COP 5615 Project 4

Name: Sijie Dong UFID: 02147344

Nmae: Fengbo Zheng UFID: 46183966

1. How to run

This project consists of two parts: the server and the client.
First, choose one machine as the server end. Run sbt under the folder */FBserver*, then type the command “run”:

```
> run
```

Now, the server is working. At first, it will initialize some information used in the FaceBook api, creating some users, pages etc. Then, it will bind to the port 2559 and begin listening to this port and wait for the clients to connect to.
Second, use another machine as the client end. Run sbt under the folder /*FBclient*, the type the run command as follows:

```
> run    serverIP    numofclients    numofrequests
```

Here serverIP refers to the IP address of the server machine, numofclients stands for the number of clients assumed to run the simulator and numofrequests stands for the number of requests each client assumed to make.
For example:

```
> run    192.168.0.1    100    10
```

This command means our client machine wants to connect to the server machine whose IP address is 192.168.0.1. In this case, we will have 100 FaceBook users and each user will make 10 requests (Get, Post, Delete) to the server.

2. System Structure

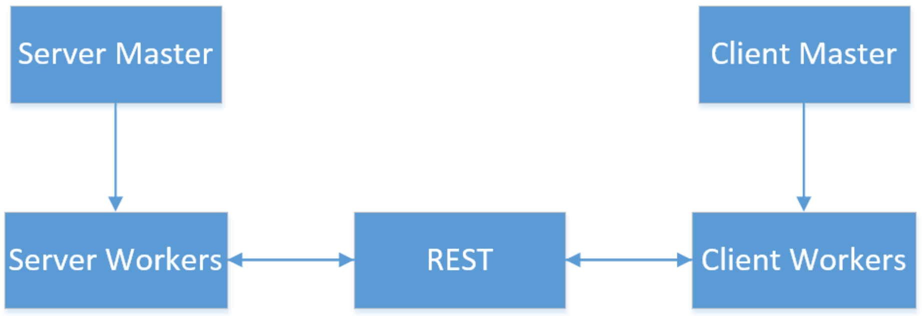
This project implements a FaceBook simulator that simulates Pages, Posts, FriendList, Profile, Album and Picture APIs. Both the server and client end uses Akka actors to simulate the FaceBook server and FaceBook users. They use messages to communicate with each other. In this project, spray is used for building the REST/HTTP-based integration layers on top of Scala and Akka. Spray-can is applied to build the HTTP layer to handle the big amount of requests from concurrent connections. More, Spray-client is used in the client part for the clients to get connection to the REST layer. Between the server and client, REST router which uses spray-routing receives and analyses messages from clients in HTTP json format and sends these messages to the server end. Also, this router receives messages from the server and send back to the clients.

The server part is consisted of one server master actor and hundreds of server worker actors. Server master is established when the server system starts. This master will initiate some information for the whole server system. In our assumption, we first create 1000 users, 100 pages as users already used the Facebook and we have already know their info. Then we generate 2000 posts, 2000 friend lists, 2000 albums and 10000 photos as already exists in the server system. These information will store in the HashMaps. We assign these posts, friend lists and albums randomly to the users and pages we created and assign the photos to the albums. After the initialization has done, we use the spray system to listen to the server's IP address on a giving port to wait for the client requests.

