    String message;
    try {
        message = new JSONObject()
            .put("responseFunction", responseName)
            .put("responseParam", responseParam)
            .toString();
    } catch (JSONException e) {
        throw new RuntimeException(e);
    }
    try {
        System.out.printf(format:"SENT: %s\n", message);
        clientHandleService.getout().writeUTF(message);
        clientHandleService.getout().flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

Hàm updateRequest sẽ được thực hiện khi client vào giao diện chính và gửi request tới socket server gồm requestFunction là updateRequest và requestParam là accountJson là Json của tài khoản mà client lấy được từ Springboot Server, sau đó socket server parse accountJson đó sang Account và lưu vào clientHandleService gửi request thông qua hàm updateCurrentClient, sau đó sẽ gửi response cho socketclient để báo đã update thành công.

```

public static void updateRequest(ClientHandleService clientHandleService, String accountJson)
throws JsonSyntaxException, IOException, InterruptedException {
    Gson gson = new Gson();
    clientHandleService.setClientAccount(gson.fromJson(accountJson, classOfT:Account.class));
    updateCurrentClient(clientHandleService);
    System.out.printf(format:"DEBUG: %s\n", clientHandleService.getClientAccount().toString());
    socketSend(clientHandleService, responseName:"updateResponse", gson.toJson(clientHandleService.getClientAccount()));
}

```

Hàm updateCurrentClient dùng để cập nhật thông tin của ClientHandleService đang được socket server quản lý hàm sẽ GET các phòng chat của ClientHandleService từ Springboot Server và lưu vào ClientHandleService được truyền vào.

```

public static void updateCurrentClient(ClientHandleService clientHandleService)
throws JsonSyntaxException, IOException, InterruptedException {
    Gson gson = new Gson();
    String roomJson = RequestService.getRequest(String.format(format:"chat_room/%d", clientHandleService.getClientAccount().getUser().getId()));
    JSONArray jsonArray = new JSONArray(roomJson);
    List<ChatRoom> listChat = gson.fromJson(jsonArray.toString(), new TypeToken<List<ChatRoom>>() {}.getType());
    clientHandleService.getClientAccount().getUser().setChatRooms(listChat);
}

```

Hàm `createPrivateRoomRequest` được gọi khi client thực hiện chat tin nhắn đầu tiên với User bất kỳ, hàm sẽ lấy `createUser`, `targetUser`, `firstMessage` từ `jsonString` phía client, sau đó lấy `ClientHandleService` tương ứng với id của `createUser` và `targetUser`. `targetUser` đang offline sẽ throw ra `Exception`, khi đó gọi hàm `createPrivateRoom`, nếu cả 2 đều đang online, gọi hàm `synchronizedCreatePrivateRoom` sau đó cập nhật `targetClient` và gửi cho `targetClient` JSON String gồm `responseFunction` là `createPrivateRoomResponse`, với parameter là json của phòng chat vừa được tạo lấy từ `Springboot Server`. Sau khi thực hiện xong, cập nhật `clientHandleService` gửi request.

```

public static String createPrivateRoomRequest(ClientHandleService clientHandleService, String jsonString)
throws IOException, InterruptedException {
    Gson gson = new Gson();
    JSONObject requestObject = new JSONObject(jsonString);
    User createUser = gson.fromJson(requestObject.getString("createUser"), classOfT:User.class);
    User targetUser = gson.fromJson(requestObject.getString("targetUser"), classOfT:User.class);
    String firstMessage = requestObject.getString("message");
    String newChatRoom;
    try {
        ClientHandleService targetClient = clientHandlers.stream()
            .filter(c -> c.getClientAccount().getUser().getId().equals(targetUser.getId())).findFirst().get();
        ClientHandleService firstClient = (clientHandleService.getClientAccount().getUser().getId() <
            targetClient.getClientAccount().getUser().getId()) ? clientHandleService: targetClient;
        ClientHandleService secondClient = (clientHandleService.getClientAccount().getUser().getId() <
            targetClient.getClientAccount().getUser().getId()) ? targetClient: clientHandleService;
        newChatRoom = synchronizedCreatePrivateRoom(targetUser, createUser, clientHandleService, firstClient, secondClient, firstMessage);
        updateCurrentClient(targetClient);
        String responseRoom = RequestService.
            getRequest(String.format(format:"chat_room/%d/%d/true", targetUser.getId(), createUser.getId()));
        socketSend(targetClient, responseName:"createPrivateRoomResponse", responseRoom);
    } catch (NoSuchElementException | NullPointerException e) {
        newChatRoom = createPrivateRoom(targetUser, createUser, clientHandleService, createUser.getId(), targetUser.getId(), firstMessage);
    }
    updateCurrentClient(clientHandleService);
    return "";
}

```

Hàm `createPrivateRoom` dùng để tạo phòng chat giữa 2 User, đầu tiên gọi API GET phòng chat từ `Springboot server`, nếu phòng chat đã tồn tại, gửi JSON String gồm `responseFunction` là `createPrivateRoomResponse` và `responseParam` là `JSON` của phòng Chat vừa lấy được từ API `Springboot server`, nếu phòng chat chưa tồn tại sẽ throw `Exception` và thực hiện lưu phòng chat mới vào database thông qua gọi API POST của `Springboot server` và gửi cho socket client gửi request thông tin phòng chat vừa tạo từ trả về của API.

```

public static String createPrivateRoom(User targetUser, User createUser, ClientHandleService clientHandleService,
Long firstId, Long secondId, String firstMessage) throws IOException, InterruptedException {
    String checkRoom = RequestService.getRequest(String.format(format:"chat_room/%d/%d/true", firstId, secondId));
    Gson gson = new Gson();
    try {
        ChatRoom checkChatRoom = gson.fromJson(checkRoom, classOf:ChatRoom.class);
        socketSend(clientHandleService, responseName:"createPrivateRoomResponse", checkRoom);
        saveMessage(clientHandleService, firstMessage, checkChatRoom.getId(), createUser.getId());
        return checkRoom;
    } catch (JsonSyntaxException ex) {
        String messageJson;
        try {
            messageJson = new JSONObject()
                .put("content", firstMessage)
                .toString();
        } catch (JSONException e) {
            throw new RuntimeException(e);
        }
        String responseBody = RequestService.postRequest(String.format(format:"chat_room/create_private_room/%d/%d",
            createUser.getId(), targetUser.getId()), messageJson);
        socketSend(clientHandleService, responseName:"createPrivateRoomResponse", responseBody);
        return responseBody;
    }
}

