Project Design

Designing Aspects of the Project

Team Number:

12

Team Member:

Boya,Lai(boyal@andrew.cmu.edu), Yu Zheng (yuzheng@andrew.cmu.edu), Yi Qiu (yiq@andrew.cmu.edu)

Project Name:

CookBootCamp

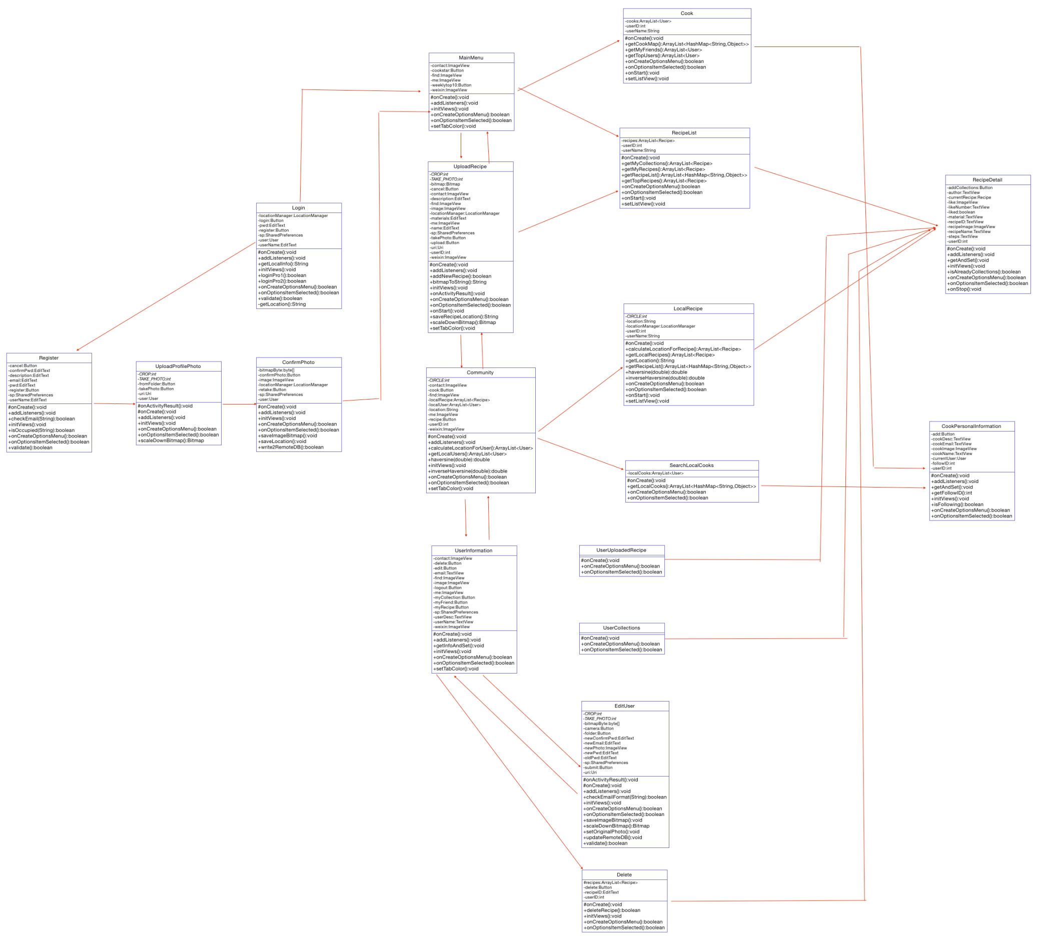
1: Design Screens

Our application supports **two resolutions: 540 * 960 pixels** and **1200 * 1920** (Google Nexus 7 with 7 inches' screen). And both of these two screens can perform well in either portrait or landscape orientation without distortion. No hard code here and all these resources are identified by their ids in java code. Also as what is recommended in project handout, all .XML files are named after "PackageName_ActivityName.xml.

What should be mentioned that since there are two types of layout, android studio simply distinct these two layout by an identifier in a bracket.

2: Design Presentation Tier

As what we have shown in our earlier Wireframe and pageflow, in this unit all activities can be reached by **Intent.** And each activity has its java file in **ui** package and has two layouts in /res/layout folder corresponding two different screen sizes. User can register, login, edit his/her information and several recipe manipulations. Most of these buttons in layout can be clicked and implement functionality except for some that needs later work in this project with web service.



