COMP-421 Project Deliverable 3 - Group 7

Github repository for the app:
https://github.com/iwantadventureinthegreatwidesomewhere/Icebreak

Attached is also a runnable JAR file for the application (Icebreak.jar)

Q1:

Function “birthdayMessage” (for complete function see separate attached SQL file) will send a birthday message to all users with birthdays on the same month and day as the input, with the default values being the current_date.

Using it with a query examining all the messages of users with a given birthdate we can see that they all receive the birthday message.

Querying messages of users with birthdays on January 8th
Before:

	msgid	status	timestamp	content	userid	chatid	conversation_number
1	83	seen	2018-09-22	its a date	28	20	<null>
2	85	sent	2018-09-22	lets meet tomorrow	28	20	<null>
3	310	seen	2017-12-10	lmao	228	81	<null>
4	311	sent	2017-12-10	lmao	228	81	<null>
5	312	pending	2017-12-10	How is it going?	228	81	<null>
6	313	seen	2017-12-10	that was enjoyable	228	81	<null>

After:

	msgid	status	timestamp	content	userid	chatid	conversation_number
1	83	seen	2018-09-22	its a date	28	20	<null>
2	85	sent	2018-09-22	lets meet tomorrow	28	20	<null>
3	310	seen	2017-12-10	lmao	228	81	<null>
4	311	sent	2017-12-10	lmao	228	81	<null>
5	312	pending	2017-12-10	How is it going?	228	81	<null>
6	313	seen	2017-12-10	that was enjoyable	228	81	<null>
7	1373	pending	2020-04-13	Happy Birthday!	28	20	<null>
8	1374	pending	2020-04-13	Happy Birthday!	228	81	<null>

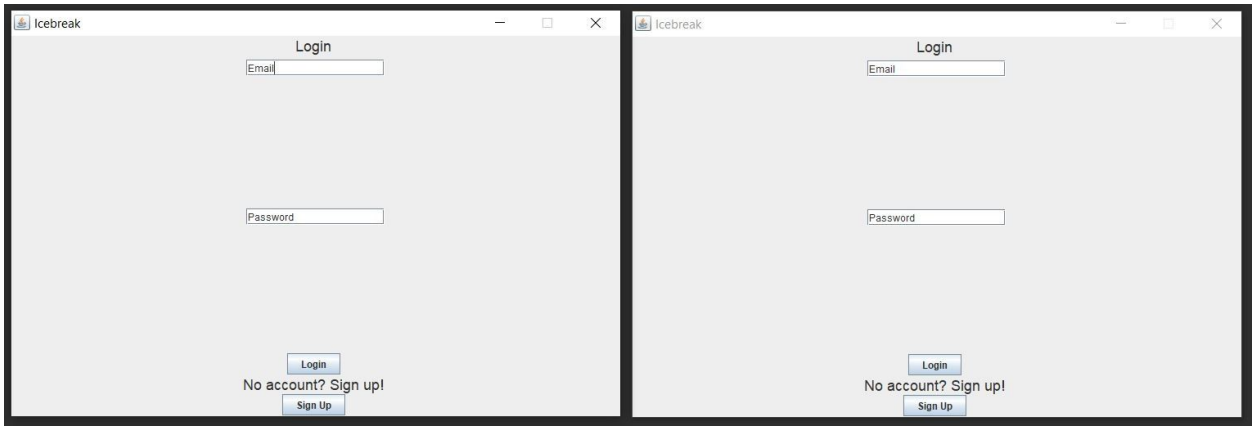
As shown in the two pictures above, after the birthdayMessage was called, each user with a birthday on the specified day received a birthday message from the server.

