2021

Group 25 – Final Project Report

DVM Relational Database Management System

DHRUMIL RAKESH SHAH
dh647095@dal.ca

MANRAJ SINGH
mn697903@dal.ca

VISHVESH NAIK
vishvesh@dal.ca

CSCI 5408 – Relational Database Management System

Problem Statement:

The CSCI 5408 final project required us to build a Relational Database Management System in JAVA by exploring various data structures that could be used to create an efficient, reliable, robust, and maintainable relational database management system. The idea is to build a system that should register users following security measures like password encryption and allow users to perform database operations that any database management system supports. Like creating databases and tables, persisting data, retrieving the stored data, and updating or deleting the data. The primary focus of the project is to work on data persistence. The application should also be able to create SQL Dumps and ERD diagrams from the persisted data. The application built is a console-based application demonstrating the relational database management system.

Group Contribution:

We have worked on this project as a group to make it a success. Regular meetings and discussions were held to plan, assess and resolve any issues faced relating to the project. Along with that, we would brainstorm how each module and functionality of the application will work. We, as a group, focused on clearly mapping out how various modules will interact with each other. As we started working on the project, we would post updates, keep other team members informed about what has been done, and request them to take the latest pull from the GitLab repo. Below are some screenshots of the meetings and the discussions that we had during this project. Coordinating and having regular meetings helped us resolve various issues that we faced during the project's development.

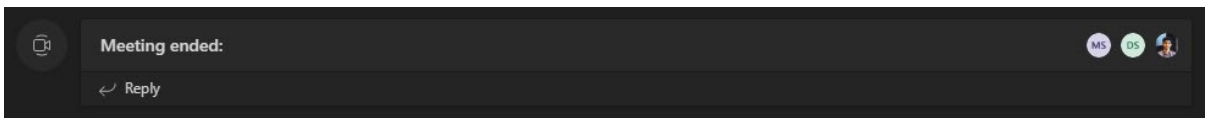

Figure 1. Meeting Log 1

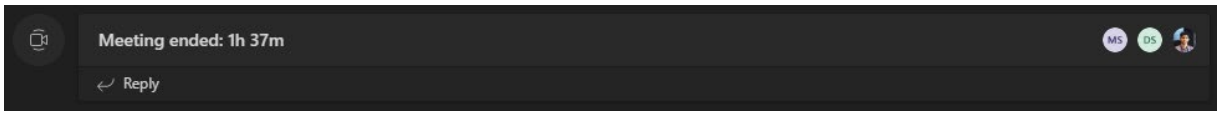

Figure 2. Meeting Log 2

Figure 3. Meeting Log 3

Figure 4. Meeting Log 4

Figure 5. Meeting Log 5

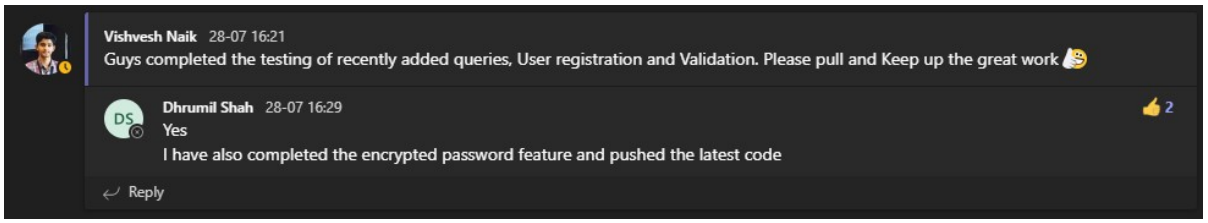

Figure 6. Meeting Log 6

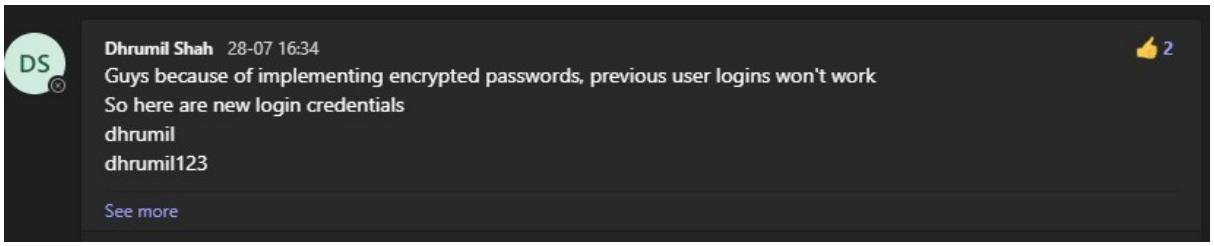

Figure 7. Meeting Log 7

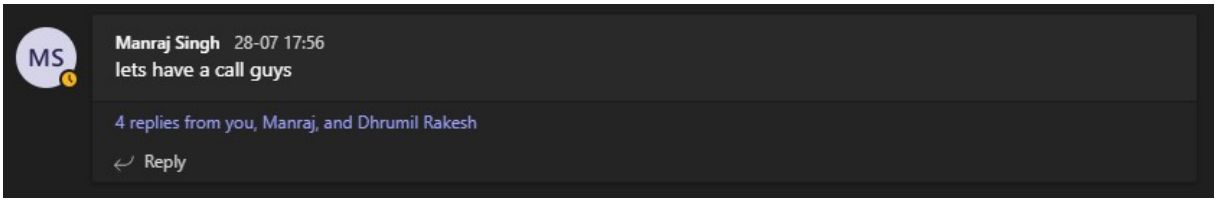

Figure 8. Meeting Log 8

Figure 9. Meeting Log 9

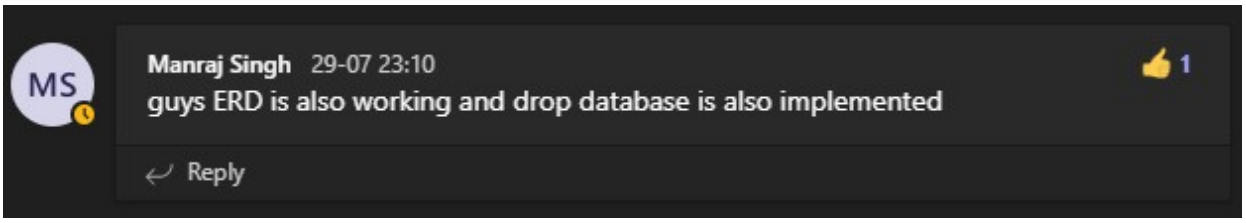

Figure 10. Meeting Log 10

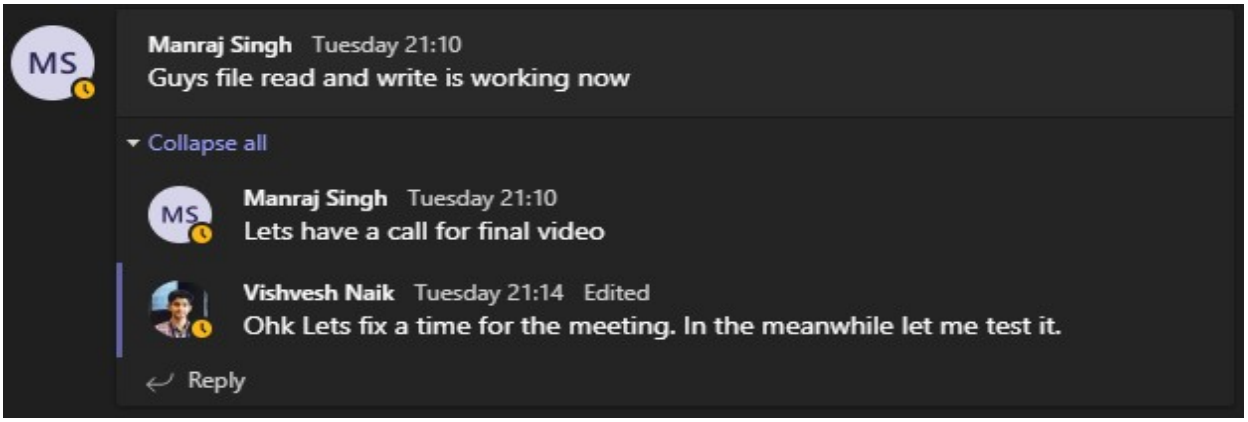

Figure 11. Meeting Log 11

Team Coordination:

This project was possible because of the efforts of the entire team. We made sure that we got to know each other well so there would be no communication gap and we would all be on the same page. Regular meetings and discussions helped in achieving that. Each member of the group contributed their fair share to make this project a success. We followed a unison approach that each team member knows and gets to work on the end-to-end development flow. This enabled us to learn new things and better understand the application and its flow.

Individual Contribution:

1. **Dhrumil Rakesh Shah (B00870600)**
    a. Designed and developed the new user registration module for the project.
    b. Designed and developed the logic for checking whether the user already exists in the DVM Relational Database application.
    c. Used SHA-256 Hashing algorithm for encrypting the passwords for better security of the application.
    d. Contributed to the tokenizer part for query parsing.
    e. Designed and developed the logic for generating log files and recording three types of logs throughout the application.
        i. General Logs
        ii. Query Logs
        iii. Event Logs
    f. Contributed to writing the logic for INSERT, SELECT, UPDATE and COMMIT database queries.
    g. Designed the initial user menu of the application.
    h. Worked on formatting the integrating the entire application along with version control.
    i. Worked on creating documentation for the final project report.

2. **Manraj Singh (B00877934)**
    a. Worked on the initial setup and project structure of the application, created a boilerplate.

b. Contributed to the tokenizer part for parsing the query.
c. Designed and developed the Create Database and Table logic along with Drop Table and Use Database.
d. Contributed to developing INSERT, SELECT and UPDATE queries.
e. Developed the ROLLBACK feature.
f. Developed the Primary and Foreign key features.
g. Designed and developed the recursive logic for the SELECT query.
h. Contributed to developing the SQL Dump feature.
i. Designed and developed the ERD feature.

3. **Vishvesh Naik (B00885077)**
   a. Analysed all the scenarios that a user might face while using the application.
   b. Generated all the test cases from the resulting scenario of the analysis.
   c. Testing the queries in the system with all the errors and validation of proper output.
   d. Optimizing and flourishing all the queries to make it more user friendly in CLI.
   e. Contributed to developing CREATE Database, DROP Table queries.
   f. Contributed to developing USE Database & SQL Dump of the Database.
   g. Worked on formatting the integrating the entire application along with version control.
   h. Worked on creating documentation for the final project report.

Understanding of the Problem:

The primary focus of the DVM Relational Database Management System is to perform the operations of an RDBMS efficiently. Persisting the data to files and then parsing them to perform various operations is also a key focus. The application is console-based built using JAVA, whose highlight is its ease of use for querying the database using predefined query structures. The challenge that we faced was how to persist the data and then read it later on, along with validating the queries entered by the user. The validations are done based on the predefined rules of our database management system. Data structures were carefully chosen for the application in order to fulfill and support the application requirements. Above all, the various modules in the application should function without affecting its flow.

Project Demo View Recording Link:

https://youtu.be/Y3xlyWI7SYQ

Project GitLab Repo Link:

https://git.cs.dal.ca/manraj/group25_final_project.git

Initial Algorithm:

Below is the initial algorithm that we had in mind while developing the DVM Relation Database Management System. The algorithm was well thought of at the project's inception, leading to clarity while developing the application.

1. Start the application
2. Register User
   2.1. If the user already exists, go to step 3.
   2.2. If the user does not exist, then the user registers.
   2.3. Storing the user information into a file, go to step 3.
3. User Login
   3.1. If the user credentials are valid, go to step 4.
   3.2. If the user credentials are not valid, go back to step 3.
4. Selecting the Schema
   4.1. If Schema validates successfully, go to step 5.
   4.2. If Schema is not validated, then Change the Schema, go to step 4.
5. User enters an SQL query
   5.1. If the query is valid, go to step 6.
   5.2. If the query is not valid, go to step 5.
6. The Parser parses the query
   6.1. If it is a Create query
      6.1.1. If the Table exists, go to step 6.1.2.
      6.1.2. Show error to the user that the Table already exists.
      6.1.3. If the Table does not exist, go to step 6.1.4.
      6.1.4. Create a new table.
   6.2. If it is an Insert query
      6.2.1. If the Table exists, go to step 7.
      6.2.2. If the Table does not exist, go to step 6.2.3.
      6.2.3. Show error to the user that the Table does not exist.
   6.3. If it is a Select query
      6.3.1. If the Table exists, go to step 7.
      6.3.2. If the Table does not exist, go to step 6.3.3.
      6.3.3. Show error to the user that the Table does not exist.
   6.4. If it is a Delete query
      6.4.1. If the Table exists, go to step 7.
      6.4.2. If the Table does not exist, go to step 6.4.3.
      6.4.3. Show error to the user that the Table does not exist.
7. Execute the query
      7.1.1. If query execution is successful, go to step 8.
      7.1.2. If query execution failed, go to step 7.1.3.
      7.1.3. Show error to the user that the query failed to execute.
8. Update the relevant tables in the DB.
9. Update General Logs
   9.1. If updated, go to step 12.
   9.2. If not updated, go to step 9.3.
   9.3. Show error to the user.
10. Update Event Logs
   10.1. If updated, go to step 12.
   10.2. If not updated, go to step 10.3.
   10.3. Show error to the user.

11. Update Query Logs
    11.1. If updated, go to step 12.
    11.2. If not updated, go to step 11.3.
    11.3. Show error to the user.
12. Update successful
    12.1. If the user wishes to continue
    12.2. If yes, go to step 5 and for different Schema, go to step 4.
    12.3. If no, go to step 13.
13. End the user session by exiting the application.