```

Hàm synchronizedCreatePrivateRoom dùng để cô lập việc gọi hàm createPrivateRoom chỉ cho 1 Thread được gọi để phòng trường hợp cả 2 user có cùng createUser và targetUser đều gửi request tạo phòng chat mới cùng 1 lúc. Tránh được trường hợp 2 User đó nhắn cùng 1 lúc tạo ra 2 phòng chat khác nhau.

```

public static String synchronizedCreatePrivateRoom(User targetUser, User createUser, ClientHandleService clientHandleService,
ClientHandleService firstClient, ClientHandleService secondClient, String firstMessage) throws IOException, InterruptedException {
    synchronized (firstClient) {
        synchronized (secondClient) {
            return createPrivateRoom([targetUser, createUser, clientHandleService ,firstClient.getClientAccount().getUser().getId(),
            secondClient.getClientAccount().getUser().getId(), firstMessage]);
        }
    }
}

```

Hàm socketReceive dùng để nhận request từ client, được dùng trong vòng lặp của Thread ClientHandleService.

```

public static String socketReceive(ClientHandleService clientHandleService) {
    try {
        String bufferIn = clientHandleService.getdIn().readUTF();
        System.out.printf(format:"RECEIVED: %s\n", bufferIn);
        return bufferIn;
    } catch (EOFException e) {
        clientHandleService.closeClientSocket();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}

```

Hàm removeClient dùng để xóa client khỏi danh sách các clientHandleService khi client đó tắt hoặc bị lỗi, hàm shutDownServer dùng để tắt server được khai báo khi cần sử dụng.

```

public static void removeClient(ClientHandleService clientHandleService) {
    clientHandleService.closeClientSocket();
    ServerService.clientHandlers.remove(clientHandleService);
}

public static void shutDownServer(ServerSocket serverSocket) throws IOException {
    if (serverSocket != null) {
        serverSocket.close();
    }
}

```

## 2. 3. Tổng quan các hàm của client

### a) Gói AsyncTask

Chứa các lớp kế thừa lớp AsyncTask để xử lý bất đồng bộ

Hàm doInBackground() của ConnectTask được sử dụng để kết nối tới socket server.

```

► PlalsMe
@Override
protected Void doInBackground(Void... voids) {
    try {
        clientFd = new Socket(LoadActivity.IP, LoadActivity.PORT);
        dOut = new DataOutputStream(clientFd.getOutputStream());
        dIn = new DataInputStream(clientFd.getInputStream());
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    return null;
}

```

Sau khi kết nối xong, hàm onPostExecute được thực thi nhằm tạo 1 thread khác để luôn lắng nghe response từ Server.

```

    ▲ PlalsMe
    @Override
    protected void onPostExecute(Void unused) {
        super.onPostExecute(unused);
        ▲ PlalsMe
        new Thread(new Runnable() {
            ▲ PlalsMe
            @Override
            public void run() {
                while (clientFd.isConnected()) {
                    try {
                        socketResponse = dIn.readUTF();
                        ((LoadActivity) activity).handleSocketResponse(socketResponse);
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                }
            }
        }).start();
    }
}

```

Sau khi tạo xong Thread, tiến hành kiểm tra xem người dùng có lưu thông tin đăng nhập hay không, nếu không thì vào authenticationActivity để login, register, nếu có thêm extra navigate là true để khi vào activity bỏ qua bước đăng nhập.

```

String loginSuccess = LoadActivity.preferencesCheck.getString( key: "check",  defaultValue: "false");
if (loginSuccess.equals("true")) {
    Gson gson = new Gson();
    LoadActivity.currentAccount = gson.fromJson(LoadActivity.preferencesAccount.getString( key: "account",  defaultValue: ""),  Account.class);
    try {
        String checkName = LoadActivity.currentAccount.getUser().getFirstName();
        Log.d( tag: "debugCheckName",  checkName);
        SendTask sendTask = new SendTask();
        sendTask.execute( ...params: "updateRequest",  String.format("%s",  gson.toJson(LoadActivity.currentAccount)));
    } catch (NullPointerException e) {
        Intent authenticationActivity = new Intent(this.activity, AuthenticationActivity.class);
        authenticationActivity.putExtra( name: "navigate",  value: "true");
        activity.startActivity(authenticationActivity);
    }
} else {
    Intent authenticationActivity = new Intent(this.activity, AuthenticationActivity.class);
    activity.startActivity(authenticationActivity);
}

```

Hàm doInBackground() của DownloadFromURLTask được sử dụng để tải hình bất đồng bộ, để chương trình không phải chờ tải hình xong mới tiếp tục thực thi.

```

@Override
protected Bitmap doInBackground(String... params) {
    Bitmap bitmap = null;
    try {
        if(params[0] != null){
            URL clouddinaryURL = new URL(params[0]);
            bitmap = BitmapFactory.decodeStream(
                clouddinaryURL.openConnection().getInputStream()
            );
        }
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    return bitmap;
}

```

Sau khi tải hình xong, kiểm tra nếu hình không null, gán hình vào user được truyền vào.

```
@Override  
protected void onPostExecute(Bitmap bitmap) {  
    super.onPostExecute(bitmap);  
    if(bitmap != null && user != null){  
        user.setDownloadAvatar(bitmap);  
        imageView.setImageBitmap(bitmap);  
    }  
}
```