3. Designing Content Provider

In this app, the context provider just include such elements:

1. Store user information
2. Update user information
3. Read user information
4. Store recipe information
5. Read recipe information
6. Store relationship between one user and another user
7. Store relationship between one user and a recipe

In order to implement these functions, we design a database and a server to help handle all requests from Android device, according to the request, we would send the corresponding massage for Android device, and our app client would deal with these massage and present to user.

First, we decide use **MySQL database** to store all information, and would run on a server. Also, we designed a server to interact with database, as a bridge between android client and database.

For **Database** part, the below show what we designed for the database:

In our app, we decide use 4 tables in our database: User, Follow, Recipe, Collection.

1. User Table

Name	**Type**	**Additional Information**
userid	INT	Auto_increment primary_key
username	VARCHAR(30)	Not null
password	VARCHAR(30)	Not null
email	TEXT	Email address
description	TEXT	Introduce yourself
location	TEXT	Coordinate
image	TEXT	Uri

2. Follow Table

Name	**Type**	**Additional Information**
userid	INT	
followid	INT	

3. Recipe Table

Name	**Type**	**Additional Information**
recipeid	INT	Auto_increment primary_key
authorid	INT	
name	VARCHAR(30)	
materials	TEXT	
description	TEXT	
location	TEXT	Coordinate
uploadTime	TEXT	Upload time
countlike	INT	

Image	TEXT	

The materials column would contain many materials, they would be split by a comma.

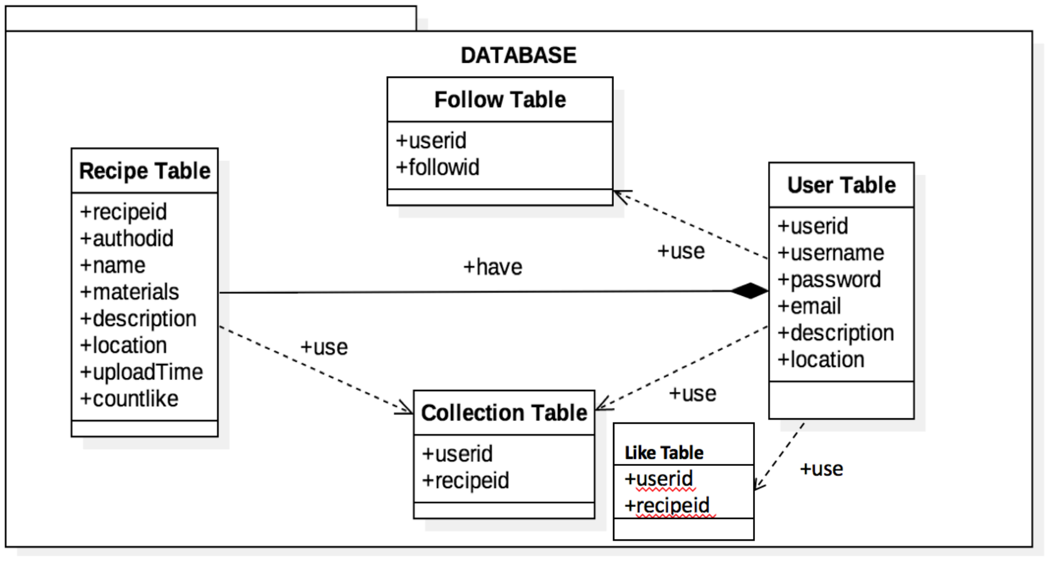
4. Collection Table

Name	**Type**	**Additional Information**
userid	INT	
recipeid	INT	

5. Like Table

Name	**Type**	**Additional Information**
userid	INT	
recipeid	INT	

Here is the UML diagram for our database schema:



According to above table, we can list all action, which user would interact with database (**CURD**):

Create:

1. User would create a new record into User table when they register a new account.
2. User would create a new record into Recipe table when they upload a new recipe.
3. User would create a new record into Follow Table when they follow a user.
4. User would create a new record into Collection Table when they add a recipe into their collection.

Read

1. User would read one record from User table when they want to review their own information.
2. User would read one record limited from User table when they want to check other user information.
3. User would read records from Follow Table when they want to review all users they have followed.
4. User would read records from Recipe Table when they want to review all recipes they have collected.