Implementation Plan:

1. Initial Data Structure Details:

For this project, we will use **Linked HashMap**, **ArrayList**, **Stack,** and **Queue** data structures. Linked HashMap is implemented using the Map interface. We chose this data structure because of its ease of use and various advantages. It is a combination of HashMap and List data structure utilizing the properties of both. Time Complexity of finding any value using a key is O(1), and also the nodes are added serially to be fetched in the same order, which is not possible in a standard HashMap.

Linked HashMap node example:

LinkedHashMapEntry<key, value> previousInput;

LinkedHashMapEntry<key, value> nextInput;

Object keyObj;

Object valueObj;

HashMapEntry<key, value> nextEntrySameIndex;

As we can observe from above, each Linked HashMap is an Array of nodes where an individual node consists of its previous and next node pointer and contains the current node's key and value objects HashMap Entry for repeated hashcode index but separate keys.

Each Linked HashMap also consists of an algorithm to find the index and hashCode of the node in the array of nodes, calculated using the key. In an ideal scenario, the algorithm produces a unique index and hashcode for every key, but in reality, there are chances of hash Collision for which the list structure of hashMap is used in which each node of the hashmap can have the next node pointer referencing to the node with the same index but different key. If the hash collision happens due to the same key, then the value of that key is overwritten, and the previous value is returned with the put() method call.

There are also chances when the node's array length exceeds, then the array's length is increased by double, and all nodes are moved from the old array to the new array. This operation has a time complexity of O(n).

Stack is the data structure that stores and retrieves data in a LIFO (Last in First Out) manner. And Queue is the data structure that stores and retrieves data in the FIFO (First in First out) manner. Both are mainly used for logging purposes.

2. Implementation Environment:

- **Programming Language** – JAVA
  - Java classes are rich and support user-defined data types.
  - Methods of Java classes provide new functions in SQL.
  - Java also provides collections such as Map, List which can be used to store or retrieve data.
  - Java also supports a relational database management system and is platform-independent.

3. Input Validation:

- The system will first perform input validations for Login and Register functionalities.
- The system will check if any of the Schema is selected.
- The system will initially parse the SQL statement that is written to check if it is from;
  - Create
  - Insert
  - Update
  - Delete
  - Alter
  - Use statement
  - Drop
- A pre-defined function of the String class will be used .contains() to determine the query's functionality.
- If the query does not have any such components (keywords), an error will be thrown.
- Specific and precise validations can be done by using the loops in Java.
- RegEx can be used to validate the query structure that is being given as input to the system.
- We can also use the split() function to differentiate the functionalities of the query.
- After all, this done, the values are pushed onto the Stack, as it follows the LIFO structure;
  - Validation is done for the brackets being closed after we find an open bracket.
  - Validation is done to check if the number of columns and the retrieved values are equal or not.
  - After completely parsing the query, Stack should be empty, so if anything is remaining in the Stack means that the query is invalid.
  - The main thing to validate using the Stack is to check whether the query is balanced or not.

4. Boundary Cases Validation:

- After query validation successfully completes, boundary cases are considered.
- The Schema and Table names will be validated. The system checks the directory looking for the Schema and the respective Table. If not found, then an error is shown.
- After the Table is being successfully identified, the system will read the Metadata file.
- The validation is done for column names, type, and size (parameter size) after validating the Table.
- Query values will then be parsed using a Parser based on the metadata.
  - Example integer should not be able to store text/varchar values.
- The primary key is validated, i.e., no duplicates are allowed.
- After that, the foreign key is validated based on its constraint. The value should be NULL or present in some other table that is linked to this Table.

- Deletions won't be completed if there are dependency constraints like a foreign key constraint, need to delete the Table whose foreign key is present in the current Table.
- Rollback and commit operations are also validated, and various logs being recorded and written in the files.

5. **Backend Operations:**

Database:

- The data in our code will be stored in ArrayList<LinkedHashMap<Key, Value>>format.
- Internal Linked HashMap will store the database column name as the key and the column row value for that row as the value.
- Only one row will be pushed to the Linked HashMap, and the whole ArrayList will represent the Table.
- A separate HashMap will be maintained to keep track of all the databases created. Its key-value pair will be the name of the database and the ArrayList reference.
- While storing the rows, if the columns are limited, then the other column values will be set null.

DBMS:

- Users can signup and login into the database. His details will be stored and fetched from a HashMap maintained with the user details.
- When the user login to the database, it will generate a thread for the user.
- After validating the query, its timestamp will be stored as query logs along with the query.
- State Logs will also be updated after every query execution with that query timestamp.
- Crash details and the exceptions will be printed in the event logs and other state details of the database.
- A stack will be maintained in the database to store values like name, row, column, the previous value, current value of the database Event Stack.
- If the user rolls back or is inactive for long, the tables will be reverted back to the last commit using Event Stack.

6. **ACID Properties:**
   - **Atomicity:** Any update in the Table will be made after the commit command. If there will be any exception while updating, the data will be rolled back.
   - **Consistency:** We will keep track of the database state after each transaction; whentrying to delete a table, all the foreign keys referencing the Table's primary key will be deleted first.
   - **Isolation:** A separate HashMap will be maintained to track the table availability. So that no two users can change the Table at the same time.
   - **Durability:** After a successful commit, the data will be visible to all the users. Even in the case of an exception, the data will still be preserved.

DVM Relational Database Management System Flow Chart:

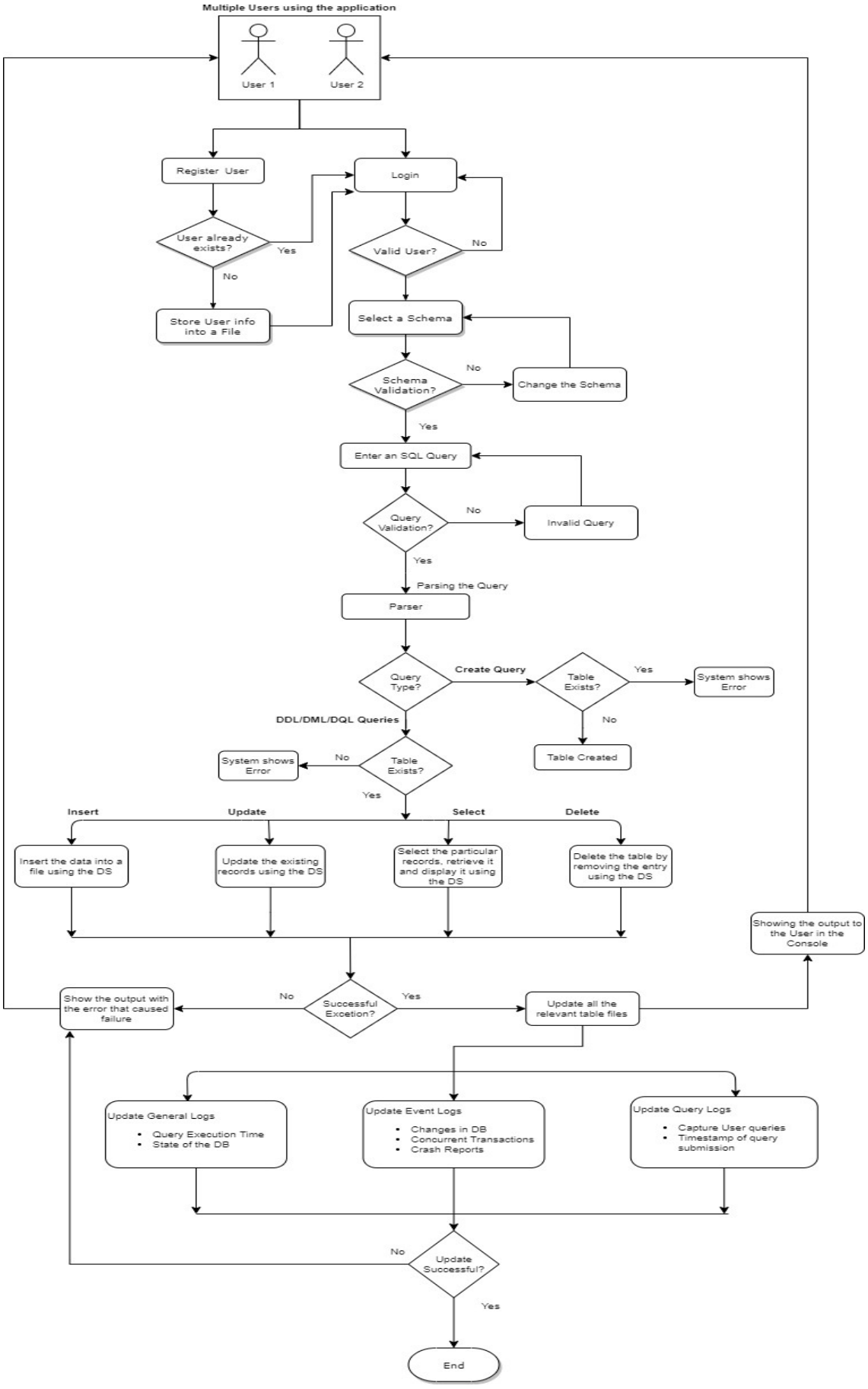

Figure 12. DVM Relational Database Management System Flow Chart

Use Cases:

Following are the use cases for the DVM Relational Database Management System.

1. Display the main menu to the user.
2. New user registration.
    a. Checks whether the user already exists.
    b. If not, then registers the user
3. User login to the application.
    a. Password encryption using the SHA-256 hashing algorithm.
4. Create a database.
    a. Query validation is done.
    b. Checks if the database name already exists.
    c. If not, then creates a new database.
5. Create a table.
    a. Query validation is done.
    b. Checks if the table name already exists.
    c. If not, then creates a new table.
6. Insertion of data into the table.
    a. Query validation is done.
    b. Checks whether the table is present.
        i. If present, then data is inserted.
    c. If not, then an error message is shown.
7. Fetching and displaying the data from tables.
    a. Query validation is done.
        i. If present, then persisted data is fetched and displayed.
    b. If not, then an error message is shown.
8. Update the data in any table.
    a. Query validation is done.
        i. If present, then persisted data is updated.
    b. If not, then an error message is shown.
9. Adding Primary and Foreign keys to the tables and their relationships.
    a. Query validation is done.
10. Deletion of the record from the table.
    a. Query validation is done.
    b. Checks whether the table exists.
        i. If yes, then the record is dropped.
    c. If not, then an error message is displayed.
11. Dropping the tables.
    a. Query validation is done.
    b. Checks whether the database exists.
        i. If yes, then the database is dropped.
    c. If not, then an error message is displayed.
12. Commit the transaction.
13. Rollback any transaction that is not committed.
14. Create SQL Dump.
15. Displaying an ERD diagram on the console.

Implementation Details:

There are many modules/features in the DVM Relational Database Management System and this section focuses on the pseudocodes for each of them.

User Login Pseudocode:

1. Accepts username and password.
2. Searches the entered username in the AllUsers file.
3. If user exists
    a. Hash the entered password and compares.
    b. If password matches,
        i. Login the user.
        ii. Else, throw "Enter Correct Credentials" error message.
4. Else, thorw user does not exist exception.

User Registration Pseudocode:

1. Read username and password from user.
2. Searches the entered username in the AllUsers file.
3. If user exists
    a. Show User already exists error message.
4. Else, validate the entered input.
    a. If input invalid
        i. Throw enter correct username or password error message.
    b. Else, register the new user.
5. Save the new users username and password in the AllUsers file.

Create Query Pseudocode:

1. Accept create query from the user.
2. Iterate over the entire query and parse it.
3. Split all column names and data types into array.
4. Check if all datatypes are valid.
    a. Else, show an error message of "Entere correct command".
5. Check if the query has a primary key constraint.
    a. If more than one column has been declared as primary key then show error message.
6. Check if query has foreign key referencing another table.
    a. Iterate user table dictionary to find the other table.
        i. If does not exist, throw error.
    b. If referenced column does not exist, throw error.
    c. If referenced column is not a primary key, throw error.
7. Initialize empty HashMap.
8. Populate HashMap with key as column name and value as column datatype.
9. Iterate user table dictionary.
    a. If table name already exists then throw an error.
10. Create new empty file for storing the data.

Select Query Pseudocode:

1. Accept the query from user which will be validated against predefined select query regex pattern in the application.
2. Select query is divided into various groups using regex pattern.
3. Pattern matcher function splits the query into group assigning the value into variable for table name, column name and query condition. (using the tokenizer to identify the tokens)
4. HashMap is created for fetching the table data from the file.
5. After fetching the data into HashMap, condition in the select query is validated against the data column mentioned in the where clause.
6. Upon successful validation, selected data is fetched from the file.
7. The content is displayed to user as the result on the console.

Update Query Pseudocode:

1. Accept the query from the user which will then be validated against a predefined update query regex pattern in the application.
2. Update query is then divided into various groups using the regex pattern and the tokenizer class using tokens.
3. Pattern matter then splits the query into group assigning the value into variable for table, column name.
4. Data is fetched from the file.
5. After data is fetched, the condition in the update query is validated.
6. Upon successful validation, the selected data is updated.
7. Updated data is then written back to the file.

Drop Query Pseudocode:

1. Read the drop query from the user.
2. Iterate the query and parse it.
3. Check if the table is present in the file storing data.
    a. If not, then show an error message.
4. If yes, then remove the entry of that table from the file.