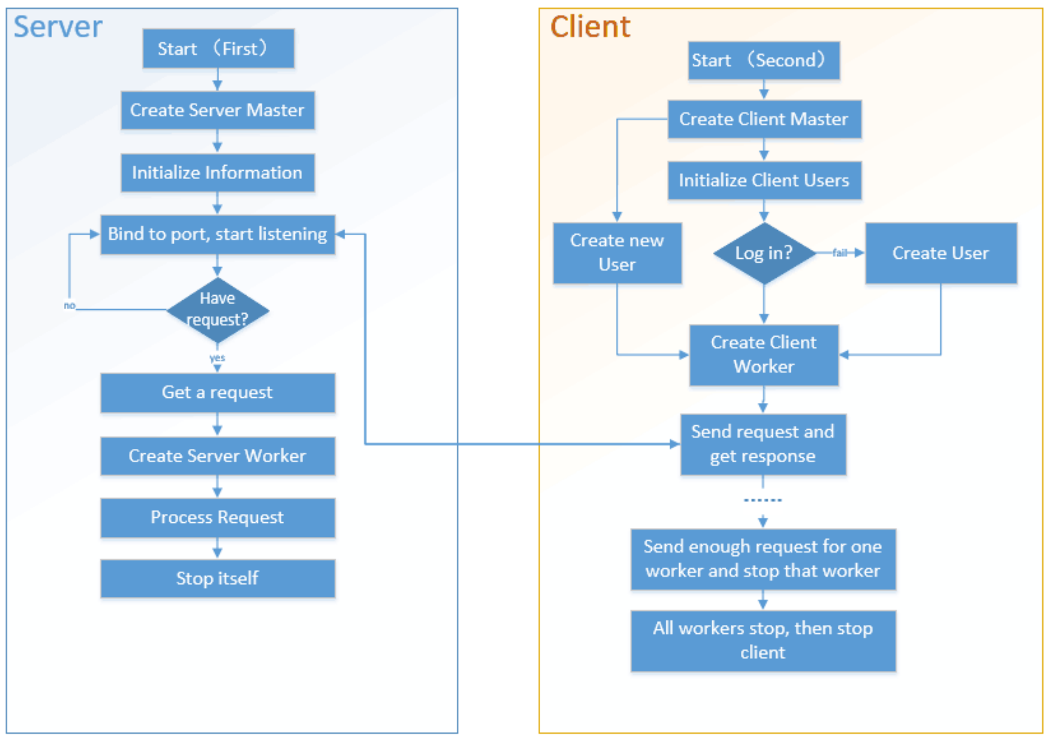
When a client request coms the REST layer first read it and get the parameters from the json form. Then it will generate a new server worker actor and send request message with the parameters to this actor. The server worker will take actions to the request (get,post or delete; page, post, friendlist…).

The client part is consisted of one client master actor and thousands of client worker actors. When the client system starts, a master actor will be created it will initial some global information for the clients. Then it will create client workers which number is inputted when running the client. Each client worker can be viewed as a Facebook user. It will send API request to sever in a schedule manner. There are three types of requests: Get, Post and Delete. We will choose request message randomly among each of the types. These requests are forward in pipelines that connect to the spray-can based REST layer. Also, the results will be sent back through the pipelines. When it has finished the number of requests required, it will stop itself and send message to the master actor. When the master actor has received terminate messages from all the workers, it will stop the whole client system.

The structure for the simulator is like follows:



The working flow for this project is like follows:



The client master will create two kinds of client workers: one is the kind of users that have already exist in the server memory. These users will first make a login request, if the server finds their information, it will return the information. If not, a new user is created in the server. Another kind of users the client master will create is users we assume as new users for the FaceBook. The server will create new users when they make the first request.

3. APIs

a. User(Profile)

User API is one of the main APIs in the FaceBook system. Here we have a Users class to store the user API elements. Users fields include id, username, firstname, lastname such basic information. Also, there is a HashMap for posts, a HashMap for friendlist and a HashMap for album that owned by the user as edges. Each user's information is stored in the memory in a HashMap where key is the user id and the value is the user structure. At the beginning, the server master will create 1000 users as already existing users, and some posts, friendlists will be assigned to the users randomly.

In user API, we have three kinds of request types, Get, Post and Delete. The Get request call to get the information of the particular user with a giving userid as a parameter. This kind of request is something in the Profile API part as it will get the profile of the aiming user. Post request creates a new user instance in the server system. The new user will be assigned a new id and recorded in the memory. Delete request deletes an existing user with the giving userid. The user will be emitted from the memory permanently.