Hàm doInBackground() của GetRequestTask được sử dụng để gọi API GET từ Springboot server để lấy dữ liệu từ database, tham số cần truyền vào là đường dẫn API, tên hàm invoke sau khi thực thi, các hàm này sẽ được đặt trong lớp HttpResponse.

```
@Override  
protected String doInBackground(String... params) {  
    try {  
        URL url = new URL(spec: LoadActivity.apiUrl + params[0]);  
        Log.d(tag: "debugGetURL", msg: LoadActivity.apiUrl + params[0]);  
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();  
        conn.setRequestMethod("GET");  
  
        responseCode = conn.getResponseCode();  
        functionName = params[1];  
  
        BufferedReader bufferedReader = (responseCode == HttpURLConnection.HTTP_OK ?  
            new BufferedReader(new InputStreamReader(conn.getInputStream())) :  
            new BufferedReader(new InputStreamReader(conn.getErrorStream())));  
        String inputLine;  
        StringBuffer response = new StringBuffer();  
        while ((inputLine = bufferedReader.readLine()) != null) {  
            response.append(inputLine);  
        }  
        bufferedReader.close();  
        return response.toString();  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }  
}
```

Hàm doInBackground() của PostRequestTask được sử dụng để gọi API POST từ Springboot server để thêm dữ liệu mới vào database hoặc đăng nhập, đăng ký, tham số cần truyền vào là đường dẫn API, jsonString như 1 requestBody của API, tên hàm invoke sau khi thực thi, các hàm này sẽ được đặt trong lớp HttpResponse.

Tương tự như PostRequestTask, PutRequestTask được sử dụng để gọi API PUT và có chung tham số như PostRequestTask để update dữ liệu trong database.

Tương tự như PutRequestTask, PatchRequestTask được sử dụng để gọi API PATCH và có cùng tham số để update 1 phần đối tượng trong database.

```

@Override
protected String doInBackground(String... params) {
    try {
        URL url = new URL(spec: LoadActivity.apiUrl + params[0]);
        Log.d(tag: "debugPostURL", msg: LoadActivity.apiUrl + params[0]);
        Log.d(tag: "debugPostContent", params[1]);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("POST");
        conn.setRequestProperty("Content-Type", "application/json");
        OutputStreamWriter osw = new OutputStreamWriter(conn.getOutputStream());
        osw.write(params[1]);
        osw.flush();
        osw.close();
        responseCode = conn.getResponseCode();
        functionName = params[2];
        BufferedReader bufferedReader = (responseCode == HttpURLConnection.HTTP_OK ?
            new BufferedReader(new InputStreamReader(conn.getInputStream())) :
            new BufferedReader(new InputStreamReader(conn.getErrorStream())));
        String inputLine;
        StringBuffer response = new StringBuffer();
        while ((inputLine = bufferedReader.readLine()) != null) {
            response.append(inputLine);
        }
        bufferedReader.close();
        return response.toString();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

```

Hàm doInBackground() của SendTask được sử dụng để gửi request tới socket server, với parameters truyền vào là tên function cần invoke bên socket server và params truyền vào socket server dạng JSON String.

```

protected Void doInBackground(String... params) {
    String request;
    try {
        request = new JSONObject()
            .put(name: "requestFunction", params[0])
            .put(name: "requestParam", params[1])
            .toString();
    } catch (JSONException e) {
        throw new RuntimeException(e);
    }
    try {
        // Gửi message qua client
        Log.d(tag: "debugSent", request);
        ConnectTask.dOut.writeUTF(request);
        ConnectTask.dOut.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}

```

Hàm doInBackground() của UploadFileTask dùng để upload avatar với tham số truyền vào là đường dẫn gọi API upload avatar của Springboot server với id của user và tên của file cần upload.

```

@Override
protected String doInBackground(String... params) {
    try {
        if(uri == null){
            return null;
        }
        URL apiUrl = new URL(spec LoadActivity.apiUrl + params[0]);
        HttpURLConnection connection = (HttpURLConnection) apiUrl.openConnection();

        Log.d(tag: "debugPostURL", msg: LoadActivity.apiUrl + params[0]);

        connection.setRequestMethod("POST");
        connection.setRequestProperty("Content-Type", "multipart/form-data; boundary=" + BOUNDARY);
        connection.setDoOutput(true);

        DataOutputStream output = new DataOutputStream(connection.getOutputStream());

        //add file part
        output.writeBytes("-" + BOUNDARY + "\r\n");
        output.writeBytes(String.format(
                "Content-Disposition: form-data; name=\"%uploadAvatar\"; filename=\"%s\"",
                params[1], "\r\n"
        ));
        output.writeBytes("Content-Type " +
                URLConnection.guessContentTypeFromName(params[1]) +
                "\r\n");
        output.writeBytes("\r\n");
        DataInputStream dataInputStream = new
            DataInputStream(context.getContentResolver().openInputStream(uri));

```

```

        DataInputStream dataInputStream = new
            DataInputStream(context.getContentResolver().openInputStream(uri));
        byte[] buffer = new byte[4096];
        int bytesRead;
        while((bytesRead = dataInputStream.read(buffer)) != -1){
            output.write(buffer, off: 0, bytesRead);
        }
        dataInputStream.close();
        output.writeBytes("\r\n");

        // End part
        output.writeBytes("-" + BOUNDARY + "\r\n");

        output.flush();
        output.close();

        int responseCode = connection.getResponseCode();
        String responseMessage = connection.getResponseMessage();

        Log.d(tag: "UploadFileResponseCode", String.valueOf(responseCode));
        Log.d(tag: "UploadFileResponseMessage", String.valueOf(responseMessage));

    } catch (Exception e) {
        throw new RuntimeException(e);
    }
    return null;
}

```

b) Các lớp xử lý response

### *Lớp HttpResponse*

Chứa các hàm dùng để invoke sau khi gọi GET, POST, PUT, PATCH async task từ Springboot server, để sử dụng phải gọi constructor, truyền activity đang gọi.