Insert Query Pseudocode:

1. Read the insert query from the user.
2. Parse the query by itearting over it.
3. Check if the table exists
    a. If no, then show an error message to the user.
4. If yes, then split the query into column names and values.
5. Check if any of the column is set as a primary key.
    a. Compare the value of primary key column with stored values in file
        i. If unique value, then write the record to the file.

ii. If not unique then show an error message.

6. Show a message that record successfully inserted.

Delete Query Pseudocode:

1. Read the delete query from the user.
2. Parse the query and iterate through it.
3. Check in the file if the table exists.
    a. If not, show an error message.
4. Then split the column name and value of the read query.
5. Check for the matching records in the file.
    a. If record does not exits then show an error message.
6. If reccord exists, then delete the record from the file containing data.
7. Display record deleteed successfully message.

Commit Command Pseudocode:

1. Read the commit command from the user.
2. Parse the command.
3. Check if the transactions have completed successfully.
    a. If not then show an error message.
4. If yes, then commit the transactions by persisiting the data in the actual files.
5. Show Commit success message to the user.

Rollback Command Pseudocode:

1. Read the rollback command from the user.
2. Parse the command.
3. Check if the transactions have completed successfully.
    a. If not then show an error message.
4. Check what was the last commit and its state.
    a. If found, then rollbcak to that state.
    b. Discrad the changes made.
5. Show Rollback success message to the user.

NOTE:

For all the above operations entries are made into their approprite log files.

Query Log – All query relaeted logs

Event Logs – All event related logs like change in database, crash reports.

General Logs – All general logs like query execution time, current database.

Testing of DVM Relational Database:

Login Query:

- If the user credentials are not valid, it will throw an error showing "Enter correct credentials."

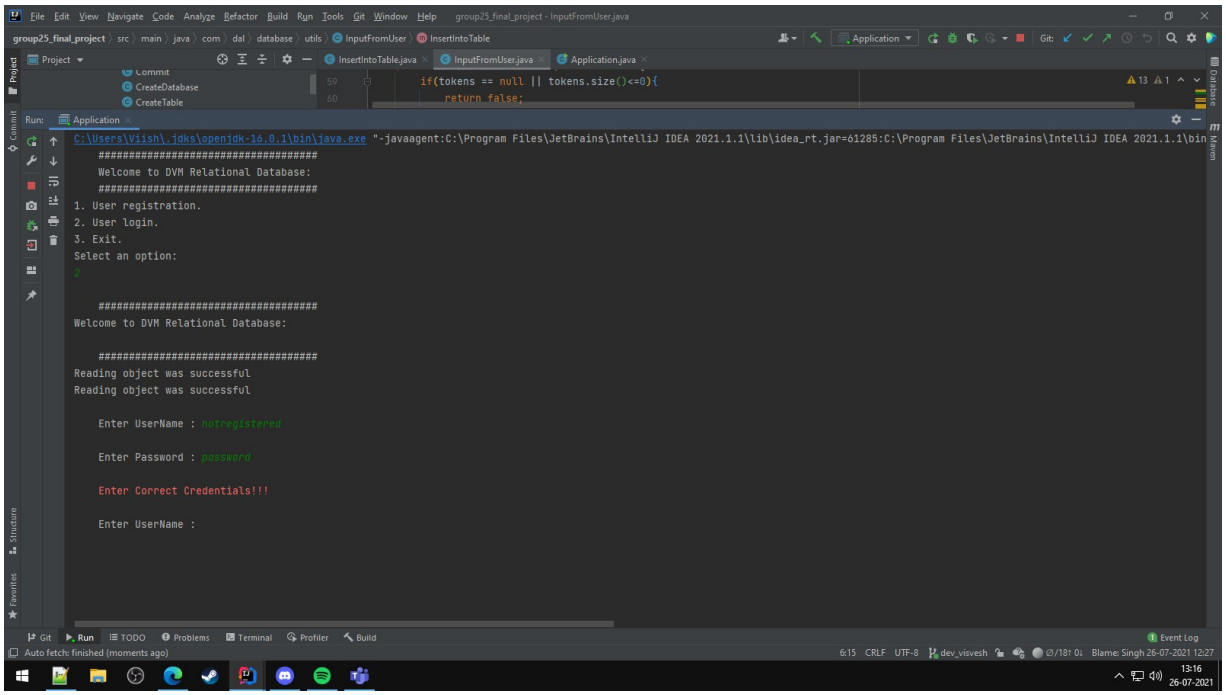

Figure 13. Test Case for not entering correct credentials.

- If the credentials are correct, then it prompts "Login Successful" and initializes the SQL processor.
- One more this thing is that every user of the system will have the username in front of the processor.
- Right now, there are only two valid users in the database – root and manu.

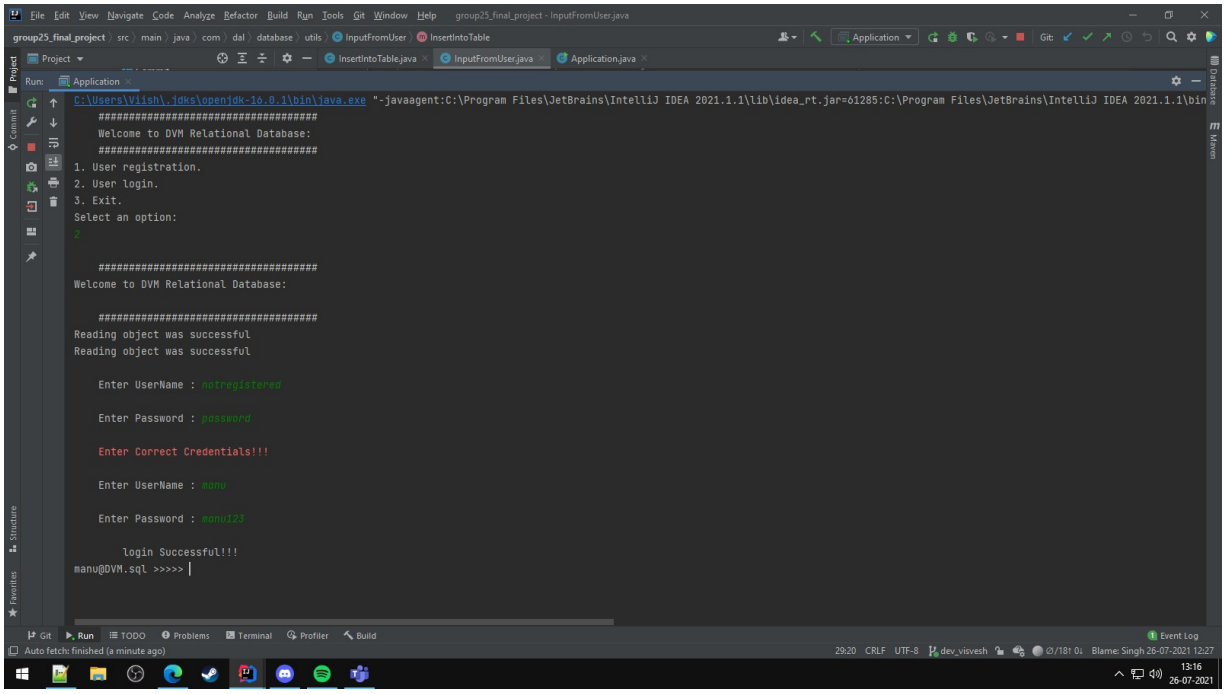

Figure 14. Test Case for successful login.

- All the passwords are encrypted using the hash algorithm (**SHA-256**).

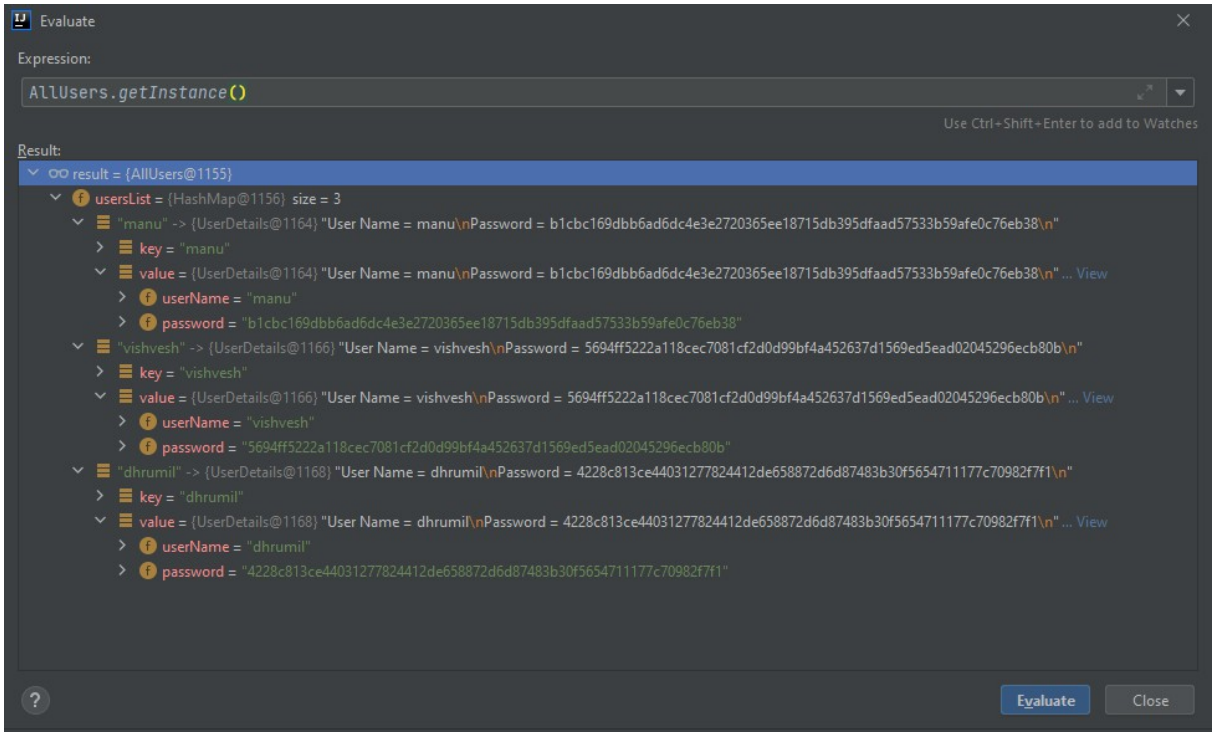

Figure 15. Test Case showing encrypted password.

Exit Query:

- For exiting the system and the application, we can use both 'exit' and 'exit;.' It does not matter with or without a semicolon. The system will accept both.

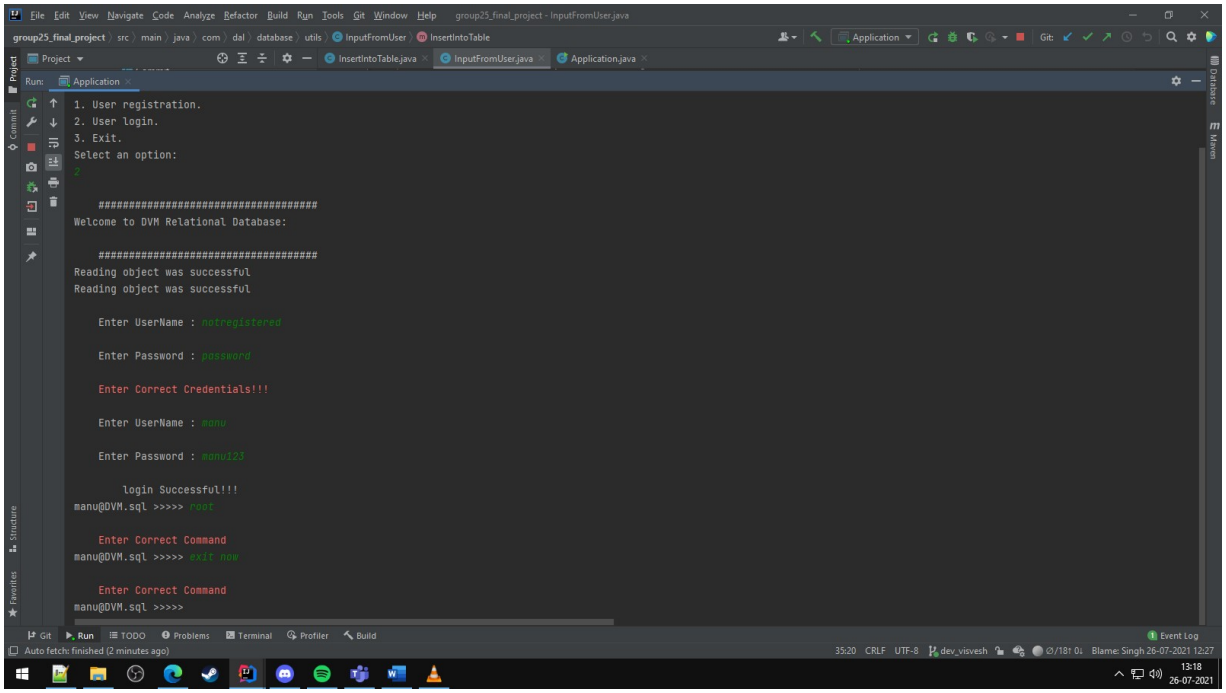

Figure 16. Test Case for an invalid EXIT command.