Update

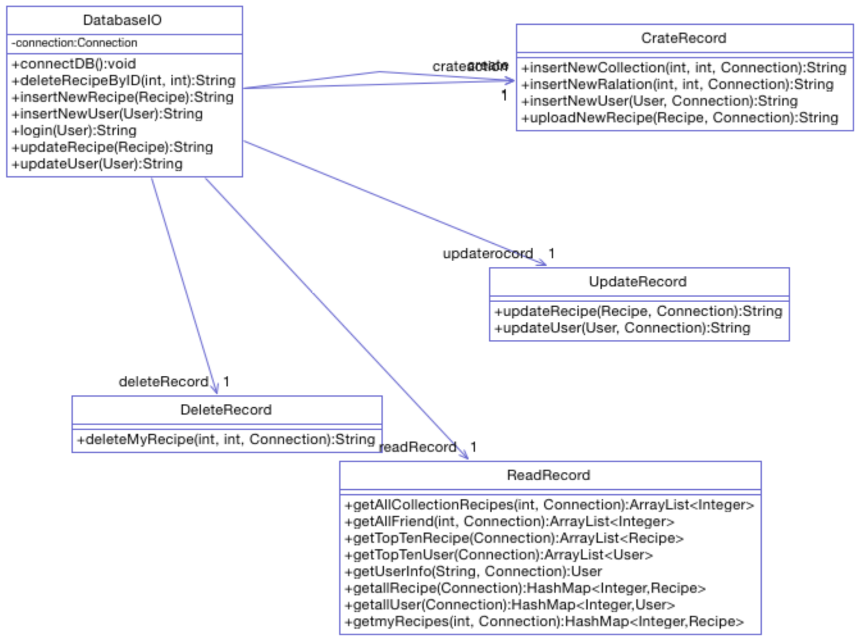
1. User can update a record in User table when they want to update their information such as password, email address and other information but user name and user ID.
2. User can update a record in Recipe table when they want to update some information about a recipe such as materials and so on.

Delete

1. User can delete a record in Follow Table when they don't want to follow the user anymore.
2. User can delete a record in Collection Table when they want to cancel collect the recipe anymore.

After we designed above **CRUD,** we create interface about the **CRUD** functions.

These user actions about **CURD** we implement it on dblayout package, crate 4 interface class: **CrateRecord.java, ReadRecord.java, DeleteRecord.java** and **UpdateRecord.java** file. They all provide correspond function about **CURD** for developer, and we can implement them in relevant class. Here is the class diagram about our implement of **CRUD:**