Hàm loginResponse nhận vào responseCode và json từ Springboot Server nếu status code là 200 thì gửi request updateRequest đến socket server còn không thì hiện Toast thông báo lỗi, nếu user chưa cập nhật thông tin cá nhân sẽ vào SubRegisterFragment.

```

public void loginResponse(int responseCode, String jsonResponse) {
    Gson gson = new Gson();
    CheckBox remember = activity.findViewById(R.id.cbRememberMeLogin);
    if (responseCode == 200) {
        LoadActivity.currentAccount = gson.fromJson(jsonResponse, Account.class);
        try {
            if (remember.isChecked()) {
                Log.d("tag: "debugRemember", msg: "true");
                SharedPreferences.Editor editorCheck = LoadActivity.preferencesCheck.edit();
                editorCheck.putString("check", "true");
                editorCheck.apply();

                SharedPreferences.Editor editorAccount = LoadActivity.preferencesAccount.edit();
                editorAccount.putString("account", jsonResponse);
                editorAccount.apply();
            }
            String checkName = LoadActivity.currentAccount.getUser().getFirstName();
            Log.d("tag: "debugCheckName", checkName);
            SendTask sendTask = new SendTask();
            sendTask.execute(...params: "updateRequest", String.format("%s", gson.toJson(LoadActivity.currentAccount)));
        } catch (NullPointerException e) {
            ((AuthenticationActivity) activity).swapFragment(R.id.fragmentContainerAuthentication,
                ((AuthenticationActivity) activity).getSubRegisterFragment());
        }
    } else {
        ExceptionError exceptionError = gson.fromJson(jsonResponse, ExceptionError.class);
        Toast.makeText(((AuthenticationActivity) activity).getApplicationContext(),
            exceptionError.getMessage(), Toast.LENGTH_LONG).show();
    }
}

```

Hàm registerResponse tương tự như login response nếu status code khác 200. Nhưng không kiểm tra thông tin user mà vào thẳng SubRegisterFragment.

```

public void registerResponse(int responseCode, String jsonResponse) {
    Gson gson = new Gson();
    if (responseCode == 200) {
        LoadActivity.currentAccount = gson.fromJson(jsonResponse, Account.class);
        ((AuthenticationActivity) activity).swapFragment(R.id.fragmentContainerAuthentication,
            ((AuthenticationActivity) activity).getSubRegisterFragment());
    } else {
        ExceptionError exceptionError = gson.fromJson(jsonResponse, ExceptionError.class);
        Toast.makeText(((AuthenticationActivity) activity).getApplicationContext(), exceptionError.getMessage(),
            Toast.LENGTH_LONG).show();
    }
}

```

Hàm handleResponseSubRegister dùng để xử lý trả về từ API update user, nếu thành công thì lưu user vào biến toàn cục currentAccount của LoadActivity, sau đó gửi request updateRequest tới socket server, nếu thất bại sẽ hiện Toast báo lỗi.

```

public void handleResponseSubRegister(int responseCode, String responseBody) {
    Gson gson = new Gson();
    if (responseCode == 200) {
        String loginSuccess = LoadActivity.preferencesCheck.getString(key: "check", defaultValue: "false");
        User newUser = gson.fromJson(responseBody, User.class);
        newUser.setChatRooms(new ArrayList<>());
        LoadActivity.currentAccount.setUser(newUser);
        if (loginSuccess.equals("true")) {
            SharedPreferences.Editor editorAccount = LoadActivity.preferencesAccount.edit();
            editorAccount.putString("account", gson.toJson(LoadActivity.currentAccount));
            editorAccount.apply();
        }
        SendTask sendTask = new SendTask();
        sendTask.execute(...params: "updateRequest", String.format("%s", gson.toJson(LoadActivity.currentAccount)));
    } else {
        ExceptionError exceptionError = gson.fromJson(responseBody, ExceptionError.class);
        Toast.makeText((AuthenticationActivity) activity).getApplicationContext(), exceptionError.getMessage(),
            Toast.LENGTH_LONG).show();
    }
}

```

Hàm loadUser được thực hiện để xử lý trả về từ việc lấy tất cả người dùng, chuyển json array thành List<User> lưu vào biến toàn cục listPeople cho MainActivity để đổ ra PeopleFragment.

```

public void loadUser(int responseCode, String jsonString) {
    Gson gson = new Gson();
    try {
        JSONArray jsonArray = new JSONArray(jsonString);

        Type userListType = new TypeToken<List<User>>() {}.getType();
        List<User> rawUserList = gson.fromJson(jsonArray.toString(), userListType);
        List<User> userList = rawUserList.stream().filter(user ->
            (user.getLastName() != null
                && user.getFirstName() != null
                && !Objects.equals(user.getId(), LoadActivity.currentAccount.getUser().getId())))
            .collect(Collectors.toList());
        MainActivity.listPeople = userList;
    } catch (JSONException e) {
        throw new RuntimeException(e);
    }
}

```

Hàm joinPrivateRoom được dùng để xử lý trả về khi load phòng chat, nếu thành công sẽ gọi API lấy tin nhắn trong phòng chat.

```

public void joinPrivateRoom(int responseCode, String jsonString) {
    Gson gson = new Gson();
    if (responseCode == 200) {
        Log.d( tag: "debugJoinPrivateRoom", msg: "200");
        ((ChattingActivity) activity).setCurrentChatRoom(gson.fromJson(jsonString, ChatRoom.class));
        ((ChattingActivity) activity).setRoomAvailable(true);
        GetRequestTask getRequestTask = new GetRequestTask(new HttpResponse(activity));
        getRequestTask.execute( ...params: String.format("message/%s",
            ((ChattingActivity) activity).getCurrentChatRoom().getId()),
            "loadMessage", "ChattingFragment", "ChattingActivity");
    } else if (responseCode == 500) {
        Log.d( tag: "debugJoinPrivateRoom", msg: "500");
        ((ChattingActivity) activity).setRoomAvailable(false);
    }
    ((ChattingActivity) activity).init();
}

