OLoml - Final Design Doc

(O Love of my life)

System Description

Core Vision

We're building a "Tinder" (dating app) for people searching for CS partners. A pooling algorithm will give each student in the class a ranked list of classmates for them to "swipe" left or right on and a matching algorithm will use a system generated compatibility score and the results from the users decisions to determine optimal pairings. The professor will have control over the time periods during which students can update their profiles and swipe.

Features

- **Algorithmically generating a pool of potential partners**
- **Using data about students and their preferences to generate their matches**
- **Viewing other people's profiles**
- **Editing your own profile**
- **"Swiping" (choosing) on potential partners**

Narrative Description

Finding a CS partner can be an enigma, but not with OLoml! We want to create a system that allows people who may not have a large social circle within the CS community to choose project partners. We want the factors that determine partners to be based on practical things like experience level and schedule matching so that groups are as successful as possible. We hope to implement extensions that give users even more data about potential partners and make algorithms more accurate.

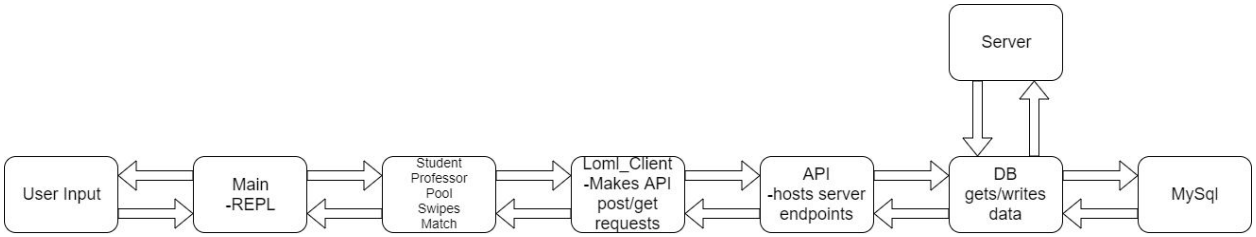
Changes

There have been no major changes to our idea since what we submitted as part of MS1. However, our modules were not as well fleshed out in MS1; in particular, we did not decide to modularize the networking and database components at the time. The code submitted for MS2 contains separate modules for server-side networking, client-side networking, and the database, as well as helper modules to help parse input from the user and store global data.

System Design

We plan on implementing a system capable of handling four different types of time periods. At each time period the student and admin will have different functions. During the NULL period students cannot do anything but a professor can set period dates and import students.

During the update period, students are allowed to update their profiles before swipe period starts. During the swipe period, students are allowed to swipe right/left on other students based on their profiles. During the match period, students receive their matched partner.



<u>User Input:</u>
The user input is a CLI that takes string inputs from the user.

<u>Main:</u>
Main runs a REPL loop which accesses functions in Student, Professor, Pool, Swipes, and Match. A user first inputs their netid which runs through two different branches of processes if the netid is an admin or a student.

If the user is a admin then we check what type of period we are currently in. If the period is not set yet, then the admin must set the periods and must import a JSON file of the students. If the period is update period, the admin can remove a student or reset the class. If the period is a swipe period, the only option admin has is to reset the class. If the period is a match period, the admin can match the students or reset the class.

If the user is a student then we check what type of period we are currently in. If the period is update period, the student can update different fields of their profile. If the period is a swipe period, the student can swipe right or left on individuals as the REPL displays the candidates profile. If the period is a match period and the professor has run the matching function, the student sees their partner.

<u>Command</u>
Command is a helper module that helps handles much of the user input parsing. It is used extensively in main to enable the repl to take and process raw input from a user.

<u>Student, Professor, Pool, Swipes, Match</u>
These five modules contain relevant functions connecting the REPL to Loml_client to retrieve post/get request data.

Student defines the relevant data that a student holds and the functions necessary to create, update, and get information from the client regarding a student. Only a certain portion of a

user's data can be changed and this is represented by the fields schedule, courses, hours, location, and profile bio.

Professor defines the relevant functions necessary for a professor/admin to manage the class. Professor can get students, delete students, set period, and import students.