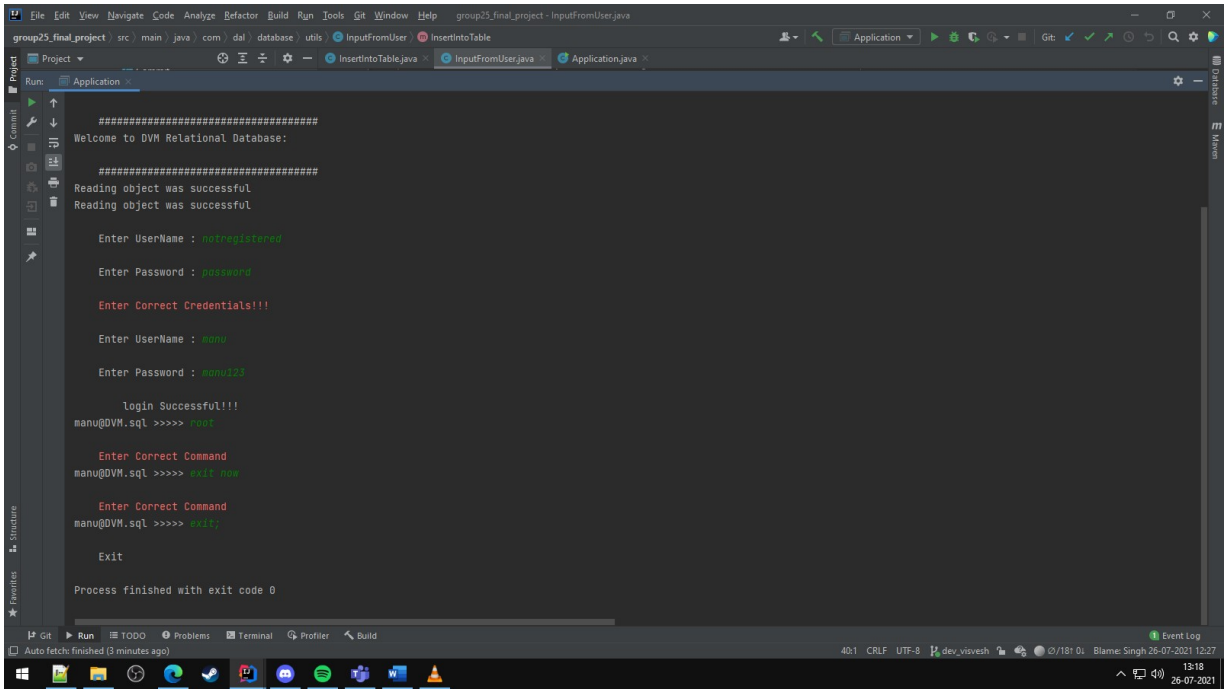

Figure 17. Test Case for a valid EXIT command.

Commit Query:

- Only the 'commit' keyword is used to commit the saves.

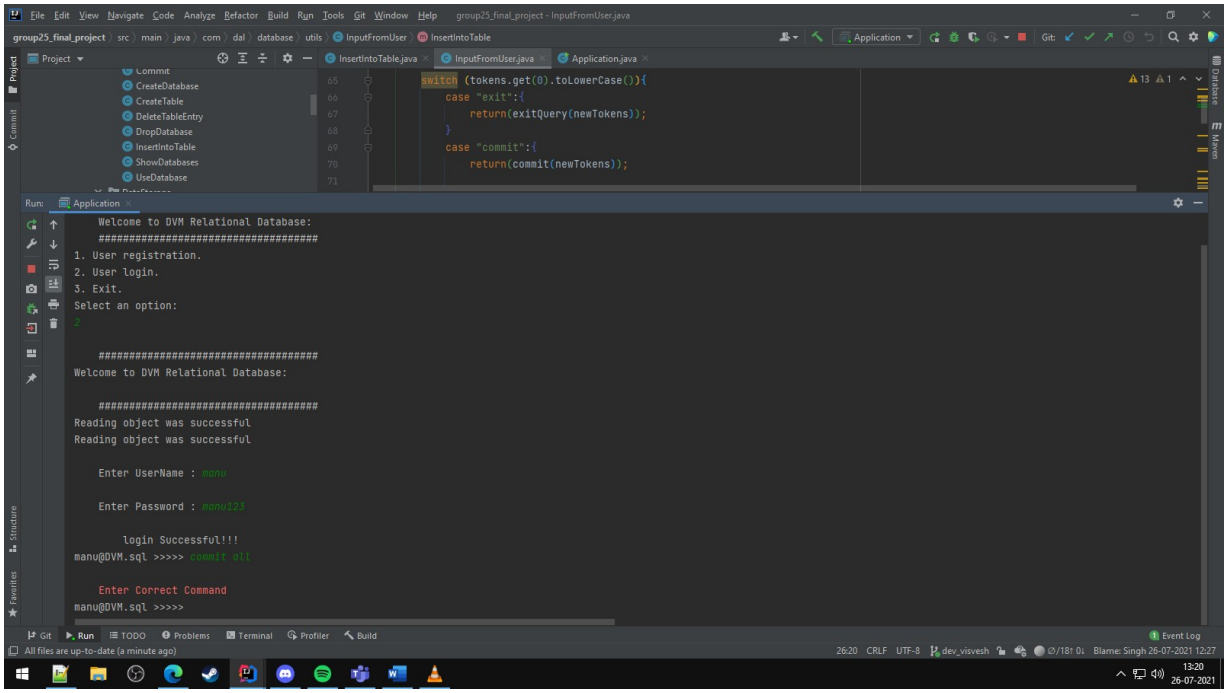

Figure 18. Test Case for an invalid COMMIT command.

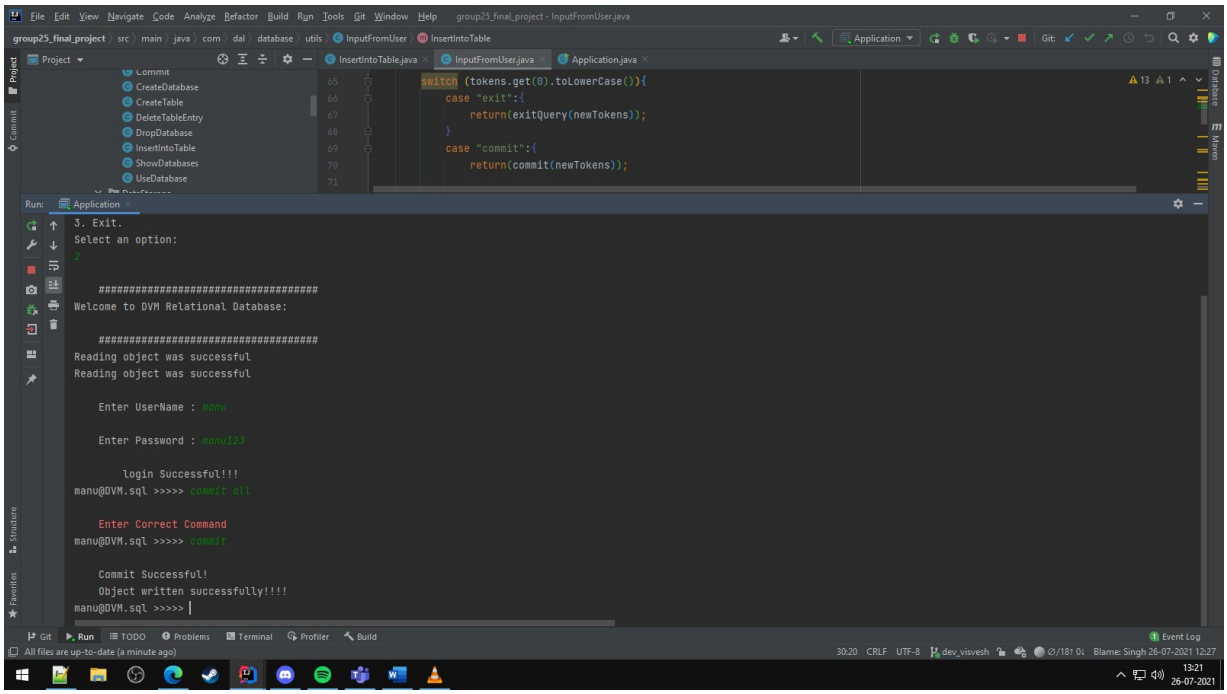

Figure 19. Test Case for successful COMMIT command.

Create Query:

- Create a database query that will generate a new database and write it in the RawData directory.

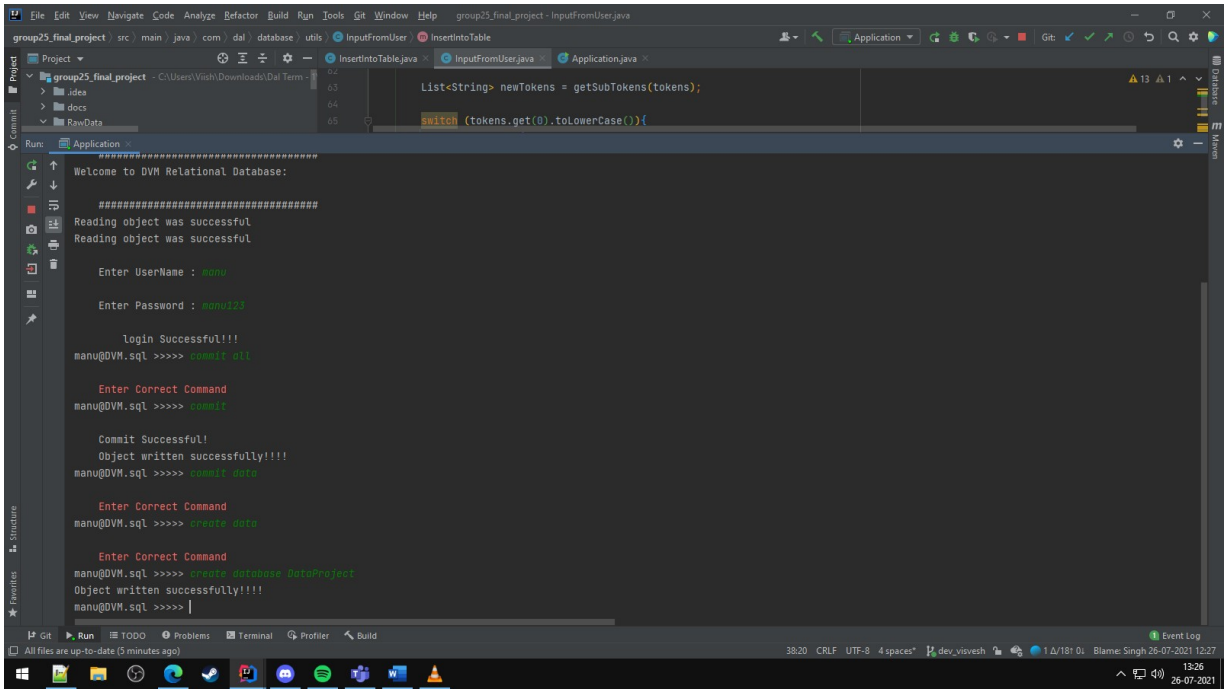

Figure 20. Test Case for CREATE Database command.

Show Query:

- 'Show' query will only accept 'databases' after the use to show.

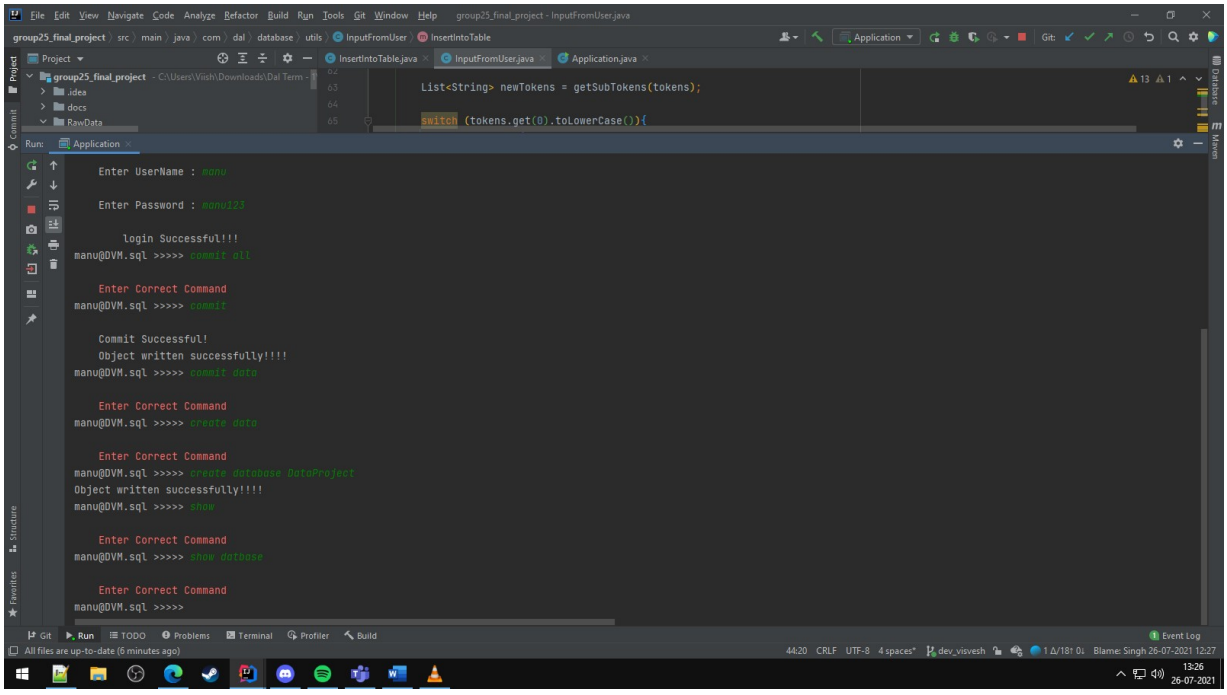

Figure 21. Test Case for an invalid SHOW Database command.