In the client side, newly created client workers will first make a create user request to the server which means they are viewed as new users for the FaceBook. Get request and Delete request will be chosen randomly in according type set with requests in other API parts. When a Get request is chosen and send, it will visit a randomly chosen user's information through REST using a Get method. When a Delete request is required, the client will send the message to the REST and the server worker will perform the delete function. At the same time, the client worker will stop itself immediately no matter the request status. The request path is shown as follows:

Get: http://192.168.0.1:2559/profile?id=userid
Post: http://192.168.0.1:2559/user
Delete: http://192.168.0.1:2559/user?id=userid

b. Page

Page API is similar to User API. Fields in Page structure includes id, name, link, postsum. Also, there is an edge information store posts in HashMap. Each page's information is stored in the memory in a HashMap where key is the page id and the value is the page structure. At the beginning, the server master will create 100 pages as already existing pages, and some posts will be assigned to the pages randomly.

In page API, we have three kinds of request types, Get, Post and Delete. The Get request call to get the information of the particular page with a giving pageid as a parameter. Post request creates a new page instance in the server system. The new page will be assigned a new id and recorded in the memory. Delete request deletes an existing page with the giving pageid.

In the client side, Get request, Post request and Delete request will be chosen randomly in according type set with requests in other API parts. When a Get request is chosen and send, it will visit a randomly chosen page's information through REST using a Get method. When a Post request is called, the client worker sends the message and the server creates a new page. When a Delete request is required, the client will send the message to the REST and the server worker will perform the delete function. At the same time, the client worker will stop itself immediately no matter the request status. The request path is shown as follows:

Get: http://192.168.0.1:2559/page?id=pageid
Post: http://192.168.0.1:2559/page
Delete: http://192.168.0.1:2559/page?id=pageid

c. Post

Fields in Post structure includes id, message, creator, creatortype, createtime. Creator refers to the user or page that create this post. Creatortype can only be "user" or "page". Each post's information is stored in the memory in a HashMap where key is the post id and the value is the post structure. At the beginning, the server master will create 2000 posts as already existing post, and these post will be assigned to users and pages randomly.

In post API, we have three kinds of request types, Get, Post and Delete. The Get request call to get the information of the particular post with a giving postid as a parameter. Post request creates a new post instance in the server system. The new post will be assigned a new id and recorded in the memory. User or page who creates the new post will have its id and type written into the post information as creator and creatortype. Delete request deletes an existing post with the giving pageid.

In the client side, Get request, Post request and Delete request will be chosen randomly in according type set with requests in other API parts. When a Get request is chosen and send, it will visit a randomly chosen post's information through REST using a Get method. When a Post request is called, the client worker sends the message and the server creates a new post with the client id. Whether the client is considered to be a user or a page is chosen randomly. When a Delete request is required, the client will send the message to the REST and the server worker will perform the delete function. It not only deletes the post in the memory but also delete the information saved in related user and page instance. The request path is shown as follows:

Get: http://192.168.0.1:2559/post?id=postid
Post: http://192.168.0.1:2559/post?id=userid or
http://192.168.0.1:2559/post?id=pageid
Delete: http://192.168.0.1:2559/post?id=postid&userid=userid or
http://192.168.0.1:2559/post?id=postid&pageid=pageid

d. Friendlist

Fields in Friendlist structure includes id, name, owner. Edges in Friendlist structure includes member. Owner refers to the user own this post. A randomly chosen users will be assigned to the member in a HashMap. Each friendlist's information is stored in the memory in a HashMap where key is the friendlist id and the value is the friendlist structure. At the beginning, the server master will create 2000 friendlists as already existing friendlists, and these friendlist will be assigned to users and some users will be added into member randomly.

In friendlist API, we have three kinds of request types, Get, Post and Delete. The Get request call to get the information of the particular friendlist with a giving friendlist id as a parameter. Post request creates a new friendlist instance in the server system. The new friendlist will be assigned a new id and recorded in the memory. User who creates the new friendlist will have its id written into the friendlist information as owner. Delete request deletes an existing post with the giving friendlist id.