Pool represents the relevant modules necessary for a student to swipe left/right on a pool of individuals. The pool is a data structure representing a small subset of the class to be swiped on.

Swipes represents a data structure with functions necessary to swipe right/left on an individual. Swipes takes left/right swipe actions and transforms them into results which also include a "compatibility" score between individuals. Swipes is also able to write these swipes into the database.

Match represents the functions necessary to match a matrix of swipes. From the swipe results, match finds the best pairings of individuals based on their compatibility score and how the individuals swiped on each other. If there are an even number of individuals in the class, everyone is matched. If there are an odd number of individuals, one individual will be left unmatched which the professor will handle at his/her own discretion. The matching algorithm is inspired by a relaxation of the transportation linear programming problem.

<u>Loml_client</u>
Loml_client represents the client side networking interface and is the only way a client can interact with the API/database. The files listed above make calls to Loml_Client in order to get or post data. Loml_client then takes the call and formats the data and makes an api request. Loml_client uses cohttp to enable its request interface and structure its request format. Loml_client contains a request function for every endpoint in the API.

<u>Oclient</u>
We have abstracted many of async components in cohttp into a separate wrapper client called oclient.ml. He plans on releasing this library similar to how ohttp was released last year by a 3110 student/TA. The functions contained inside oclient.ml enable GET, POST, and DELETE requests with intuitive syntax and no need to understand any async concepts. They are used heavily in loml_client.ml as part of the request functions.

<u>Api</u>
Api contains the callbacks for all the endpoints in our RESTful API, which is hosted on the coecis server. GET requests can be made to the API to retrieve MySQL data and POST/DELETE requests can be made to modify data in the database. The API utilizes the request/response format specified in the ohttp library. This part of the code is not exposed to the client. The documentation is for api is modeled directly after the documentation of some popular APIs such as the Facebook and Twitter APIs.

Server

The server module simply integrates all of the callbacks defined in api.ml and runs a server on port 8000 with those endpoints. Again, the server/api interface relies primarily on the ohttp library. As mentioned, the api/server interface is hosted on the coecis server and is not exposed to the client.

Backend_lib

Backend_lib is simply an environment/globals file containing values necessary for the backed code that may or may not change in the future (i.e. endpoint names, base API url). The reason for this module is that it is much easier to change the url in this one file than everywhere it occurs in the code.

DB

DB handles all of the data retrieval and posting to the database. DB makes MySQL queries and executes them. It utilizes the ocaml-mysql library to interface directly with the database. Like the API, the database is hosted on our coecis server and is not directly exposed to the clients.

Data

Table: Students

Netid*	Name*	Year*	Schedule	Courses	Hours	Profile	Location
str	str	str	str	str	str	str	str

```
mysql> describe students;
+----------+---------------+------+-----+---------+-------+
| Field    | Type          | Null | Key | Default | Extra |
+----------+---------------+------+-----+---------+-------+
| netid    | varchar(7)    | YES  |     | NULL    |       |
| name     | varchar(50)   | YES  |     | NULL    |       |
| year     | varchar(10)   | YES  |     | NULL    |       |
| schedule | varchar(1500) | YES  |     | NULL    |       |
| courses  | varchar(1500) | YES  |     | NULL    |       |
| hours    | int(11)       | YES  |     | NULL    |       |
| profile  | varchar(500)  | YES  |     | NULL    |       |
| location | varchar(40)   | YES  |     | NULL    |       |
+----------+---------------+------+-----+---------+-------+
```

Asterisked fields are immutable once the class has been created

- **Here the Netid and Name are both strings**
- **Year is represented by string: “freshman”, “sophomore”, “junior” or “senior”**
- **Schedule is stored in the database as a string version of a boolean list where there is a true/false stored for each of morning, afternoon, and evening on each day of the week**

- **Courses are stored in the database as a string list of all the CS courses that a student has taken**
- **Hours is stored as a string and represents the number of of hours a student is willing to dedicate to this class every week**
- **Profile is a string that the student can update with any additional information about themself**
- **Location is a string that represents where a student lives/wants to meet**