- It will show a list of all available databases in the project.

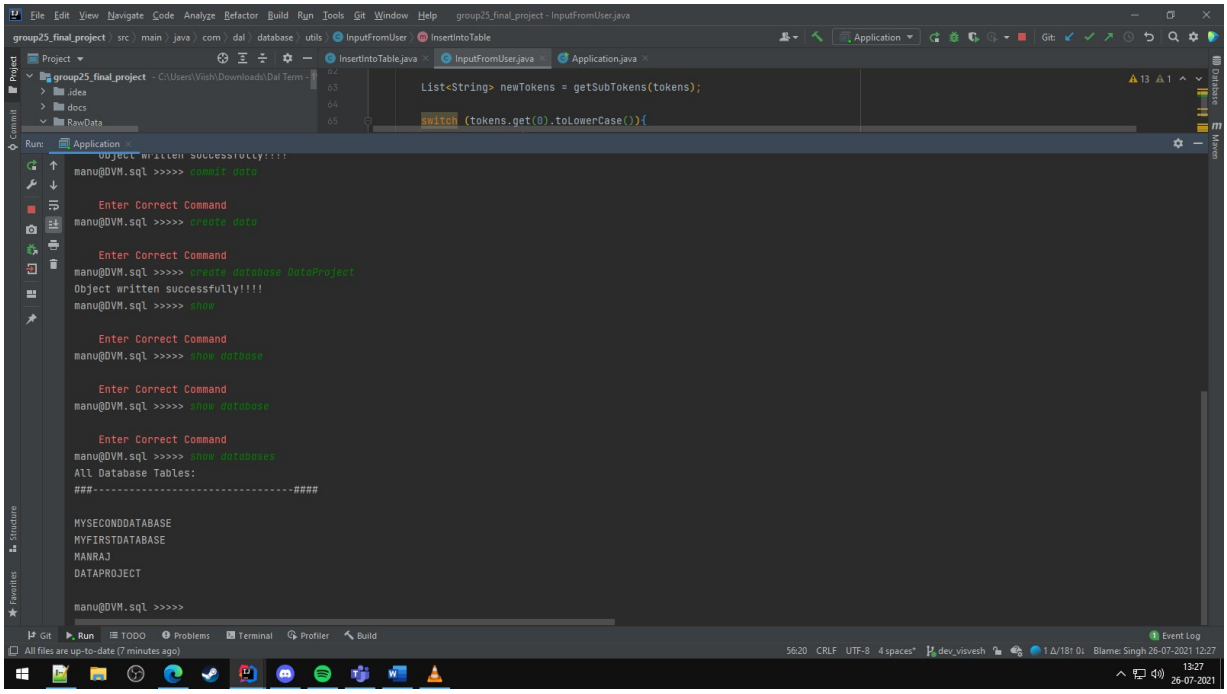

Figure 22. Test Case for a valid SHOW Database command.

Use Query:

- We are using the keyword use without the proper database name from the list.

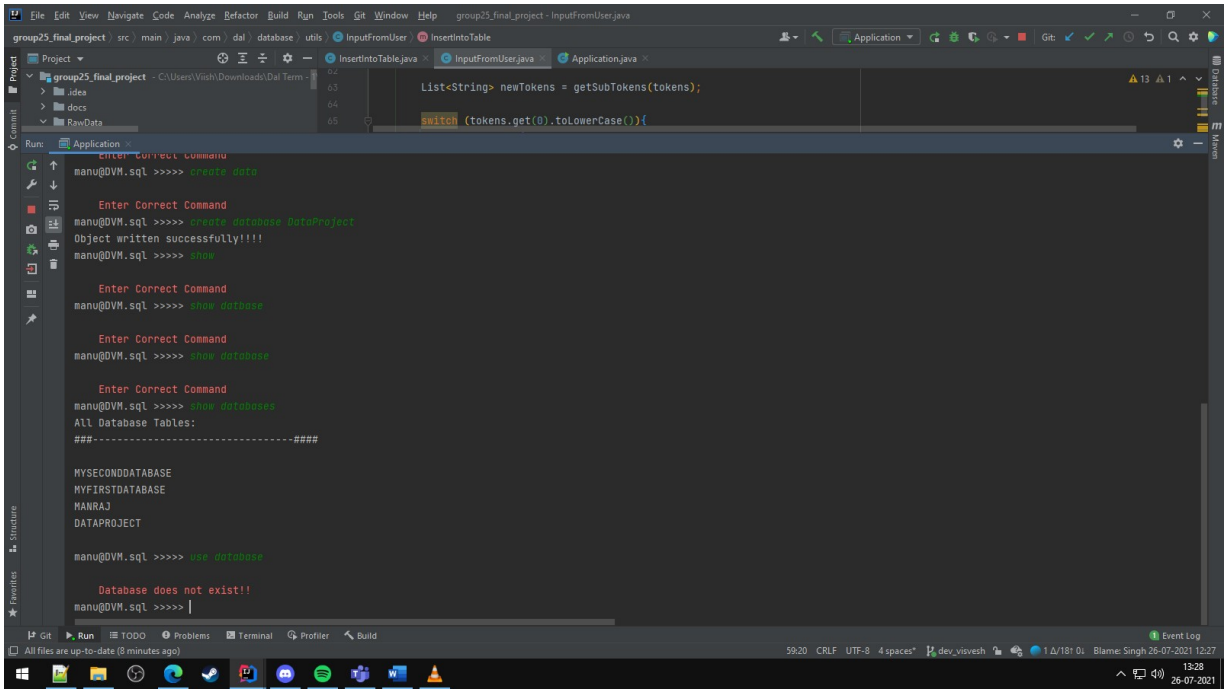

Figure 23. Test Case for an invalid USE Database command.

- You can choose any database from the list by using use 'database_name' to use that database.

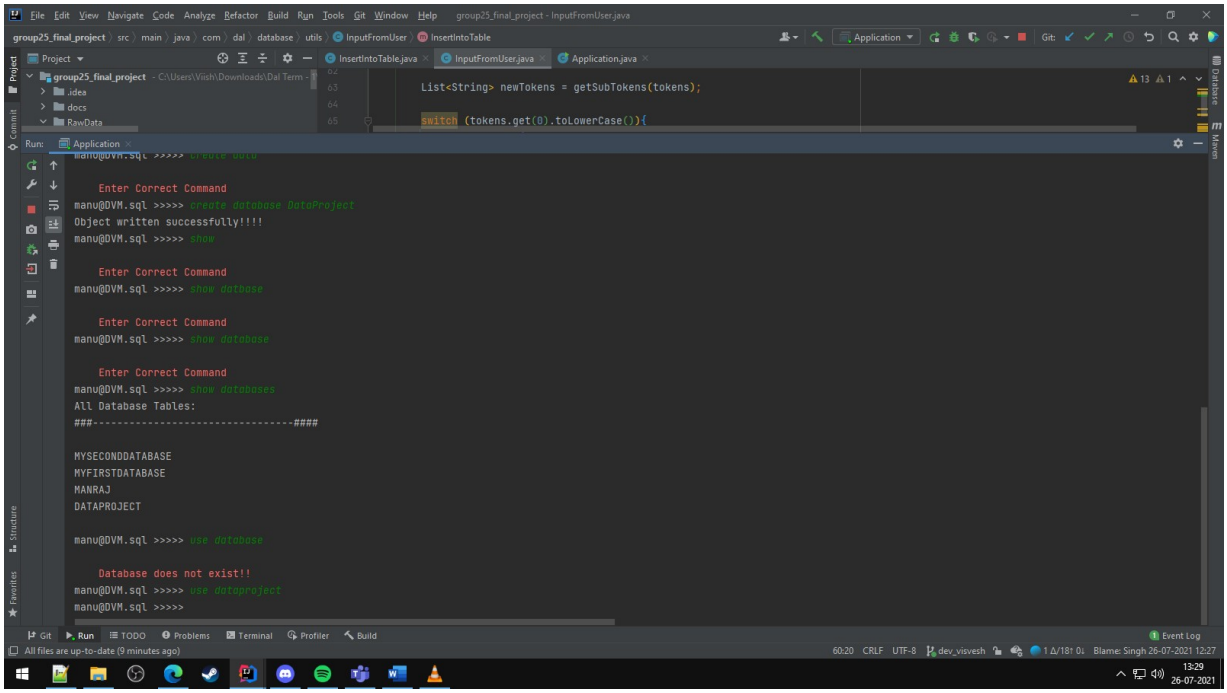

Figure 24. Test Case for a valid USE Database command.

- As you can see * in front of the current using database.

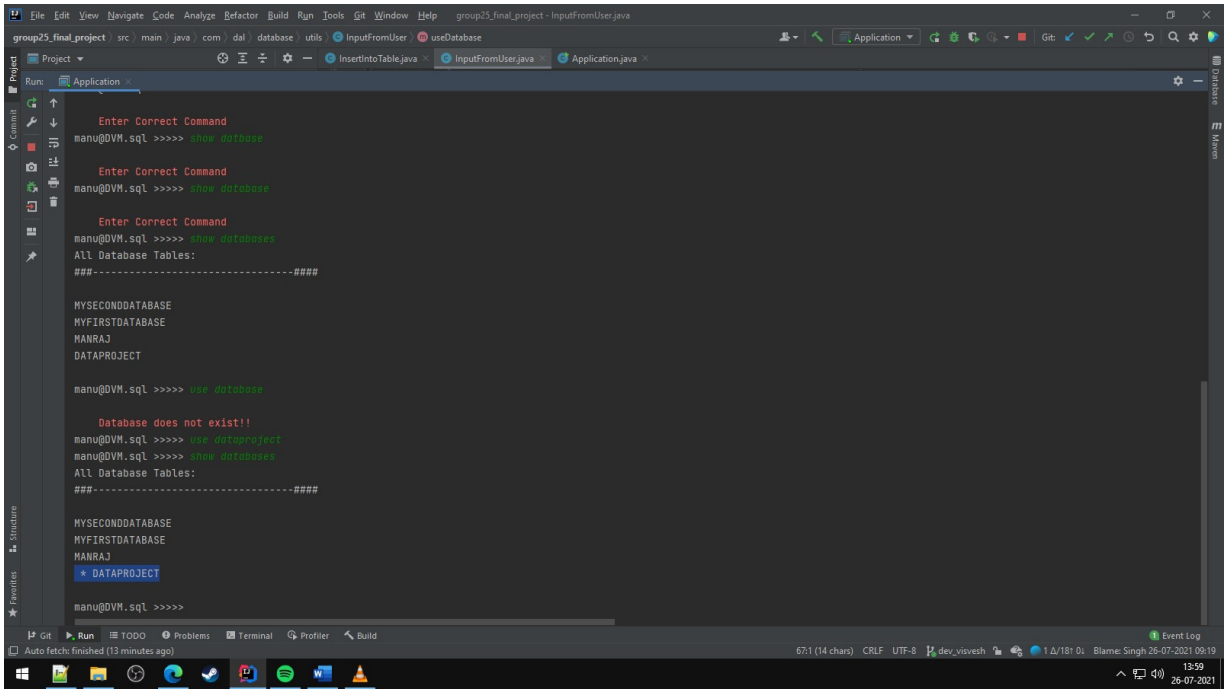

Figure 25. Test Case showing the database currently being used.

Create Query:

- Only valid characters are used as a table name.

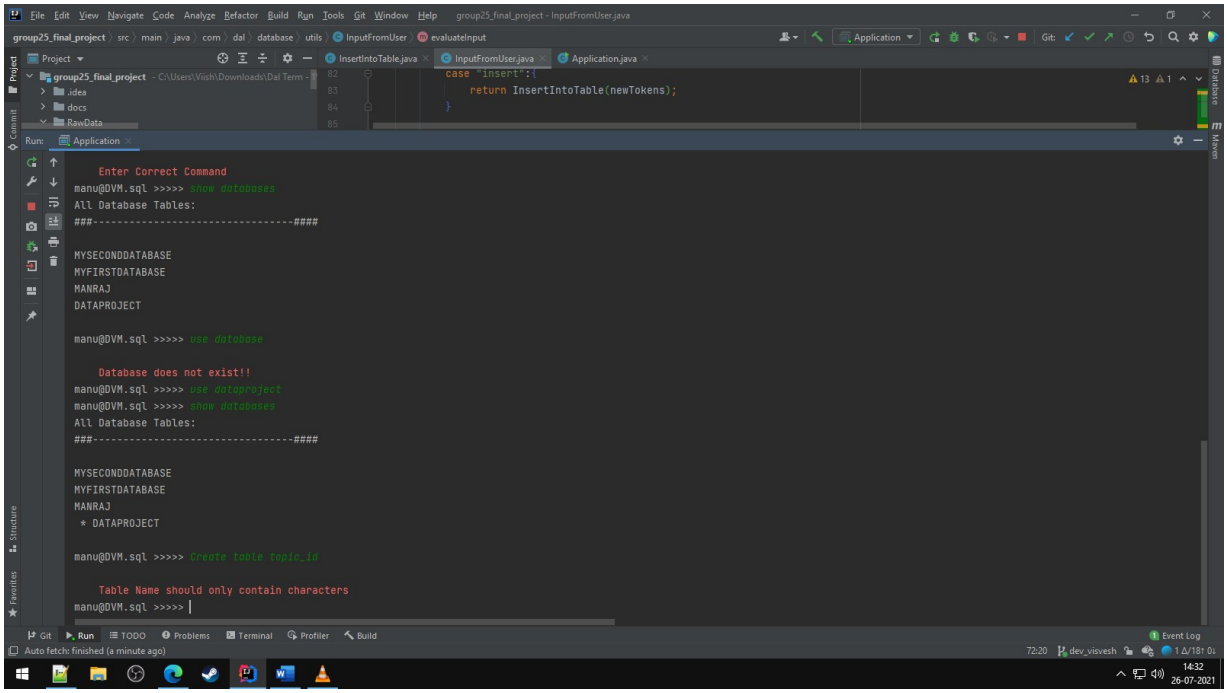

Figure 26. Test Case for an invalid CREATE Table command 1.