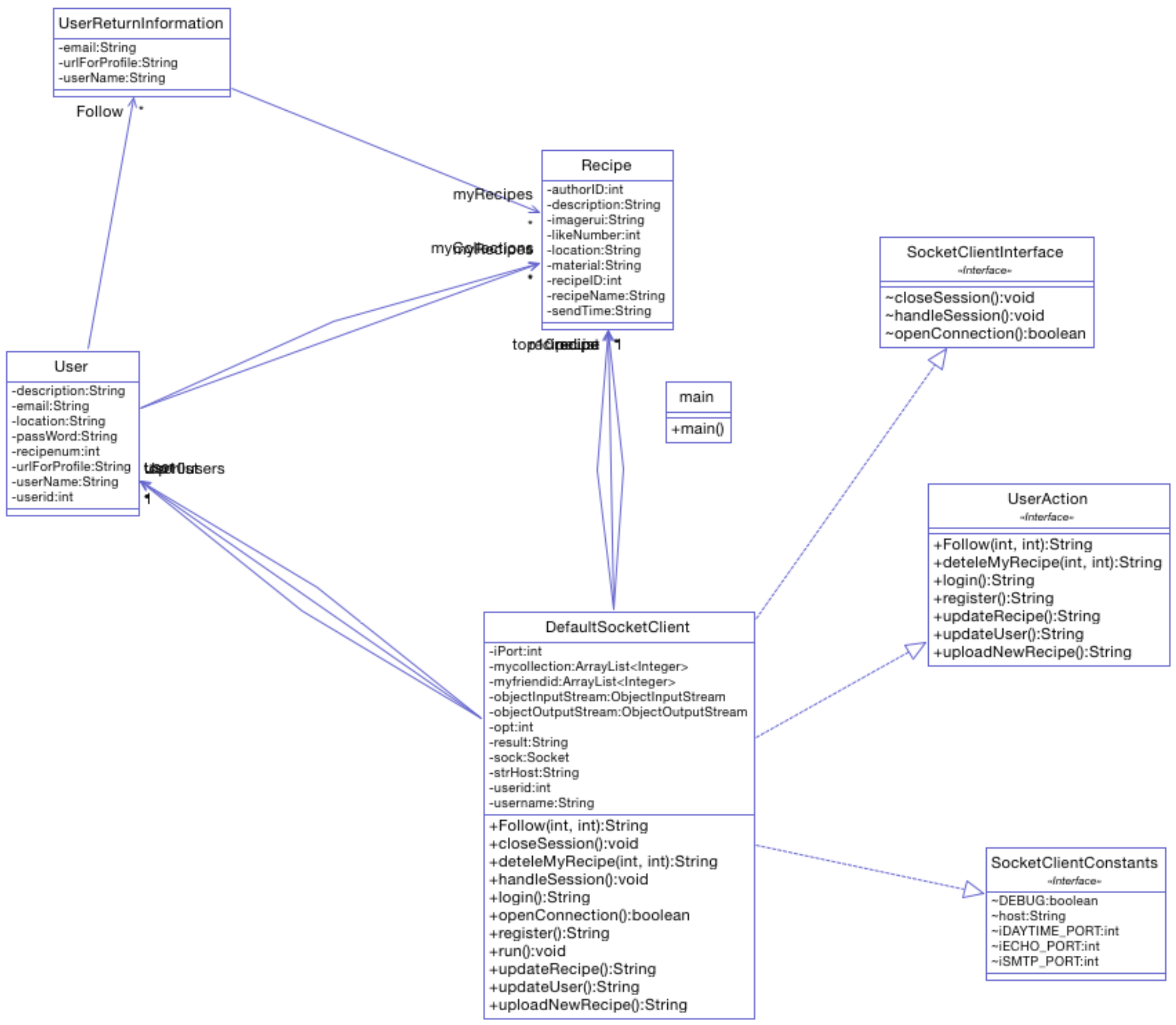

DatabaseIO
-connection:Connection
+connectDB():void
+deleteRecipeByID(int, int):String
+insertNewRecipe(Recipe):String
+insertNewUser(User):String
+login(User):String
+updateRecipe(Recipe):String
+updateUser(User):String
crateaction
1
CrateRecord
+insertNewCollection(int, int, Connection):String
+insertNewRalation(int, int, Connection):String
+insertNewUser(User, Connection):String
+uploadNewRecipe(Recipe, Connection):String
updaterocord 1
UpdateRecord
+updateRecipe(Recipe, Connection):String
+updateUser(User, Connection):String
deleteRecord 1
DeleteRecord
+deleteMyRecipe(int, int, Connection):String
readRecord 1
ReadRecord
+getAllCollectionRecipes(int, Connection):ArrayList<Integer>
+getAllFriend(int, Connection):ArrayList<Integer>
+getTopTenRecipe(Connection):ArrayList<Recipe>
+getTopTenUser(Connection):ArrayList<User>
+getUserInfo(String, Connection):User
+getallRecipe(Connection):HashMap<Integer,Recipe>
+getallUser(Connection):HashMap<Integer,User>
+getmyRecipes(int, Connection):HashMap<Integer,Recipe>

As we can see that, the DatabaseIO class is the main class to handle all logic action about database. And each action has their own class file, this 4 class include all action from user.

Also, after we finish design the database, we have to implement remote server to handle the massage from Android device. For this part, we decide to use Server Socket and multithread method to implement this logic business.

The server socket includes three main methods: open connection, close connection and handle session.

Handle session is the key method we have to implement. For this method, server socket would receive all request messages from client, which is Android device, and then it would interact with database (crate, delete, update or read) according to the message. After interact with database, the handle session would send back corresponding result to user client (Android device), so the user client can present the message as user want. To implement the server, we encapsulate all function, and we would only create one server running on a PC, but it can deal with multiple thread, which means it can handle many client, when one client wants to interact with server, it would create a new thread, after finish the interaction, the server would kill this thread. Below is the Class Diagram of our server routine.