Table: Matches

stu1	stu2
Netid (str)	Netid (str)

```
mysql> describe matches;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| stu1  | varchar(12) | YES  |     | NULL    |       |
| stu2  | varchar(12) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
```

- **Stu1 and stu2 are both strings that contain the netid's of the student and his or her match**

Table: Credentials

Netid	Password
str	str

```
mysql> describe credentials;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| netid    | varchar(7)  | YES  |     | NULL    |       |
| password | varchar(50) | YES  |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
```

- **Credentials contains the netid (string) and password(string) for each student in the class.**
- **The professor netid is stored as "admin"**

Table: Periods

Update changed to u	Swipe changed to s	Match changed to m
float	float	float

- **Periods contains the float representation of the epoch date for the start date of the update period (u), swipe period (s) and match period (m)**

```
mysql> describe periods;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| u     | varchar(50) | YES  |     | NULL    |       |
| s     | varchar(50) | YES  |     | NULL    |       |
| m     | varchar(50) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
```

Table: Swipes

Netid	Swipes
str	Str

```
mysql> describe swipes;
+--------+--------------+------+-----+---------+-------+
| Field  | Type         | Null | Key | Default | Extra |
+--------+--------------+------+-----+---------+-------+
| netid  | varchar(7)   | YES  |     | NULL    |       |
| swipes | varchar(500) | YES  |     | NULL    |       |
+--------+--------------+------+-----+---------+-------+
```

- **Swipes contains the netid of the student (stored as a string) and the student's swipe results (stored as a string representation of a json object)**

We are storing data in a MYSQL database that will be running on a remote server. Our OCaml code will make HTTP requests to the server storing the database to make reads and writes. The general database schema is shown above.

<u>External Dependencies</u>

Having the MySQL database stored on a remote server allows different students to swipe concurrently. Our OCaml code never reads/writes the database directly; rather, it makes HTTP requests to send data to be written to the database or retrieve data from the database. In order to accomplish this, we created a RESTful client/API interface which relies primarily on ohttp (included as httpServer.ml) for the backend and cohttp (which we've refactored into oclient.ml as mentioned above) for the client side. To use the Mysql database we use the ocaml-mysql library and write the SQL queries inline. Originally, we planned on using the pgocaml library and camlp4 syntax to make queries to the database but ultimately, after extensively pursuing this option, we were unable to resolve connectivity issues and found that a lack of up to date documentation on this library made the code hard debug. Switching

over to the ocaml-mysql library proved much more successful in connecting to the database and querying the relevant data. Lastly, we use YoJSON to help structure data that flowed between the API and the client; although the API took and returned string bodies, we required that we only used strings that were able to be converted to jsons using Yojson.Basic.Util.from_string to allow us to work with a more uniform datatype.

Testing Plan

In general, our testing strategy for correctness was geared more towards user and interactive testing. The reason for this was that our code base relies primarily on components that change in real time (server state, database state) and have lots of side effects. Rather than writing unit tests that we would constantly have to rewrite, we felt it was better to test interactively using utop, the MySQL command line interface, and curl/wget/Postman (a desktop app for making HTTP requests). With respect to the networking and the database specifically, we also made an effort to test those modules in isolation, ensuring that we had a functional server-side architecture. As for the repl, we felt it best to test our application internally as well as put it in the hands of the users who we intended it for, students.

Server/Database Isolated Testing

In order to test the database in isolation, we first installed all the database dependencies and tested directly in the MySQL command line. We made sure the tables had been created properly and were storing all the right types. To do so, we ran simple CRUD (create-read-update-delete) operations in the CLI using the MySQL standard syntax. Once we knew that the database was set up correctly, we ran the functions in db.ml interactively, asserting that the results of a write showed up in the CLI and the results of the retrievals reflected what we had written previously. We also tried operations such as table truncates to ensure some of our expected functionality (such as reset class) would be feasible.

