3.1

Solution 1:

Solution 1 would involve encrypting the user's information, then storing the user's data in a big JSON file. This way it would be easily portable, and very easily readable. This idea was abandoned due to its poor security, and scalability becoming an issue. It also made it difficult to order the information correctly. If there were 1000 users, the system would need to check every entry in this JSON file for the user's information in a sea of other user's information.

3.2

Solution 2:

Solution 2 involved using MySQL to store the user's information. There would be 2 separate databases, one with users, and another database containing user tables. This way it would be easier to separate what information is being accessed. However, this idea was abandoned due to the issue of interfacing java with MySQL. It would be a nightmare to troubleshoot if something went wrong (making testing much more difficult), because it may not initially be clear if the issue is with the java program, or the MySQL instance. If the MySQL instance had some sort of error that was not immediately visible, or if the java code had some issue that was not immediately visible, it may take some time to figure out which one it is, and then some more time to actually solve the problem. Also, systematic testing using JUnit would be practically impossible to do across the entire system, as it would not have direct access to the database, and would only be accessing it though whatever interface was set up. The final method chosen would be much more flexible and much easier to troubleshoot.

(I just realised now, on July 31 2025, that I forgot to upload this document on time. I apologise for this error, and I hope that this at least puts my final design document into context).