Q2: Demo (shown with two separate instances of the app for two separate users that will match)

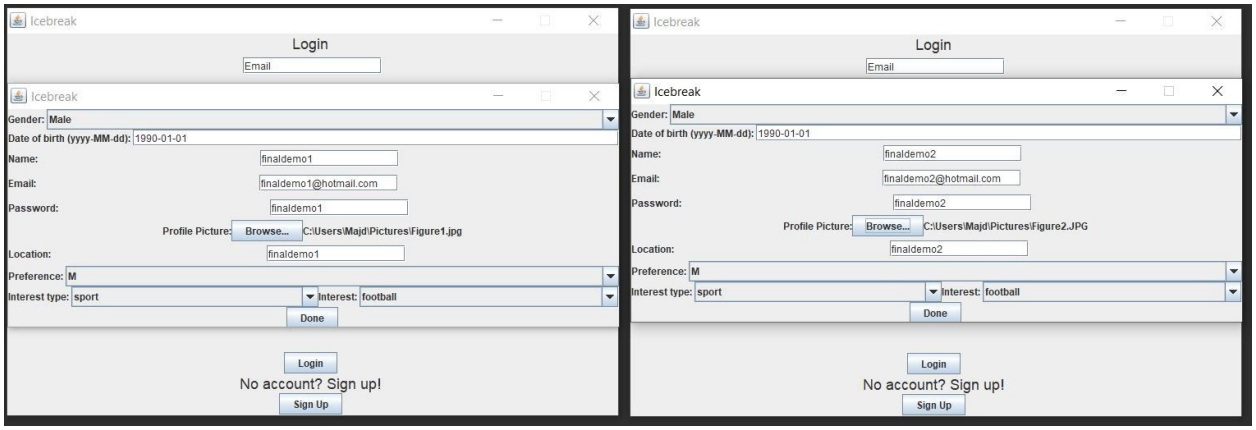
The operations that will be demoed (each has their own method in the App class of our project):

- Logging into a user account (login() method)
- Signing up a new user (signup() method)
- Matchmaking (match() method)
- Loading the chat (refreshChat() method)
- Sending a message in an existing chat (sendMessage() method)

When opening the app, the LoginFrame is displayed, where the user can either log in to their existing account with their email and password, or sign up. We will start by signing up two new user accounts, which will navigate to the SignupFrame.



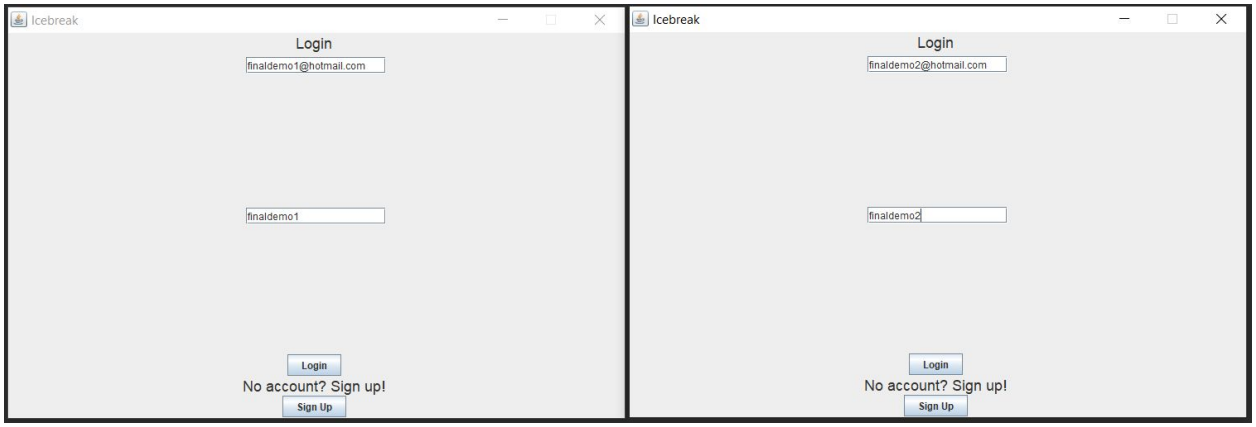
We enter the user account details for the two new users, ensuring that their dating preference and interest are the same so they can be matched later in the demo. The signup() method is called to insert the data into the database.