As for the networking, we first tested the api side architecture with dummy endpoints to make sure the API was accessible. These dummy endpoints can be found in the api as test_get and test_post. Test_get simply returns "Hello world!" and test_post takes a string response body and returns "Hello " ^ response_body. To do so, we used a combination of curl/wget/postman. Once we had the server working locally, we tried testing it from other computers on the same network. Here, we found that we couldn't make requests from computers other than the one hosting the server, even if we used the correct IP address. We quickly realized that we needed to disable the Ubuntu default firewall, a fairly trivial fix that we were luckily able to catch early on. Once we had these test endpoints running, we tested the client side code (specifically the oclient abstractions) in utop by making requests to the test_get and test_post endpoints with designated clients. Those test functions can be found in loml_client under the same name.

Once we knew we had a working network and database, we worked to integrate them as part of the larger application and follow the testing procedures mentioned in the next two sections.

User Testing

After we completed an initial version of our product (which including testing the database and network in isolation and writing initial submission instructions for how to compile and run the project), we created a "pseudo-class" of ~6 friends and asked them to follow our instructions to obtain partners. We asked for feedback in terms of:

- **Obvious bugs**
- **Ease of use**
- **Whether or not they would use this product in real life (and why)**
- **Areas of expansion**

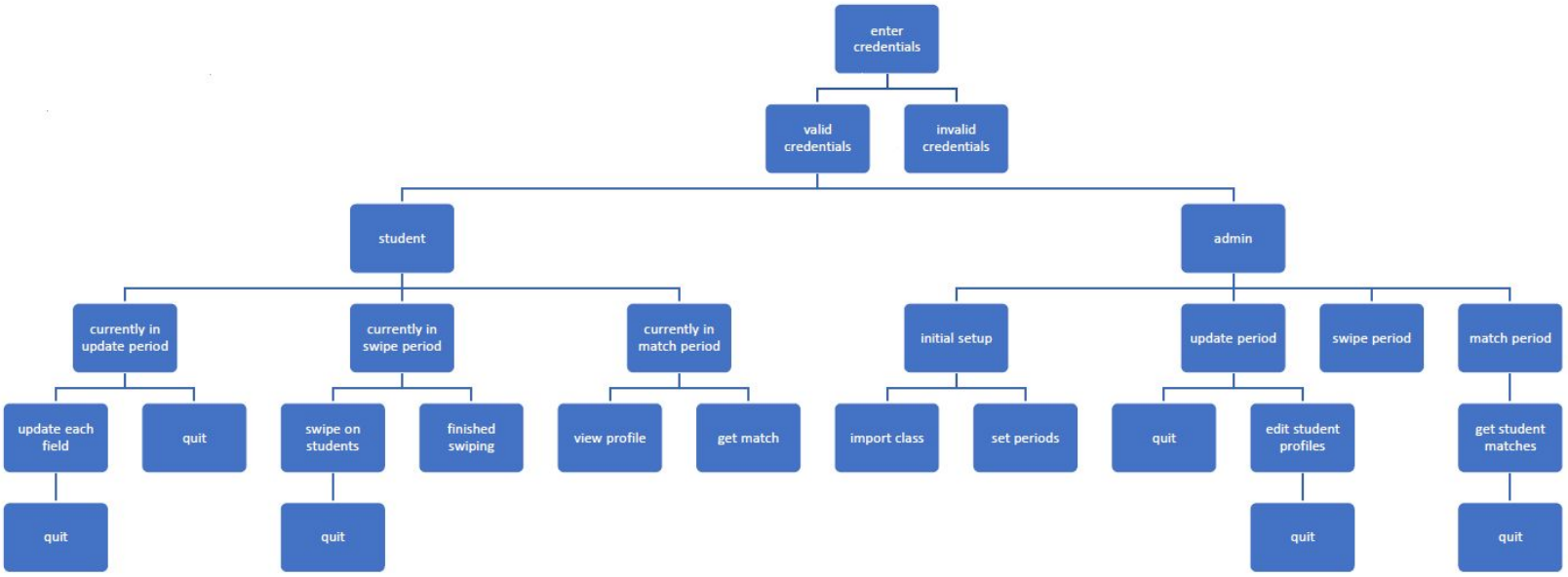
Here we found that the main feedback we were getting was in regards to the bulky and unintuitive user interface. Here are some major points and how we addressed them:

- **Students were complaining that it took too long to enter information about their schedule. Previously the system was set up such that students were given 21 different times (7 days a week, morning, afternoon, evening for each) and say whether they were free or not for all of them. Students requested that we make it so that they just had to enter the times they were free, and that is what it is right now.**
    - **After this change, students complained that the format of the expected input was unintuitive - we required students to indicate availability on Sunday morning as "sunday,morning". Students felt it was more intuitive to have a space instead of a comma, so we changed that too.**
- **Students also complained that some aspects of the repl asked them to indicate a choice using a number rather than a word, i.e. enter 1 to update your location, enter 2 for collegetown. They requested that it be changed to simply typing in the word location or collegetown, which we adapted across the system.**
- **Students complained that after updating their system, they were taken back to the main menu rather than back to the update menu. They had to reload their profile over and over again; to change this, we simply redirected them back to the update menu after the loop.**
- **Students also found several ungraceful exits (usually in edge case inputs), all of which we have handled. Some examples of this are:**
    - **When the HTTP requests failed**
    - **When incorrect dates were provided for the periods**
    - **When invalid location data was provided (i.e. an unsupported location).**

Aside from these bugs, people generally liked the product and said they would use it to help find partners. The areas for expansion they suggested generally followed directly from the ideas we had in the earlier milestones, such as super likes, supporting multiple classes and supporting pairings with more than 2 students.

Interactive Testing

As mentioned earlier we felt unit testing infeasible due the each module's reliance on the database and the data stored within it and instead decided instead to interactively test the code using a model that would allow us to step through all possible paths of the program.



Use link for larger view of tree (also attached as an appendix to the end of the document) https://drive.google.com/file/d/1v3gIRxykQfJgvHQ2zJ_rPEsjHwa3iU8u/view?usp=sharing)

Using this step model of the REPL we were able to test the following at each stage of the program and for each role of the user (student versus administrator):

1. **Check that the main produces the desired output after a command (i.e student should not be allowed to quit while updating a specific field of their profile)**
    a. **Check that the main does not give access to any commands that are outside of the scope for the current period as we define it**
2. **Check that the database is updated as expected for the command entered using a combination of the MySQL command line and the retrieval functions we wrote.**

3. **Check that quitting the program produces the expected output and has the expected side effect on the database (especially at critical stages that are marked on the model)**

Additionally, we followed an incremental approach. Although we developed all the features concurrently, we decided to incrementally integrate them into our repl and test them; we started with the admin path, then the student update path, then added the swipe logic, and then finally added the matching logic. Using this model we were able to extensively find and document bugs in our program and implementation logic without having to search through which stage of the program the error occurred. Some examples of bugs we found and fixed were:

- **There was a stack overflow whenever we called the server. We found that a few of our database functions had infinite loops and were evaluating as soon as we started the server because they were not defined as functions, leading to the stack overflow.**
- **The database kept resetting every time we reset the server. At first we thought we had installed MySQL incorrectly. However, after looking at the MySQL logs in the CLI we found that we were running a truncate command every time the server started. Upon further inspection, we found that the reset_class function was not actually defined as a function but rather simply a unit and thus was evaluated every time we called the server (similar to the first mistake).**
- **Everytime we reset the class, we had to add the admin credentials again; we modified the reset_class function to remove everything except the admin credentials.**
- **We weren't able to write the epoch times in the database correctly and soon realized this was a parsing issue with how we were handling floats.**
- **The repl was always stuck in a pre-class state which we had implemented in the case where the professor did not set the period dates yet. We knew that the error was either in the database or in the API, where we validated the result from the database. We eventually found that we were handling the dates wrong in the database and thus returning the periods incorrectly.**
- **Retrieving all students returned a `Not_found message from the Cohttp_server. This was very perplexing since our API callbacks only returned `OK, `Unauthorized or `No_response statuses. We realized that the endpoint to retrieve all students, student_get, had not been registered with the server.**
- **For writing swipe results, we found that all our writes were failing with a `No_response status. We found that in our backend header verification logic, we were checking for the wrong headers on swipe_post requests and thus the code assumed the request data was wrong.**
- **The match results were not being generated properly; we found that the swipe and match modules were using different JSON schemas, which led to inconsistencies in how things were being parsed.**

Even though our modules were difficult to test alone (outside of on dummy data in utop) we found a way to verify the performance of each standalone module, and the interaction between modules, at each stage of the user experience. This gave us a high level of

confidence that our code was performing as expected (especially after we were able to catch and resolve these bugs) and allowed us to focus our user testing on the user interface rather than the correctness of our code. At this point, we do not know of any other major bugs we still have as we were able to uncover so many of them with our testing strategy.