```

Hàm loadMessage để xử lý trả về khi load tin nhắn, chuyển json array từ Springboot server thành List<Message> và set vào List tin nhắn, Adapter của ChattingFragment.

```

public void loadMessage(int responseCode, String jsonString) throws JSONException {
    Gson gson = new Gson();
    JSONArray jsonArray = new JSONArray(jsonString);
    Type userListType = new TypeToken<List<Message>>() {}.getType();
    ((ChattingFragment) (((ChattingActivity) activity).getChattingFragment()))
        .setListMessages(gson.fromJson(jsonArray.toString(), userListType));
    ((ChattingFragment) (((ChattingActivity) activity).getChattingFragment()))
        .setAdapter(new MessageAdapter(((ChattingActivity) activity), activity.getApplicationContext(),
            ((ChattingFragment) (((ChattingActivity) activity).getChattingFragment())).getListMessages()));
    ((ChattingFragment) (((ChattingActivity) activity).getChattingFragment()))
        .getListViewModel().setAdapter(((ChattingFragment) (((ChattingActivity) activity)
            .getChattingFragment())).getAdapter());
    ((ChattingActivity) activity).init();
}

```

### Lớp SocketResponse

Chứa các hàm dùng để invoke sau khi nhận response từ socket server. Để sử dụng, gọi constructor truyền Context của Activity đang gọi.

Hàm chattingResponse được sử dụng để xử lý response nhắn tin từ server, nếu client đang trong ChatActivity và đang ở đúng cửa sổ chat với người gửi tới thì gọi hàm appendMessage() để thêm tin nhắn và List<Message> của ChattingFragment và updateUiChatRoom() để cập nhật giao diện, hai hàm trên liên quan tới giao diện nên phải đặt trongUiThread. Nếu không ở đúng cửa sổ chat với người gửi tới thì thực hiện thông báo cho người dùng.

Nếu client đang ở ngoài MainActivity thì thực hiện xử lý thời gian thực giao diện thêm phòng chat đó vào ChatRoomFragment cùng tin nhắn mới nhận được.

```

public void chattingResponse(String message) {
    Gson gson = new Gson();
    Message receivedMessage = gson.fromJson(message, Message.class);
    if (context instanceof ChattingActivity) {
        if (((ChattingActivity) context).getTargetUser().getId().equals(receivedMessage.getUser().getId())) {
            ((ChattingActivity) context).runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    ((ChattingFragment) ((ChattingActivity) context).getChattingFragment()).appendMessage(receivedMessage);
                    ((ChattingActivity) context).updateUiChatRoom(receivedMessage);
                }
            });
        } else {
            NotificationService.sendNotification(context, receivedMessage);
            ((ChattingActivity) context).updateUiChatRoom(receivedMessage);
        }
    } else if (context instanceof MainActivity) {
        NotificationService.sendNotification(context, receivedMessage);
        ((MainActivity) context).runOnUiThread(new Runnable() {
            @Override
            public void run() {
                ((ChatRoomFragment) ((MainActivity) context).getChatRoomFragment()).realTimeUiChatRoom(receivedMessage);
            }
        });
    }
}

```

Hàm createPrivateRoomResponse dùng để xử lý response khi người dùng tạo phòng chat mới hoặc người dùng khác muốn tạo phòng chat với client đang dùng. Thực hiện thêm phòng chat vào List<ChatRoom> trong biến toàn cục currentAccount của LoadActivity. Nếu người dùng đang ở ChattingActivity, set dữ liệu phòng chat từ socket server vào biến CurrentChatRoom và set cho giá trị boolean isRoomAvailable là true. Nếu người dùng đang ở giao diện Main, thêm phòng chat mới vào chatRoomAdapter.

Sau đó thực hiện gửi thông báo cho người dùng.

```

public void createPrivateRoomResponse(String chatRoom) {
    Gson gson = new Gson();
    ChatRoom room = gson.fromJson(chatRoom, ChatRoom.class);
    ChatRoomFragment.chatRoomList.add(index: 0, room);
    LoadActivity.currentAccount.getUser().setChatRooms(ChatRoomFragment.chatRoomList);

    if (context instanceof ChattingActivity) {
        ((ChattingActivity) context).setCurrentChatRoom(room);
        ((ChattingActivity) context).setRoomAvailable(true);
    } else if (context instanceof MainActivity) {
        ▲ PlalsMe
        ((MainActivity) MainActivity.mainContext).runOnUiThread(new Runnable() {
            ▲ PlalsMe
            @Override
            public void run() { ChatRoomFragment.chatRoomAdapter.notifyDataSetChanged(); }
        });
    }

    Message pushMessage = room.getLatestMessage();
    if (!pushMessage.getUser().getId().equals(LoadActivity.currentAccount.getUser().getId())) {
        pushMessage.setChatRoom(new ChatRoom(room.getId(), room.getRoomName(), room.isPrivate()));
        NotificationService.sendNotification(context, pushMessage);
    }
}

```

### c) Các Activity và Fragment

#### *LoadActivity*

Activity đầu tiên khi chạy chương trình, dùng để kiểm tra thông tin đăng nhập người dùng có lưu lại hay không, gọi phương thức bắt đồng bộ ConnectTask để kết nối tới Socket server.

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_load);
    preferencesCheck = getSharedPreferences(name: "check", MODE_PRIVATE);
    preferencesAccount = getSharedPreferences(name: "account", MODE_PRIVATE);
    currentContext = this;
    IP = Utils.getIPv4(mainActivity: this, fileName: "my_ipv4.json");
    apiUrl = String.format("http://%s:8080/api/", IP);
    IP = "34.101.218.243";
    apiUrl = String.format("http://%s:8080/api/", "34.128.104.216");
    init();
    progressBar.setVisibility(View.VISIBLE);
    ConnectTask connectTask = new ConnectTask(activity: this);
    connectTask.execute();
}

```

Hàm handleSocketResponse() dùng để invoke các hàm dựa theo tên hàm lấy từ response của socket server, các hàm này được viết trong SocketResponse.