Below are the database records for the two newly created user accounts in the Users table.

```
cs421=> select * FROM users WHERE name LIKE 'finaldemo%';
 userid | gender | birth_date |    name    |         email          | password  | date_time_created | profile_photo | is_active | is_matchmaking |  location
--------+--------+------------+------------+------------------------+-----------+-------------------+---------------+-----------+----------------+------------
    524 | M      | 1990-01-01 | finaldemo2 | finaldemo2@hotmail.com | finaldemo2 | 2020-04-12        |             0 | t         | f              | finaldemo2
    525 | M      | 1990-01-01 | finaldemo1 | finaldemo1@hotmail.com | finaldemo1 | 2020-04-12        |             0 | t         | f              | finaldemo1
(2 rows)
```

Usually after signing up, the MainFrame will be displayed, where the new user can begin matchmaking, but to demonstrate the ability to also log in an existing user, we backed out and started the app again for both users. Now they will execute the login() with their user credentials and, if successful, will be directed to the MainFrame of the app.



On the MainFrame of the app, users can either attempt to find a new match, or see all recent chats with previous matches (starts with no chats as no matches have been made for these two new users).

The way matchmaking works on our app is by calling the match() method which will check if there is already another user searching for a match with the same dating preference and interest, and if not, will wait for 30 seconds for someone to match to

them. If a match is made, then a ChatFrame is displayed so that the users can begin exchanging messages. If no match is found, then a message will be displayed to the users. In this demo, user finaldemo1 clicks the match button, finds no one to match to, and so waits for someone else to match to them, and user finaldemo2 will then also click on the match button (within the 30 second wait window of user finaldemo1), and immediately find a match with user finaldemo1. Both apps therefore proceed to the ChatFrame. You can also see that now finaldemo2 is in finaldemo1's recent chat on the MainFrame, and that finaldemo1 is in finaldemo2's recent chat as well. Clicking on the recent chat will execute the refreshChat() method which loads the chat details, such as the name of the user you're exchanging messages with, and the current message history of the chat (ordered). refreshChat() is also used in the ChatFrame() periodically to load the latest messages in the chat.

Below are the database records in the Participates table showing that, after finaldemo1 (userid = 525) and finaldemo2 (userid = 524) are matched and a new chat is created for them, finaldemo1 and finaldemo2 are participating in the same chat and can now exchange messages.

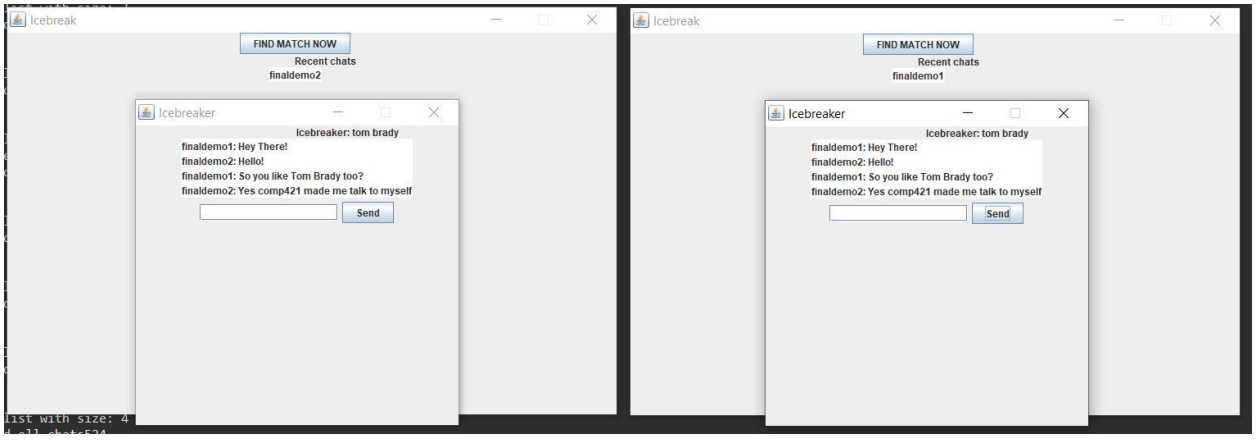
```
cs421=> SELECT * FROM participates WHERE userid = 524 OR userid = 525;
 userid | chatid
--------+--------
    524 |    395
    525 |    395
(2 rows)
```