4. Designing Application Tier

For **remote** package, we use client socket to interact with remote server, which we have done at **Design Content Provide part**. In client side (Android device), when user input some or click button and anything else, once it has to get some information from server, it will use socket client build up a new pipeline with remote server, and then recipe result from remote server, and send these result information to other logic tier to present them to user. Here is the basic Class Diagram for our client socket:

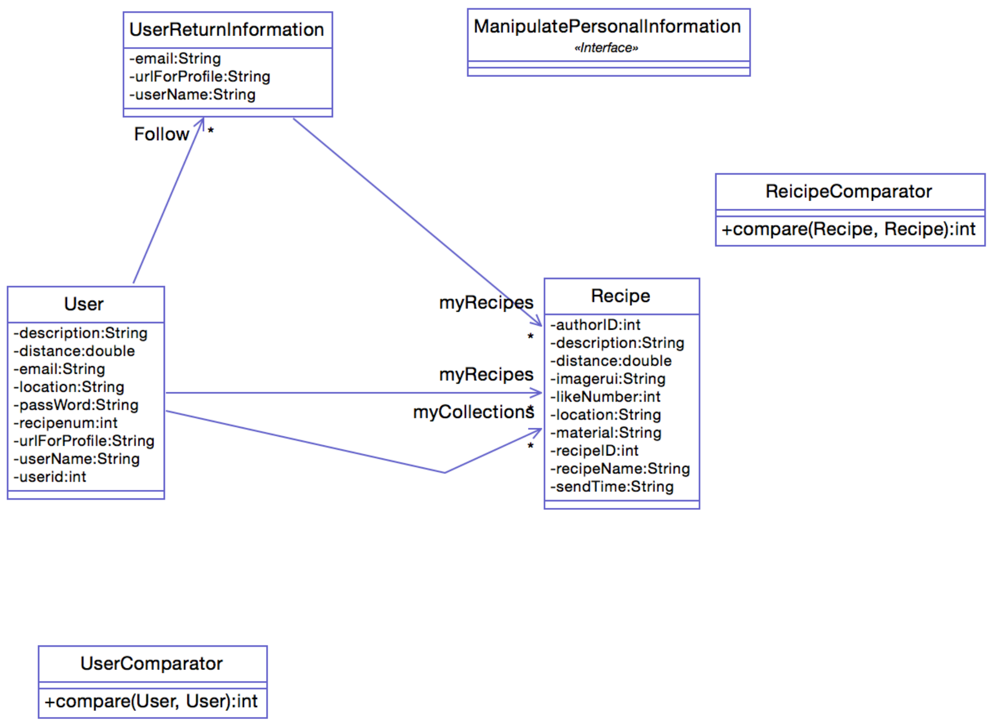
As we can see, when user want to interact with remote socket, it would create a new instance called UserClient, and the UserClient would invoke all methods from DefaultSocketClient.class, DefaultSocketClient.class implement all methods which belong to interface, UserAction.java.

When User finish interact with remote server, the socket would be kill, and next time, when user want to interact with server, a new socket client will be build up and interact with server.

For **Model** part, we create a class Recipe to represent all the information a recipe should contain. Each Recipe has a unique recipe ID, its own name, the ID of its author, description about how to cook, the number of people who like it, the time the recipe was sent to the server as well as an inner class called Material, which represents all materials needed by cooking. Basically, this Material Class contains an ArrayList of material names. It can add, delete, modify its material list. The external class, Recipe, also provides method for setting and getting its ID, Name, author ID, description, like number and send time.

To represent the user, we create a parent class Person， which implements the ManipulatePersonalInformation interface. Each person has his/her own location, email and profile, which can be get and modified by methods in the ManipulatePersonalInformation interface. The User class extends from the Person class, with new properties like: username, password, user's own collections, user's own created Recipes and User's followers. Those properties can also be added, deleted and modified. Exception for manipulating these properties, the User class can also search local recipe and local users and return its own message to other users.

When a user returns its information to another user, it can't and shouldn't return everything (like password). Instead, it will return an object of class UserReturnInformation. Class UserReturnInformation represents the reasonable information that one user returns to another. It contains the username, email address, profile and recipes for the user.



5: Exception handling

Exception handling is a very important function in Apps. An app can never be accepted widely if it gives bad user experience. Mainly in our team project, main exception comes from user input. Several cases should be considered now and maybe some more in later work:

1. In register page, if user select a user name and unfortunately found occupied in database, an exception occurs to remind user input another username;
2. In register page, if user input no username, or no password, or the previous password doesn't match the later one, an exception occurs to remind user check his/her input;
3. In the upload recipe page, if user input no name, or descriptions, or no steps for this recipe, an exception occurs to remind user check his/her input;
4. In edit user information page, if user input password is not the same as the old one, an exception occurs to remind user check his/her password again.

As mentioned above, all these exceptions come from user input and this should be handled properly in case of app crash.