```

1 usage  ↳ PlalsMe
public void handleSocketResponse(String jsonResponse) {
    Gson gson = new Gson();
    Response response = gson.fromJson(jsonResponse, Response.class);
    try {
        Log.d( tag: "debugCurrentContext", currentContext.toString());
        Method responseMethod = SocketResponse.class.getDeclaredMethod(response.getResponseFunction(), String.class);
        responseMethod.invoke(new SocketResponse(currentContext), response.getResponseParam());
    } catch (InvocationTargetException e) {
        throw new RuntimeException(e);
    } catch (IllegalAccessException e) {
        throw new RuntimeException(e);
    } catch (NoSuchMethodException e) {
        throw new RuntimeException(e);
    }
}

```

### *AuthenticationActivity*

Activity này dùng để thực hiện đăng ký và đăng nhập, gồm 3 fragments là LoginFragment, RegisterFragment và SubRegisterFragment, khi khởi tạo activity, nếu extra navigate là true đồng nghĩa với việc user đã lưu thông tin đăng nhập nên sẽ vào subRegisterFragment.

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_authentication);
    fragmentManager = getSupportFragmentManager();
    loginFragment = fragmentManager.findFragmentById(R.id.fragmentContainerAuthentication);
    Intent prevIntent = getIntent();
    try {
        if (prevIntent.getStringExtra( name: "navigate").equals("true")) {
            swapFragment(R.id.fragmentContainerAuthentication, subRegisterFragment);
        }
    } catch (NullPointerException e) {}
}

```

Hàm swapFragment dùng để chuyển đổi giữa LoginFragment và RegisterFragment.

```

public void swapFragment(int fragmentContainerId, Fragment fragment) {
    fragmentManager.beginTransaction()
        .replace(fragmentContainerId, fragment, tag: null)
        .setReorderingAllowed(true)
        .addToBackStack( name: "name")
        .commit();
}

```

Trong LoginFragment gồm các trường input về gmail, password để đăng nhập khi nút login được bấm vào sẽ gọi API signin từ Spring boot server.

```

btnLogin.setOnClickListener(new View.OnClickListener() {
    @PlalsMe
    @Override
    public void onClick(View v) {
        EditText editTextEmail = (EditText) view.findViewById(R.id.editTxtEmailLogin);
        EditText editTextPassword = (EditText) view.findViewById(R.id.editTxtPasswordLogin);
        String jsonString;
        try {
            jsonString = new JSONObject()
                .put("email", editTextEmail.getText())
                .put("password", editTextPassword.getText())
                .toString();
        } catch (JSONException e) {
            throw new RuntimeException(e);
        }
        PostRequestTask postRequestTask = new PostRequestTask(new HttpResponse(authenticationActivity));
        postRequestTask.execute(...params: "account/signin", jsonString, "loginResponse");
    }
});
return view;

```

Tương tự với LoginFragment, ở RegisterFragment cũng có các trường input về gmail, password để đăng ký khi nút Register được nhấn vào sẽ gọi API signup.

```

btnRegister.setOnClickListener(new View.OnClickListener() {
    @PlalsMe
    @Override
    public void onClick(View v) {
        EditText editTextEmail = (EditText) view.findViewById(R.id.editTxtEmailRegister);
        EditText editTextPassword = (EditText) view.findViewById(R.id.editTxtPasswordRegister);
        String jsonString;
        try {
            jsonString = new JSONObject()
                .put("email", editTextEmail.getText())
                .put("password", editTextPassword.getText())
                .toString();
        } catch (JSONException e) {
            throw new RuntimeException(e);
        }
        PostRequestTask postRequestTask = new PostRequestTask(new HttpResponse(authenticationActivity));
        postRequestTask.execute(...params: "account/signup", jsonString, "registerResponse");
    }
});

return view;

```

SubRegisterFragment là Fragment gồm các trường input về thông tin user để cập nhật user, và upload avatar của user.

### **MainActivity**

Activity này là giao diện chính của client, gồm 2 Fragment con là ChatRoomFragment và PeopleFragment, khi vừa vào giao diện chính, API loadUser của Springboot server sẽ được gọi để đổ vào danh sách người dùng bên people fragment.

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    NotificationChannels.createNotificationChannels(context, this);

    mainContext = this;
    LoadActivity.currentContext = this;

    GetRequestTask getRequestTask = new GetRequestTask(new HttpResponse(activity, this));
    getRequestTask.execute(...params: "user", "loadUser");
}

```

PeopleFragment khi được khởi tạo sẽ đổ danh sách user vào useAdapter để hiện lên giao diện.

```

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                        Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    View view = inflater.inflate(R.layout.fragment_people, container, attachToRoot: false);
    listViewPeople = view.findViewById(R.id.listViewPeople);
    userAdapter = new UserAdapter(mainActivity.getApplicationContext(), MainActivity.listPeople);
    listViewPeople.setAdapter(userAdapter);
}

```

ChatRoomFragment khi được khởi tạo sẽ lấy danh sách phòng chat từ biến toàn cục currentAccount của LoadActivity để đổ vào chatRoomAdapter hiện lên giao diện.

```

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                        Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    View view = inflater.inflate(R.layout.fragment_chat_room, container, attachToRoot: false);
    LoadActivity.currentContext = this.getContext();

    chatRoomList = LoadActivity.currentAccount.getUser().getChatRooms();
    Log.d(tag: "debugChatRoomList", chatRoomList.toString());
    chatRoomAdapter = new ChatRoomAdapter(this.getContext(), chatRoomList);
}

```

Hàm realTimeUiChatRoom của ChatRoomFragment sẽ xử lý realtime cho giao diện phòng chat khi có tin nhắn mới phòng chat đó sẽ tự nhảy lên đầu.