- Direct table names are not accepted.

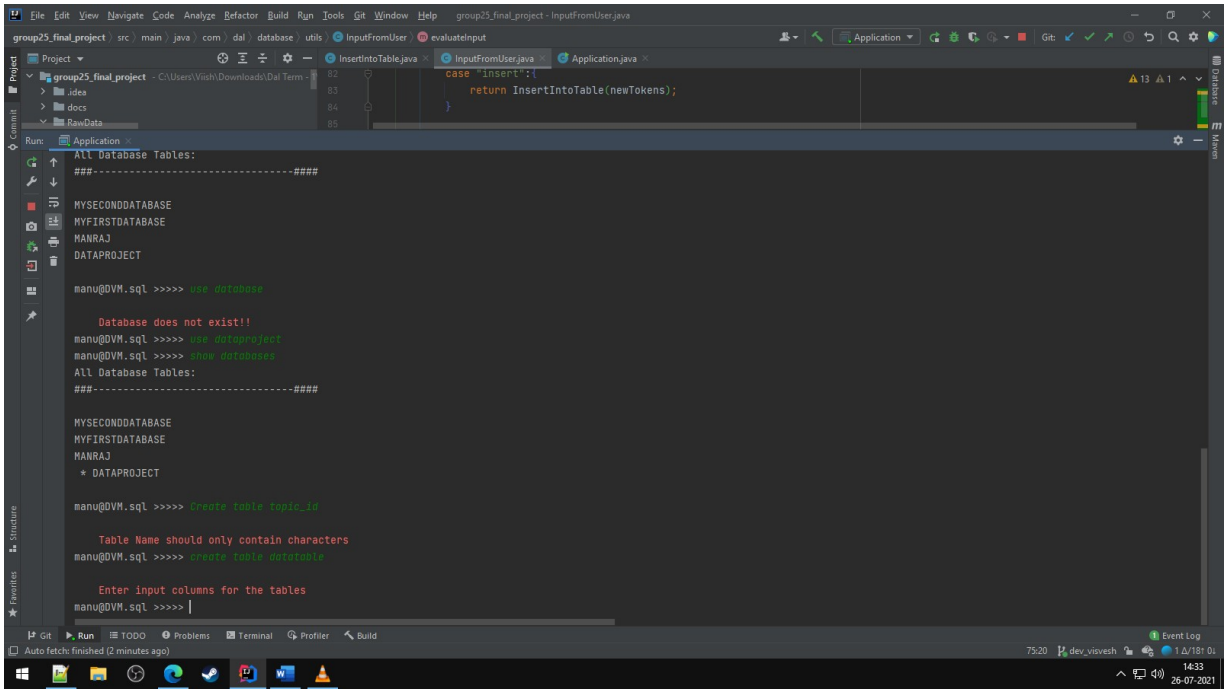

Figure 27. Test Case for an invalid CREATE Table command 2.

- Create a table name requires a valid argument with proper datatypes as an input field.
- Integer or int any will work.

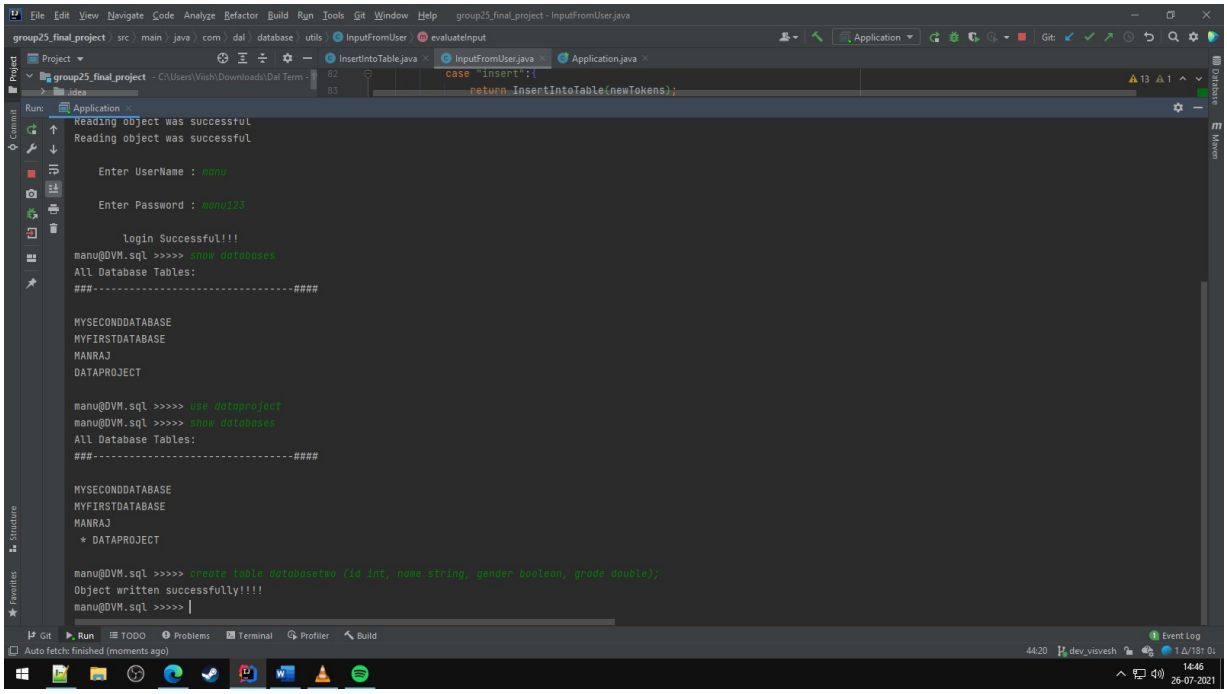

Figure 28. Test Case for a valid CREATE Table command 1.

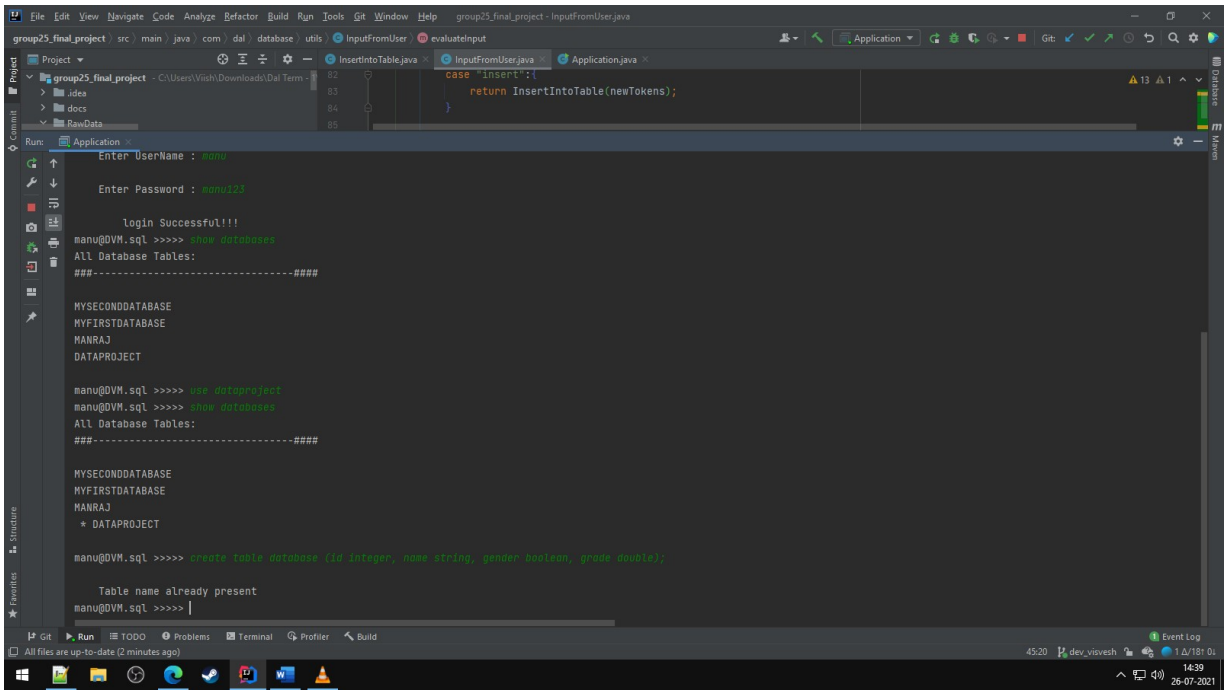

Figure 29. Test Case for a valid CREATE Table command 2.

- From the above image, if you insert unfamiliar data types, it will return an error.
- If you insert proper data types, it will return a proper response.

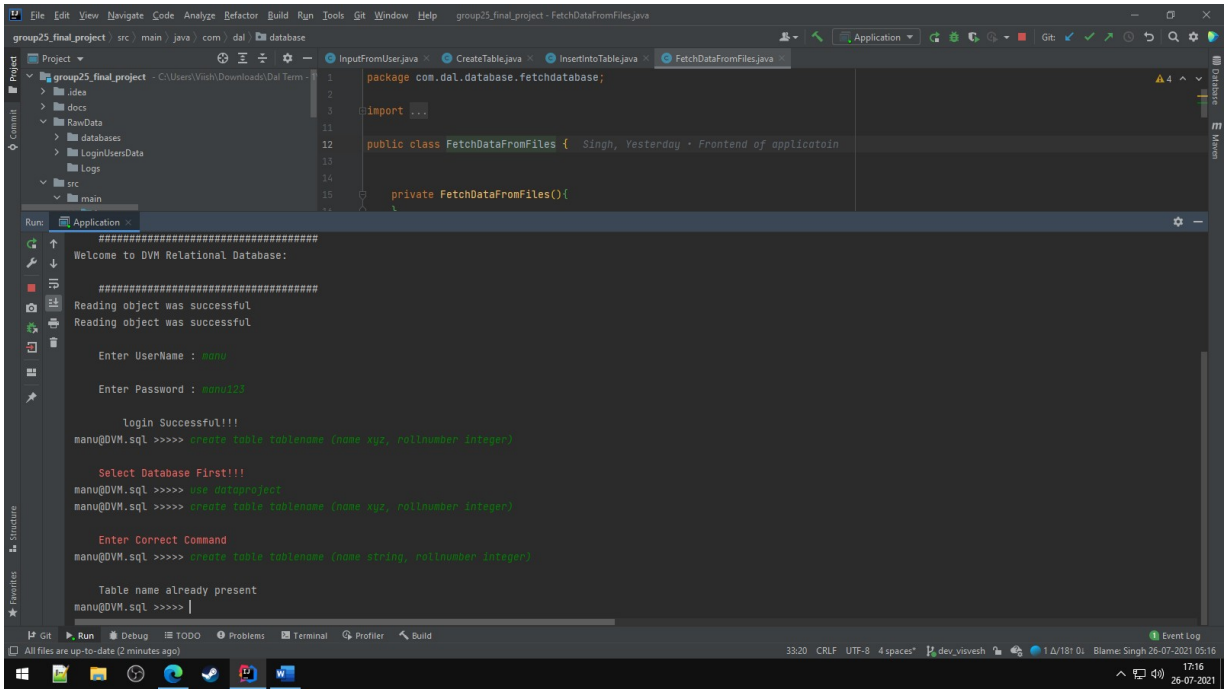

Figure 30. Test Case for a CREATE Table command with valid and invalid data types.

Insert Query:

- If the database is not selected yet, in other words, not used 'use' query.

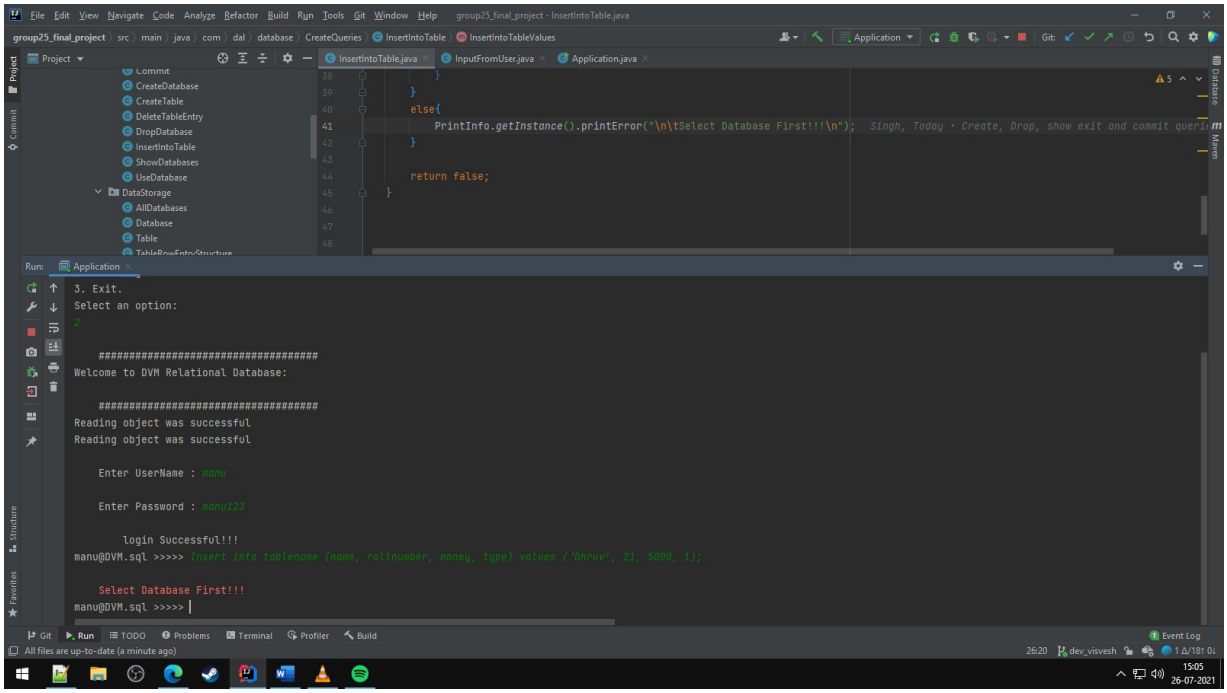

Figure 31. Test Case for not selecting a database before using INSERT query.