Accountability

Testing during development was an expectation. In addition, we documented bugs by creating a text file within our repository. There we identified what the error was followed by how to recreate the error and which file(s) the bug will likely need to be fixed in. We communicated primarily through groupme and sent a message to the group chat every time someone pushed a major commit. Additionally, we adapted and strictly enforced the following commit message format: "[files_modified] details of change" to allow us to better target when files were changed and traceback broken code quicker.

<u>Division of Labor</u>

- **Aayush worked primarily on the network connectivity for the project. He found an implemented the external libraries that we use to connect to the server and wrote the modules relating to the server: server.ml, oclient.ml, loml_client.ml, and api.ml. As part of developing the network connectivity, he spent several hours designing the structure of the API, including endpoints, request types, and request/response formats. He also requested and configured coecis server space that allowed us to implement a lot of our code without having to recreate the database schema and the data within it. Since the API was central in tying all the other modules, he was heavily involved in working on everyone else's code to make sure everything followed the API design, interacted with the network correctly, and was producing/expecting the correct data types. He spent an estimated 75 hours on the project.**
- **Samantha worked primarily on the main, where she wrote the logic for the REPL that prompts the users to enter data about themselves, swipe on profiles and get their matches. She also worked on the Student module which explains the types of various fields that describe the students and calls a lot of the functions that allow the student to update his or her profile. She also worked on the swiping module which handles the results of the students' swiping and writes the results to the database. Just like the API, the main was an integral part in tying together all the other modules and thus she worked extensively on all the other modules as well to make sure the integration worked as planned. She spent an estimated 65 hours on the project.**
- **Savarn worked primarily on the professor and matching modules. He implemented the logic that allows the administrator to set the start dates for the periods and import the initial class of students and their passwords. He also implemented the algorithm that is used in the matching module to use all the data about the students and their swiping results to generate their final pairings. Lastly, he helped reconfigure some of**

the database endpoints to ensure swipes and matches were being processed correctly. He spent an estimated 35 hours on the project.

- **Myra implemented the database module which makes all set and get requests to the database using the correct data and types expected by the other modules. In order to do this, she researched and implemented various external dependencies. She first used the pgocaml library to implement a postgres database but later had to refactor that code into the ocaml-mysql library due to connectivity issues with pgocaml and lack of documentation. The primary challenges here were learning how to use the external library and coordinating with all the other modules to make sure the types of all variables were being handles correctly. She spent an estimated 30 hours on the project.**

Appendix

Chart 1:

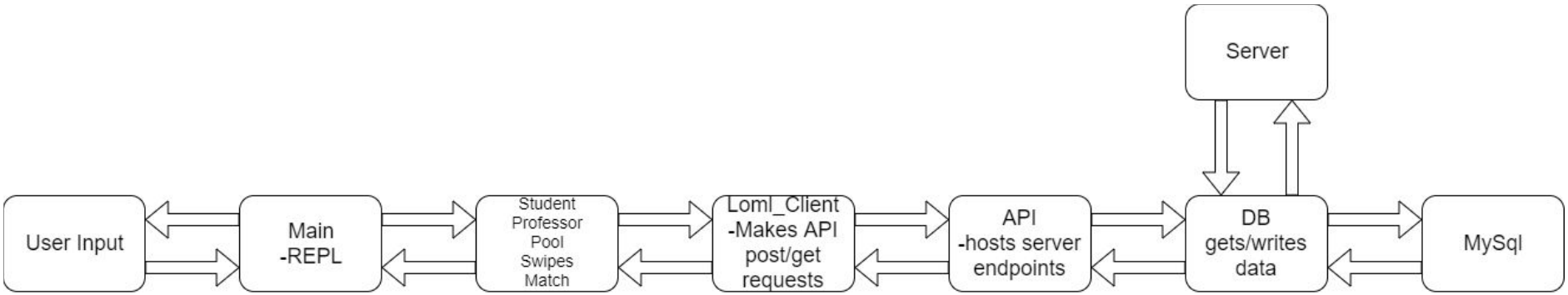
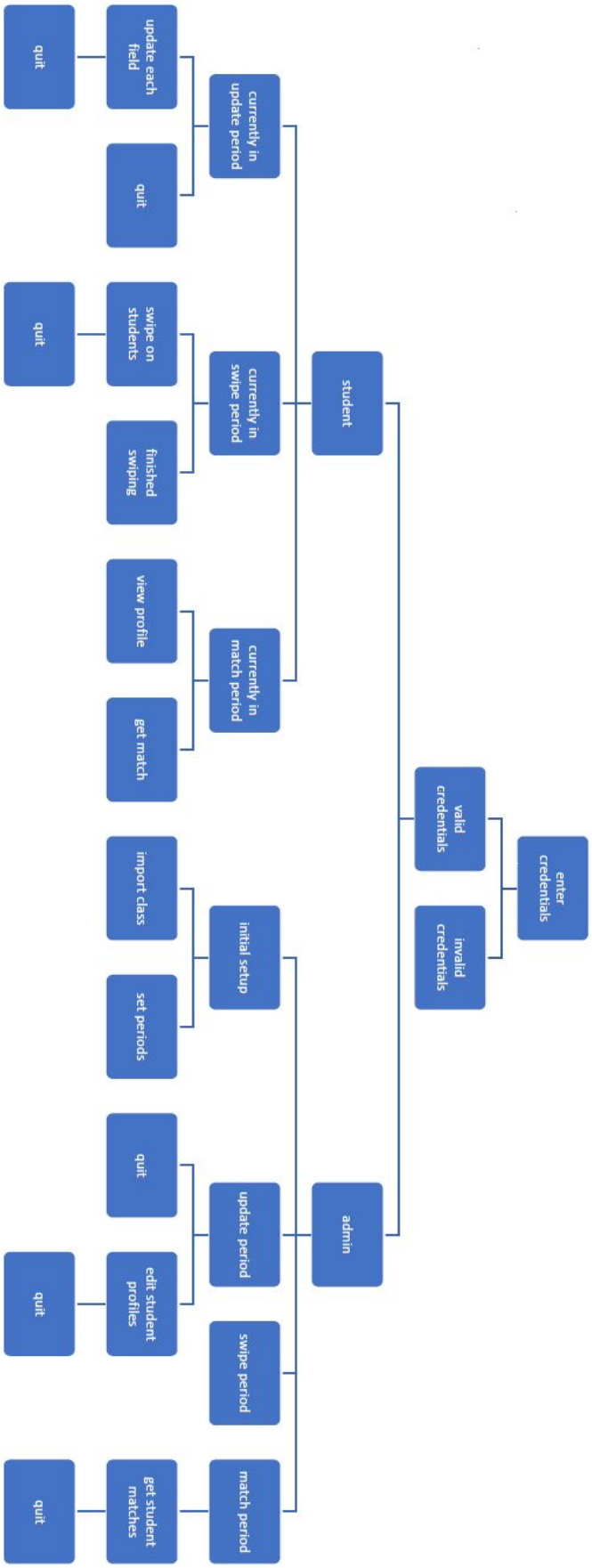

Server
User Input
Main
-REPL
Student
Professor
Pool
Swipes
Match
Loml_Client
-Makes API
post/get
requests
API
-hosts server
endpoints
DB
gets/writes
data
MySql

Chart 2:


enter credentials
valid credentials
invalid credentials
student
currently in update period
update each field
quit
quit
currently in swipe period
swipe on students
quit
finished swiping
currently in match period
view profile
get match
admin
initial setup
import class
set periods
update period
quit
edit student profiles
quit
swipe period
match period
get student matches
quit