Further, the icebreaker topic for their chat is created and can be seen below in the Icebreakers table (chatid matches that of the chat that finaldemo1 and finaldemo2 are participating in).

```
cs421=> SELECT * FROM icebreakers WHERE chatid = 395;
 conversation_number | chatid |  subject  | time_duration
---------------------+--------+-----------+---------------
                2176 |    395 | tom brady |            60
(1 row)
```

In the ChatFrame, the icebreaker topic related to the common interest that the two matched users share is displayed at the top. Users type messages in the textfield and when they click the send button to send the message, the sendMessage() method is called to insert the message into the database. In the demo, you can see some of the

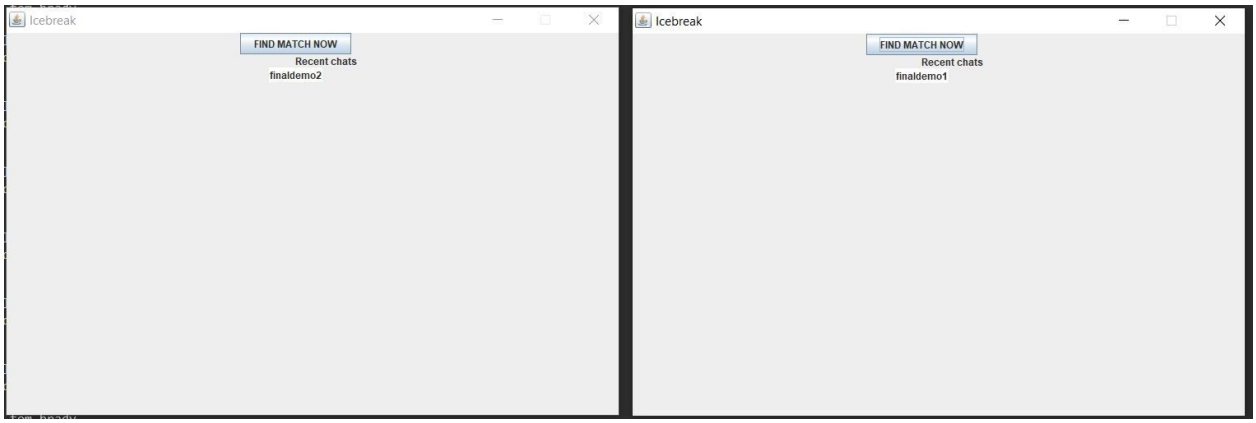
messages that finaldemo1 and finaldemo2 have exchanged related to their love for football (and apparently Tom Brady too).



Below are the database records in the Message table showing the messages sent by either finaldemo1 or finaldemo2 in their chat.

```
cs421=> SELECT * FROM messages WHERE chatid = 395;
 msgid | status | timestamp  |              content               | userid | chatid | conversation_number
-------+--------+------------+------------------------------------+--------+--------+---------------------
  1369 | sent   | 2020-04-12 | Hey There!                         |    525 |    395 |                2176
  1370 | sent   | 2020-04-12 | Hello!                             |    524 |    395 |                2176
  1371 | sent   | 2020-04-12 | So you like Tom Brady too?         |    525 |    395 |                2176
  1372 | sent   | 2020-04-12 | Yes comp421 made me talk to myself |    524 |    395 |                2176
(4 rows)
```

Both users can close the chat, and as previously mentioned, the chat can be accessed again via the recent chats section of the MainFrame, which loads the chat again via the refreshChat() method.



Q3:

-- chatid is used by the match(), refreshChat(), sendMessage() methods, some of which perform multiple queries involving chatid (e.g. inserting a new message).
CREATE INDEX index_msg_chatid ON Messages(chatid);
Since chatid **IS NOT** a primary key in the Messages table, and is instead a foreign key, we can create an index for it. We chose chatid as searching for relevant records was costly, and most importantly, was executed frequently, and so an index would help to reduce the computational cost. For example, chatid is used to acquire all the messages in a chat, ordered by when they were sent, to keep the chat history in the UI as up-to-date as possible.