In the client side, Get request, Post request and Delete request will be chosen randomly in according type set with requests in other API parts. When a Get request is chosen and send, it will visit a randomly chosen friendlist's information through REST using a Get method. When a Post request is called, the client worker sends the message and the server creates a new friendlist with the client id. When a Delete request is required, the client will send the message to the REST and the server

worker will perform the delete function. It not only deletes the friendlist in the memory but also delete the information saved in related user instance. The request path is shown as follows:
Get: http://192.168.0.1:2559/ friendlist?id= friendlistid
Post: http://192.168.0.1:2559/ friendlist?id=userid
Delete: http://192.168.0.1:2559/ friendlist?id= friendlistid&userid=userid

e. Album

Fields in Album structure includes id, name, from, createtime. Edges in Album structure includes photos. From refers to the user who created this album. Photos will be assigned to the edge photos in a HashMap when they are created. Each album's information is stored in the memory in a HashMap where key is the album id and the value is the album structure. At the beginning, the server master will create 2000 albums as already existing albums and some photos will be assigned to the albums randomly.

In album API, we have three kinds of request types, Get, Post and Delete. The Get request call to get the information of the particular album with a giving album id as a parameter. Post request creates a new album instance in the server system. The new album will be assigned a new id and recorded in the memory. Users who create the new album will have its id written into the album information as from. Delete request deletes an existing post with the giving album id.

In the client side, Get request, Post request and Delete request will be chosen randomly in according type set with requests in other API parts. When a Get request is chosen and send, it will visit a randomly chosen album's information through REST using a Get method. When a Post request is called, the client worker sends the message and the server creates a new album with the client id. When a Delete request is required, the client will send the message to the REST and the server worker will perform the delete function. The request path is shown as follows:
Get: http://192.168.0.1:2559/ album?id= albumid
Post: http://192.168.0.1:2559/ album?id=userid
Delete: http://192.168.0.1:2559/ album?id= albumid

f. Photo

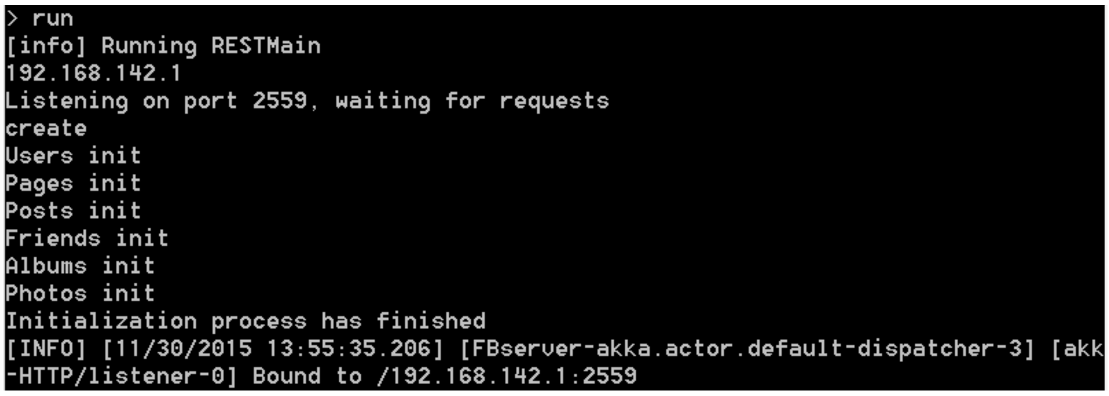
Fields in Photo structure includes id, name, from, album, picture. From refers to the user own this post. Album refers to the album it belongs to. Each photo's information is stored in the memory in a HashMap where key is the photo id and the value is the photo structure. At the beginning, the server master will create 10000 photos as already existing photos, and these photos will be assigned to albums randomly.

In photo API, we have three kinds of request types, Get, Post and Delete. The Get request call to get the information of the particular photo with a giving photo id as a parameter. Post request creates a new photo instance in the server system. The new photo will be assigned a new id and recorded in the memory. User who creates the new photo will have its id written into the photo information as from and the album id will be assigned to album. Delete request deletes an existing post with the giving photo id.