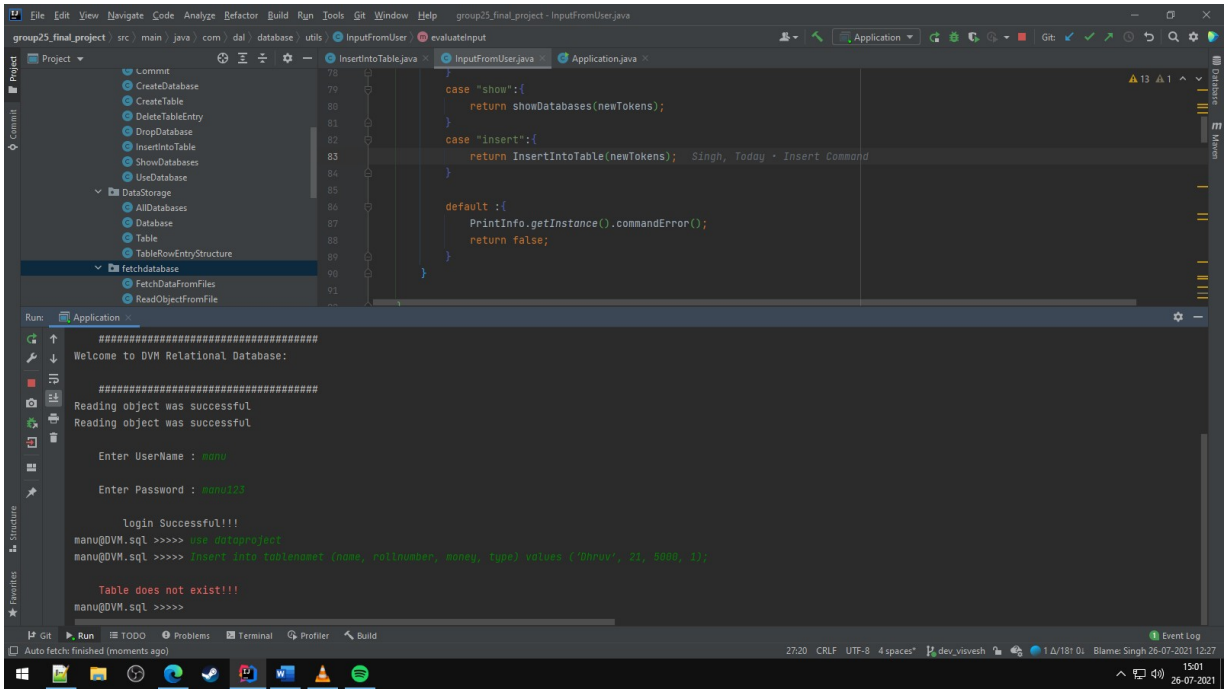

Figure 32. Test Case for INSERT query for a table that does not exist.

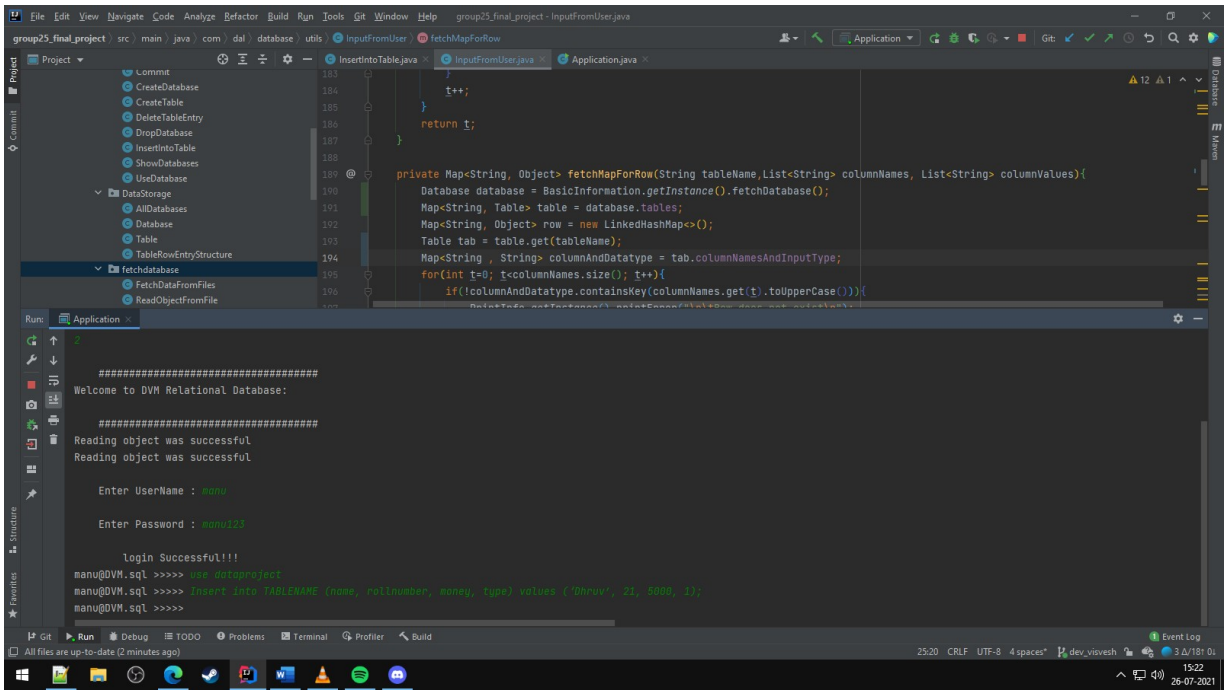

Figure 33. Test Case for a valid INSERT query.

- Now Insert query can use both uppercase or lowercase words.

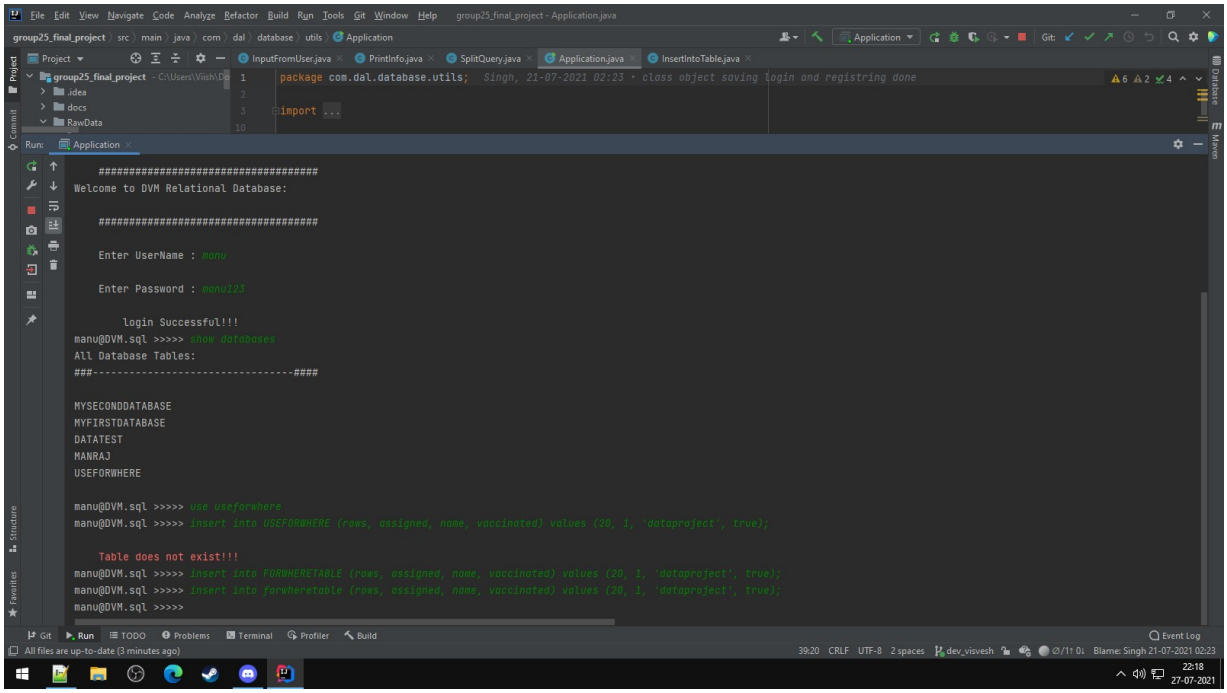

Figure 34. Test Case for a valid INSERT query which is case insensitive.

- If the parameter's data type is different from the value given, it will throw an error – The input parameter 'The_field' is wrong.
- Here assigned is an integer data type, but we enter a Boolean in the values to throw an error.

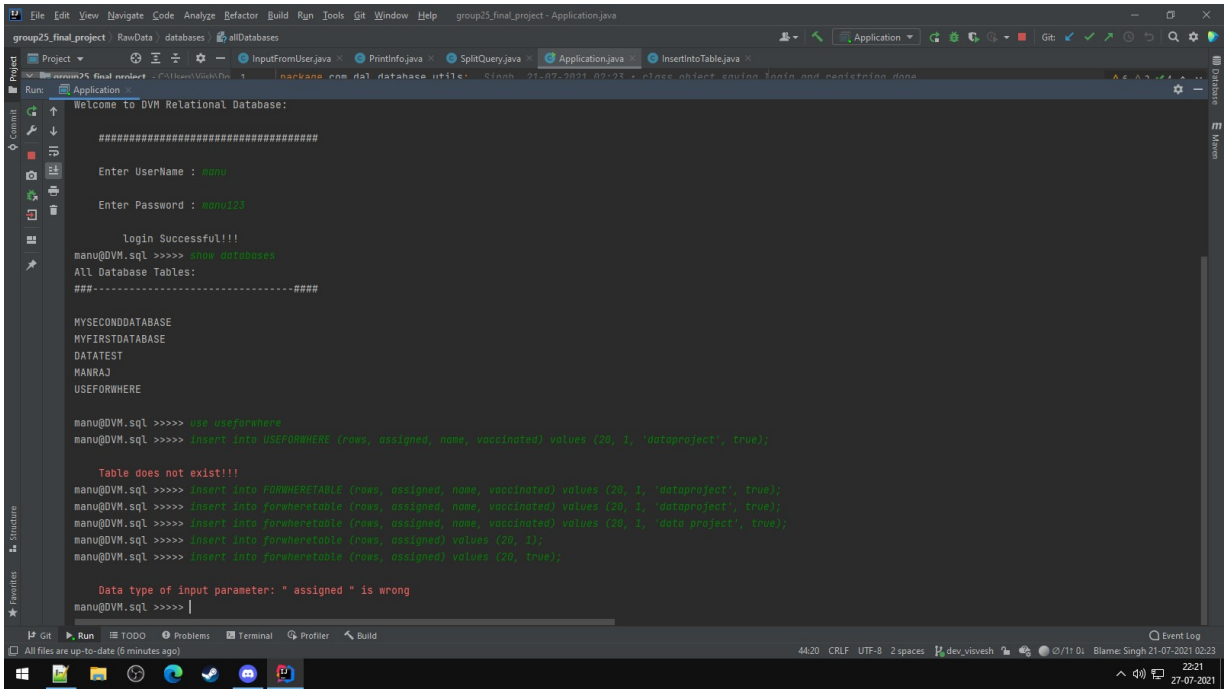

Figure 35. Test Case for an INSERT query with invalid data types.

- Above is the result of the query inserted,
- manu@DVM.sql >>>>> insert into forwheretable (rows, assigned) values (20, 1);
- As you can see, if we don't enter the values for the whole table it will provide them with entry null.