-- is_matchmaking is used by the match() method whenever users are seeking to find a match on the app
CREATE INDEX index_is_matchmaking ON Users(is_matchmaking);
Given that the app is based on the idea of creating as many matches as possible, it is therefore vital to swiftly find all users that are matchmaking via the is_matchmaking boolean attribute. When finding matches, if a match cannot be initially found, then users wait for a short time for someone else to match with them. During this time, the is_matchmaking flag is switched to true for 30 seconds (if no match is made within the allotted time, the match fails). Finding current matchmaking users with their is_matchmaking flag set to true must be fast in order for the match to occur within the 30-second waiting time, before it gets switched back to false. This is especially important if the Users table is massive due to their being many registered user accounts.

we make use of an interval of 30 seconds after clicking the find match button. During this time, the is_matchmaking boolean is temporarily switched to true. If the system does not find a match for the user at the beginning of the 30 seconds, the user stays available to receive chats from any new match that started searching after them.

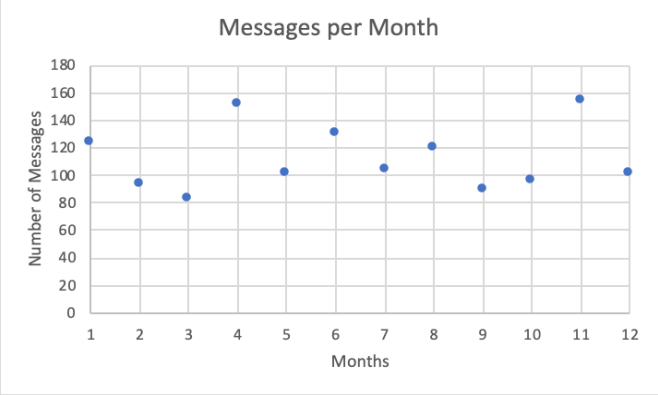
Q4:

A useful data visualization would be to see which months users are historically messaging (and likely matchmaking) the most. This could be a great way to choose, for

example, which months have more expensive advertising prices on an application. Here is the SQL query used to provide us the data:

```
SELECT EXTRACT(month FROM (timestamp)), count(*)
FROM Messages
GROUP BY EXTRACT(month FROM (timestamp))
ORDER BY EXTRACT(month FROM (timestamp));
```

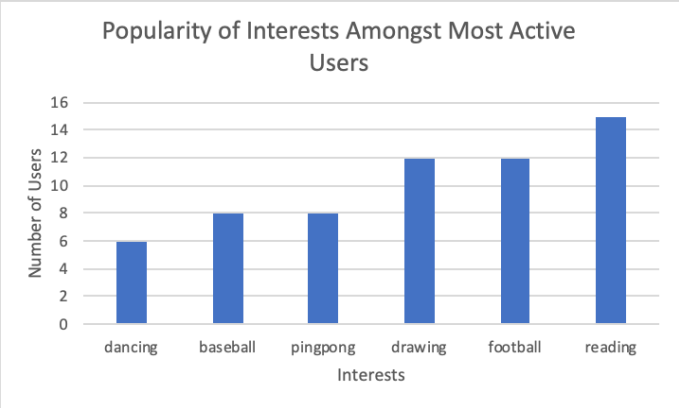
After plotting the resulting output in Excel, here is the data visualized:



Another useful data visualization is to see the popularity of certain interests amongst the most active (e.g. 5 or more messages sent) users on the application. This would provide a better understanding of the application's demographic and things they like to talk about on the app. Here is the SQL query used to fetch us the data:

```
SELECT l.type, count(*)
FROM Likes l, (SELECT u.userid
   FROM Users u, Messages m
   WHERE u.userid = m.userid
   GROUP BY u.userid
   HAVING count(*) > 5) mostActive
WHERE l.userid = mostActive.userid
GROUP BY l.type;
```

After plotting the resulting output in Excel, here is the data visualized:



Q5:

We developed a simple graphical user interface for our application.