```

public void realTimeUiChatRoom(Message message) {
    int oldPosition = chatRoomAdapter.getPositionByChatRoom(message.getChatRoom());
    ChatRoom updatedChatRoom = (ChatRoom) listViewChatRoom.getItemAtPosition(oldPosition);
    updatedChatRoom.setLatestMessage(message);
    chatRoomList.remove(oldPosition);
    chatRoomList.add(index: 0, updatedChatRoom);
    LoadActivity.currentAccount.getUser().setChatRooms(chatRoomList);
    chatRoomAdapter.notifyDataSetChanged();
}

```

## *ChattingActivity*

Activity này đảm nhiệm cho chức năng nhắn tin khi khởi tạo sẽ kiểm tra nếu user nhắn từ ChatRoomFragment, lấy currentChatRoom từ extra và gọi API lấy tin nhắn, nếu user nhắn từ PeopleFragment, lấy targetUser từ extra và gọi API lấy chatroom từ 2 user id.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_chatting);

    LoadActivity.currentContext = this;

    FragmentManager fragmentManager = getSupportFragmentManager();
    chattingFragment = fragmentManager.findFragmentById(R.id.fragmentContainerChatting);

    Intent prevIntent = getIntent();
    Gson gson = new Gson();
    try {
        From chat room fragment
        currentChatRoom = gson.fromJson(prevIntent.getStringExtra("name: "currentChatRoom"), ChatRoom.class);
        Log.d("tag: "debugIntentCurrentChatRoom", currentChatRoom.toString());
        targetUser = currentChatRoom.getTargetUser();
        Log.d("tag: "debugIntentUser", targetUser.toString());
        isRoomAvailable = true;
        GetRequestTask getRequestTask = new GetRequestTask(new HttpResponse(activity: this));
        getRequestTask.execute(...params: String.format("message/%s", currentChatRoom.getId()), "loadMessage");
    } catch (NullPointerException e) {
        From people fragment
        targetUser = gson.fromJson(prevIntent.getStringExtra("name: "targetUser"), User.class);
        Log.d("tag: "debugIntent", targetUser.toString());
        GetRequestTask getRequestTask = new GetRequestTask(new HttpResponse(activity: this));
        String path = String.format("chat_room/%s/%s/true", LoadActivity.currentAccount.getUser().getId(), targetUser.getId());
        getRequestTask.execute(...params: path, "joinPrivateRoom");
    }
}
```

Hàm updateUiChatRoom trong ChattingActivity nhằm cập nhật tin nhắn mới vào dữ liệu phòng chat hiện tại vào biến toàn cục currentAccount. getUser().getChatRoomList() của LoadActivity.

```
3 usages ▾ PlalsMe
public synchronized void updateUiChatRoom(Message message) {
    List<ChatRoom> chatRoomList = LoadActivity.currentAccount.getUser().getChatRooms();
    ChatRoom currentChatRoom = chatRoomList.stream().filter(c -> c.getId().equals(message.getChatRoom().getId())).findFirst().get();
    LoadActivity.currentAccount.getUser().getChatRooms().remove(currentChatRoom);
    message.setChatRoom(null);
    User messageUser = new User(
        message.getUser().getId(),
        message.getUser().getLastName(),
        message.getUser().getFirstName(),
        message.getUser().getDob(),
        message.getUser().getAvatar(),
        message.getUser().getGender()
    );
    message.setUser(messageUser);
    currentChatRoom.setLatestMessage(message);
    chatRoomList.add(index: 0, currentChatRoom);
    LoadActivity.currentAccount.getUser().setChatRooms(chatRoomList);
}
```

ChattingFragment là Fragment con của ChattingActivity nhằm chứa các xử lý gửi nhận tin nhắn.

Hàm sendMessage() dùng để gửi tin nhắn đến Socket server dưới dạng json thông qua Async Task có tên SendTask, sau đó gọi hàm updateUiChatRoom() để cập nhật, sau đó gọi hàm appendMessage để thêm tin nhắn đó vào List<Message> và gọi hàm updateUiChatRoom để cập nhật vào biến toàn cục đồng thời đồng bộ giao diện.

```
public void sendMessage() {
    String messageJson;
    try {
        messageJson = new JSONObject()
            .put("name", "content", editTxtMessage.getText().toString())
            .put("name", "roomId", chattingActivity.getCurrentChatRoom().getId())
            .put("name", "userId", LoadActivity.currentAccount.getUser().getId())
            .toString();
    } catch (JSONException e) {
        throw new RuntimeException(e);
    }
    SendTask sendTask = new SendTask();
    sendTask.execute(...params: "chattingRequest", messageJson);
    Message newMessage = new Message(editTxtMessage.getText().toString(),
        LoadActivity.currentAccount.getUser(), chattingActivity.getCurrentChatRoom());
    appendMessage(newMessage);
    chattingActivity.updateUiChatRoom(newMessage);
}
```

#### Hàm appendMessage

```
3 usages  ↗ PlalsMe *
public void appendMessage(Message message) {
    listMessages.add(message);
    adapter.notifyDataSetChanged();
}
```

#### d) Service

##### *NotificationChannels*

Trong lớp NotificationChannels sẽ có 3 thuộc tính tĩnh gồm : CHANNEL\_GROUP\_ID (chuỗi id dùng cho kênh gom nhóm), CHANNEL\_ID\_DEFAULT (là chuỗi id dùng cho kênh mặc định), CHANNEL\_ID\_HIGH (là chuỗi id dùng cho kênh có ưu tiên cao).

```
6 usages  ↗ Hai Phan
public class NotificationChannels{
    5 usages
    public static final String CHANNEL_GROUP_ID = "your_channel_group_id";
    3 usages
    public static final String CHANNEL_ID_DEFAULT = "channel_default";
    1 usage
    public static final String CHANNEL_ID_HIGH = "channel_high";
```