In the client side, Get request, Post request and Delete request will be chosen randomly in according type set with requests in other API parts. When a Get request is chosen and send, it will visit a randomly chosen photo's information through REST using a Get method. When a Post request is called, the client worker sends the message and the server creates a new photo with the client id. When a Delete request is required, the client will send the message to the REST and the server worker will perform the delete function. It not only deletes the photo in the memory but also delete the information saved in related album instance. The request path is shown as follows:
Get: http://192.168.0.1:2559/ album/ photo?id= photoid
Post: http://192.168.0.1:2559/ photo?userid=userid&albumid=albumid
Delete: http://192.168.0.1:2559/ photo?id= photoid& albumid=albumid

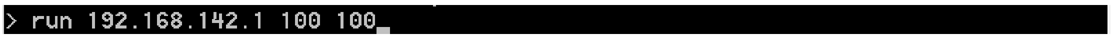
4. Running results and Analysis

First we run the server machine. After an initialization process done, the REST actor bind to and listen to the port, waiting for requests.



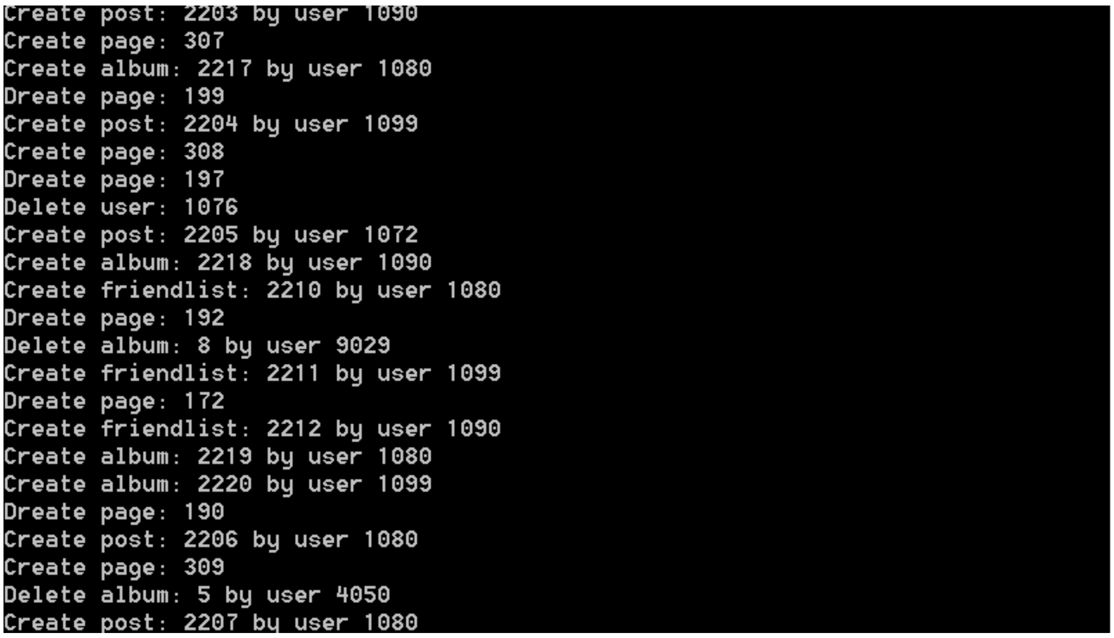
```
> run
[info] Running RESTMain
192.168.142.1
Listening on port 2559, waiting for requests
create
Users init
Pages init
Posts init
Friends init
Albums init
Photos init
Initialization process has finished
[INFO] [11/30/2015 13:55:35.206] [FBserver-akka.actor.default-dispatcher-3] [akk
-HTTP/listener-0] Bound to /192.168.142.1:2559
```

In this case, we can find that the IP address of the server is 192.168.142.1 and the port is 2559.
Then, we start our client machine. The client machine is connected to the server machine and it will have 100 existing users and 100 new users to be created. Each user will make 100 requests to the server.