Hàm `createNotificationChannels` dùng để tạo ra các kênh thông báo gồm kênh mặc định và kênh có độ ưu tiên cao và một nhóm kênh. Tiến hành tạo ra một nhóm kênh có tên **OU Messenger Group**, nhóm kênh này sẽ được dùng để gán vào group cho một channel, trong code minh họa sẽ tạo ra 2 kênh có độ ưu tiên `IMPORTANCE_DEFAULT` và `IMPORTANCE_HIGH`. Để có thể tiếp tục tạo ra các kênh khác chỉ cần gán các tham số cần thiết cho hàm khởi tạo 3 tham số của `NotificationChannel` gồm 1 chuỗi id, 1 chuỗi tên kênh và độ ưu tiên, sau cùng là gán cho kênh được tạo ra một nhóm kênh và dùng 1 instance của `NotificationManager` gọi `createNotificationChannel` để hoàn tất việc tạo kênh thông báo.

```
1 usage  ▲ Hai Phan
public static void createNotificationChannels(Context context) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        // Set any additional properties for channel 2

        NotificationManager manager = context.getSystemService(NotificationManager.class);
        manager.createNotificationChannelGroup(new NotificationChannelGroup(
            CHANNEL_GROUP_ID,
            name: "OU Messenger Group"
        ));

        NotificationChannel channel1 = new NotificationChannel(
            CHANNEL_ID_DEFAULT,
            name: "Channel Default",
            NotificationManager.IMPORTANCE_DEFAULT
        );
        channel1.setGroup(CHANNEL_GROUP_ID);
        // Set any additional properties for channel 1

        NotificationChannel channel2 = new NotificationChannel(
            CHANNEL_ID_HIGH,
            name: "Channel hHigh",
            NotificationManager.IMPORTANCE_HIGH
        );
        channel2.setGroup(CHANNEL_GROUP_ID);

        manager.createNotificationChannel(channel1);
        manager.createNotificationChannel(channel2);
    }
}
```

### ***NotificationService***

Tạo một lớp dịch vụ gửi thông báo đến cho người dùng, trong lớp đó sẽ có phương thức `sendNotification(Context context, Message message)` trong đó context sẽ được dùng để xác định ngữ cảnh hàm được gọi để tạo ra 1 đối tượng Intent chuyển từ một thông báo khi người dùng nhấn vào sẽ vào thẳng đoạn chat với người gửi tin nhắn. Khi một tin nhắn được gửi đến, hàm này sẽ được kích hoạt để thực hiện tạo thông báo và hiển thị trên thanh thông báo của người dùng.

```

3 usages  ▾ Hai Phan +1
public static void sendNotification(Context context, Message message) {
    Gson gson = new Gson();
    Intent chattingIntent = new Intent(context, ChattingActivity.class);
    Log.d( tag: "debugSetTargetUser", message.getUser().toString());
    message.getChatRoom().setTargetUser(message.getUser());
    chattingIntent.putExtra( name: "currentChatRoom", gson.toJson(message.getChatRoom()));
    TaskStackBuilder builder = TaskStackBuilder.create(context);
    builder.addNextIntentWithParentStack(chattingIntent);

    PendingIntent pendingIntent = builder.getPendingIntent( requestCode: 0,
        flags: PendingIntent.FLAG_IMMUTABLE | PendingIntent.FLAG_UPDATE_CURRENT);

```

Có thể hoàn toàn điều chỉnh chỉnh loại nhạc chuông cho thông báo gửi đến người bằng cách sửa đổi TYPE\_NOTIFICATION thành 1 loại âm báo khác. Để xây dựng một thông báo cần thiết lập các thuộc tính cơ bản gồm :

- setSmallIcon() hàm này sẽ tạo ra icon nhỏ của thông báo.
- setContentTitle() hàm này sẽ dùng để tạo tiêu đề của thông báo.
- setContentText() dùng để tạo nội dung của thông báo.
- setSound() dùng để tạo ra âm thanh của tin.
- setContentIntent() tạo intent chuyển từ thông báo vào trực tiếp đoạn chat của người gửi tin.
- setGroup() tạo nhóm gồm các tin được gom vào 1 cùng kênh nhóm.

```

// Chọn loại nhạc chuông cho thông báo
Uri defaultSoundUri = RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION);

// Xây dựng thông báo
NotificationCompat.Builder notificationBuilder =
    new NotificationCompat.Builder(context, NotificationChannels.CHANNEL_ID_DEFAULT)
        .setSmallIcon(R.drawable.chat_application_logo)
        .setContentTitle(String.format("%s %s", message.getUser().getFirstName(),
            message.getUser().getLastName()))
        .setContentText(message.getContent())
        .setAutoCancel(true)
        .setSound(defaultSoundUri)
        .setContentIntent(pendingIntent)
        .setGroup(NotificationChannels.CHANNEL_GROUP_ID);

```

Dùng Builder để tạo ra một thông báo dùng để nhóm thông báo vừa tạo ở trên vào một kênh nhóm có cùng group id, các thuộc tính dùng để cấu hình thông báo này sẽ tương tự như các thông báo bình thường khác nhưng đặc biệt sẽ có thiết lập thuộc tính trong phương thức setGroupSummary(true) để chỉ ra thông báo này sẽ gom nhóm các thông báo khác còn lại vào cùng một kênh nhóm. Sau cùng chỉ cần gọi phương thức notify() của lớp NotificationManager để hoàn tất việc gửi một thông báo đến cho người dùng hiện tại.

```
NotificationCompat.Builder summaryBuilder =
    new NotificationCompat.Builder(context, NotificationChannels.CHANNEL_ID_DEFAULT)
        .setContentTitle("OU Messenger")
        // Set content text to support devices running API level < 24.
        .setContentText("New message")
        .setSmallIcon(R.drawable.chat_application_logo)
        // Specify which group this notification belongs to.
        .setGroup(NotificationChannels.CHANNEL_GROUP_ID)
        // Set this notification as the summary for the group.
        .setGroupSummary(true);

notificationManager.notify(getNotificationId() /* ID of notification */,
    notificationBuilder.build());
notificationManager.notify( id: 0, summaryBuilder.build());
```