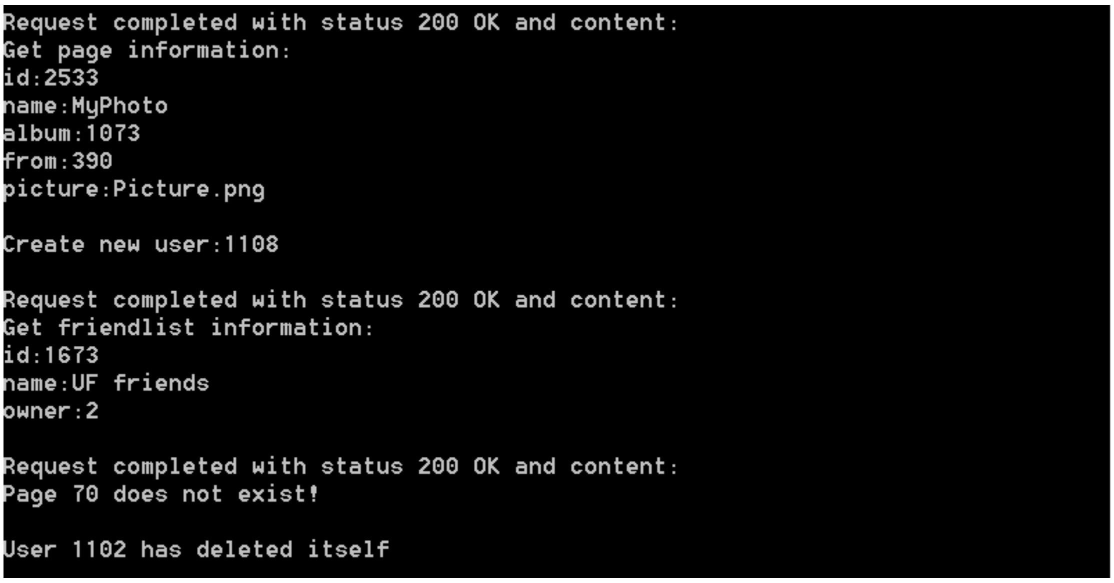
```
> run 192.168.142.1 100 100
```

We can see the result on the server part:



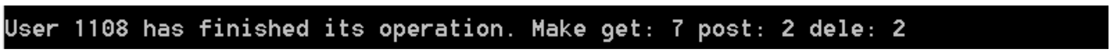
```
Create post: 2203 by user 1090
Create page: 307
Create album: 2217 by user 1080
Dreate page: 199
Create post: 2204 by user 1099
Create page: 308
Dreate page: 197
Delete user: 1076
Create post: 2205 by user 1072
Create album: 2218 by user 1090
Create friendlist: 2210 by user 1080
Dreate page: 192
Delete album: 8 by user 9029
Create friendlist: 2211 by user 1099
Dreate page: 172
Create friendlist: 2212 by user 1090
Create album: 2219 by user 1080
Create album: 2220 by user 1099
Dreate page: 190
Create post: 2206 by user 1080
Create page: 309
Delete album: 5 by user 4050
Create post: 2207 by user 1080
```

Here, we find that on the server end, some new pages, posts, photos etc. are created or deleted by some users.
One the client part, we find information like that:



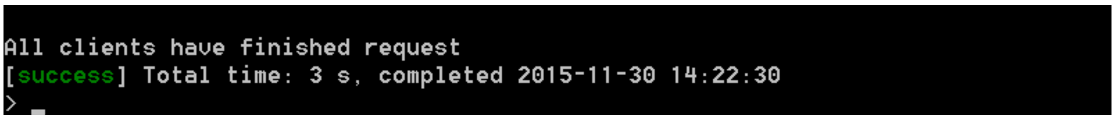
```
Request completed with status 200 OK and content:
Get page information:
id:2533
name:MyPhoto
album:1073
from:390
picture:Picture.png

Create new user:1108

Request completed with status 200 OK and content:
Get friendlist information:
id:1673
name:UF friends
owner:2

Request completed with status 200 OK and content:
Page 70 does not exist!

User 1102 has deleted itself
```

Some users visit the pages, posts, profiles etc. and the information is returned from the server and print under the client workers. Some new users are created and some users stop using FaceBook and require the server delete itself.
When a user finishes his requests, we can see the client worker show this information like follows:



```
User 1108 has finished its operation. Make get: 7 post: 2 dele: 2
```

This information shows the user who stops and the number of gets, posts and deletes during working.
When all users finish their request, we get information as follows:



```
All clients have finished request
[success] Total time: 3 s, completed 2015-11-30 14:22:30
>
```