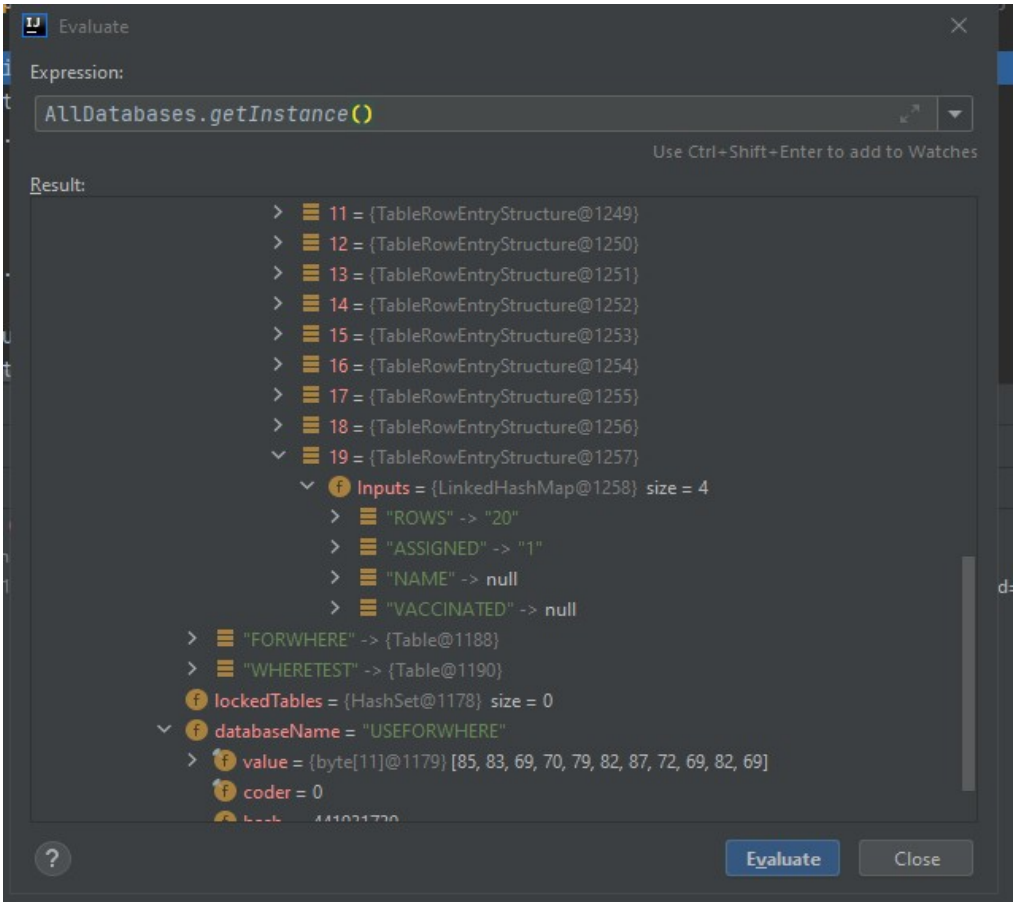

Figure 36. Test Case for a valid INSERT query with default values being filled as NULL.

Select Query:

- Selecting all the rows from the tables.

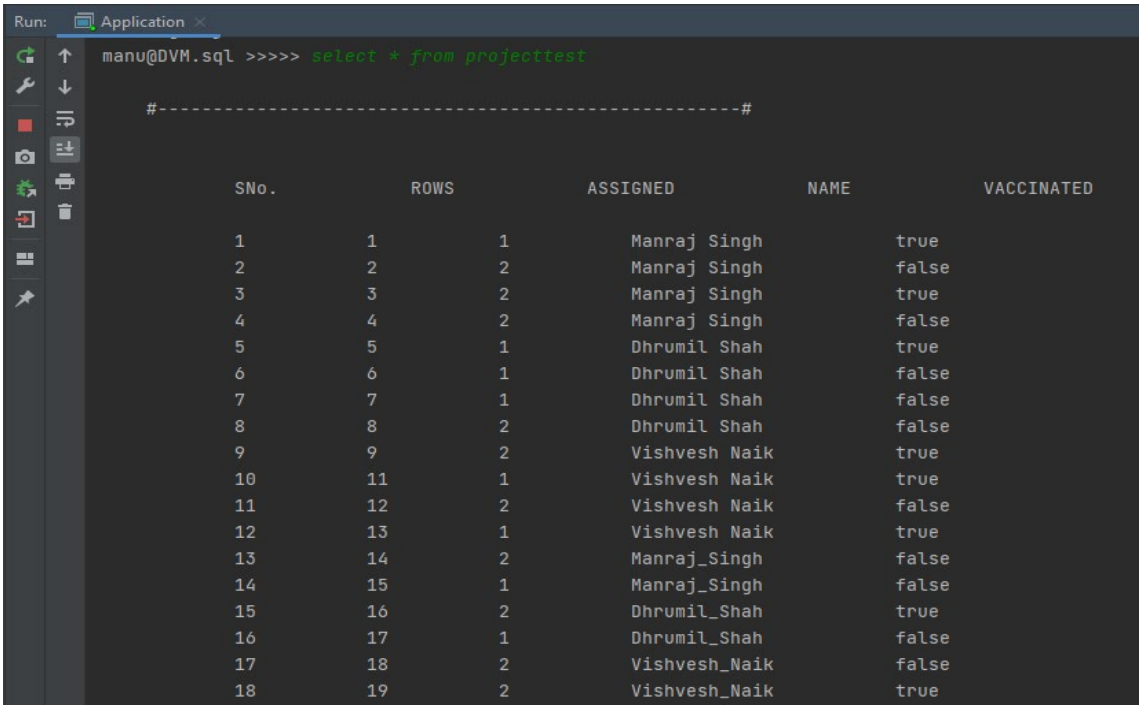

Figure 37. Test Case for a valid SELECT query.

- Select query with where clause selecting only the fifth row.

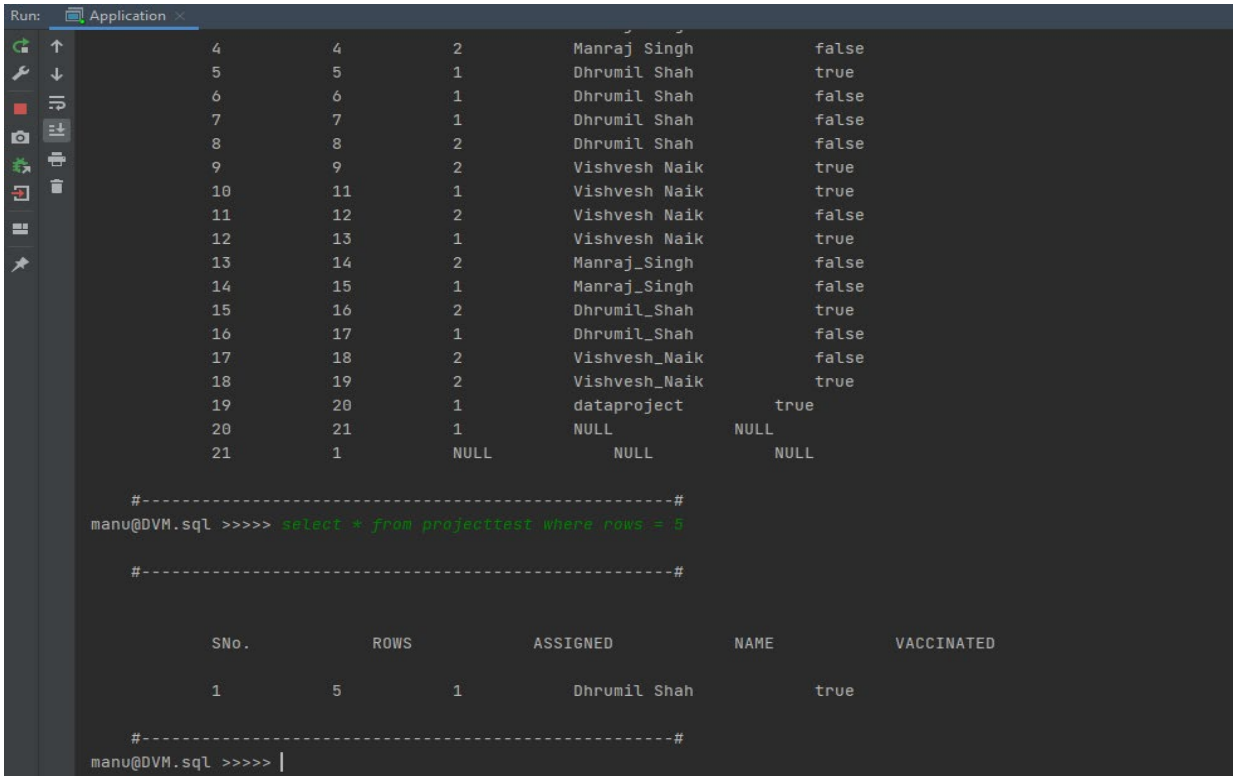

Figure 38. Test Case for a valid SELECT query with a single WHERE clause on an INTEGER field.

- Selecting all rows using where the name is 'Vishvesh Naik.'

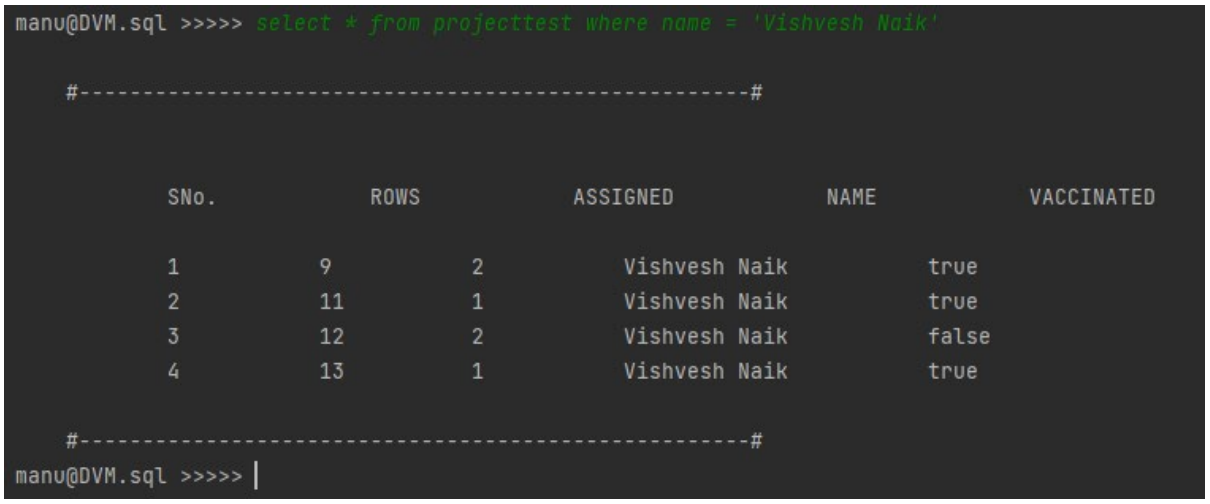

Figure 39. Test Case for a valid SELECT query with a single WHERE clause on a name field.

- Selecting all rows which are not vaccinated.

```
manu@DVM.sql >>>>> select (rows) from projecttest where vaccinated = 'false'

    #-------------------------------------------------#


          SNo.          ROWS

          1          2
          2          4
          3          6
          4          7
          5          8
          6          12
          7          14
          8          15
          9          17
          10         18

    #-------------------------------------------------#
manu@DVM.sql >>>>>
```

Figure 40. Test Case for a valid SELECT query with a single WHERE clause on a BOOLEAN field 1.

- Selecting all rows and names who are not vaccinated.

```
manu@DVM.sql >>>>> select (rows, name) from projecttest where vaccinated = 'false';

    #-------------------------------------------------#


          SNo.          ROWS          NAME

          1          2          Manraj Singh
          2          4          Manraj Singh
          3          6          Dhrumil Shah
          4          7          Dhrumil Shah
          5          8          Dhrumil Shah
          6          12         Vishvesh Naik
          7          14         Manraj_Singh
          8          15         Manraj_Singh
          9          17         Dhrumil_Shah
          10         18         Vishvesh_Naik

    #-------------------------------------------------#
manu@DVM.sql >>>>>
```

Figure 41. Test Case for a valid SELECT query with a single WHERE clause on a BOOLEAN field 2.

Delete Query:

- Delete query deletes the required rows to be deleted. Here the 21 rows is getting deleted from the table.

```
manu@DVM.sql >>>>> delete from projecttest where rows = '21';

    Deleted 1 rows successfully
manu@DVM.sql >>>>>
```

Figure 42. Test Case for a valid DELETE query.

Update Query:

- Update Query will update required the rows as per the need.

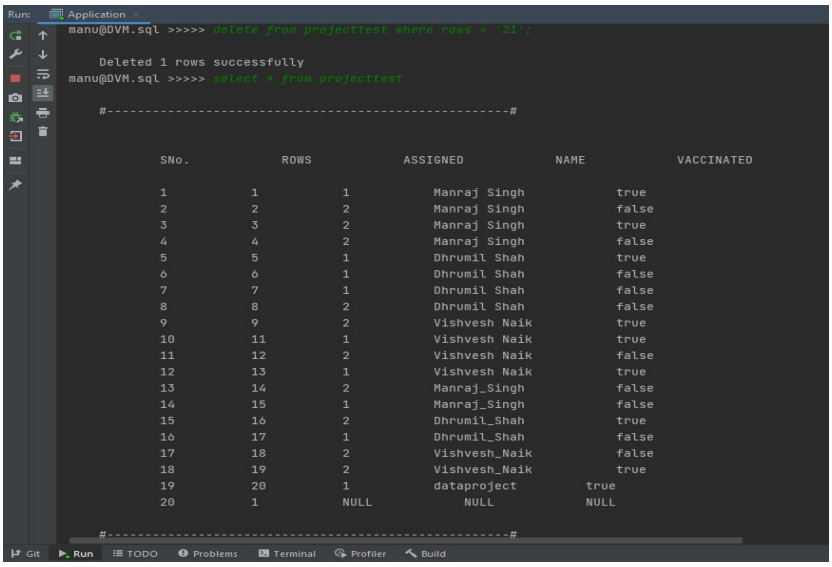

Figure 43. Test Case for a valid UPDATE query 1.

- Here the string of data project is being updated to data_project_test.

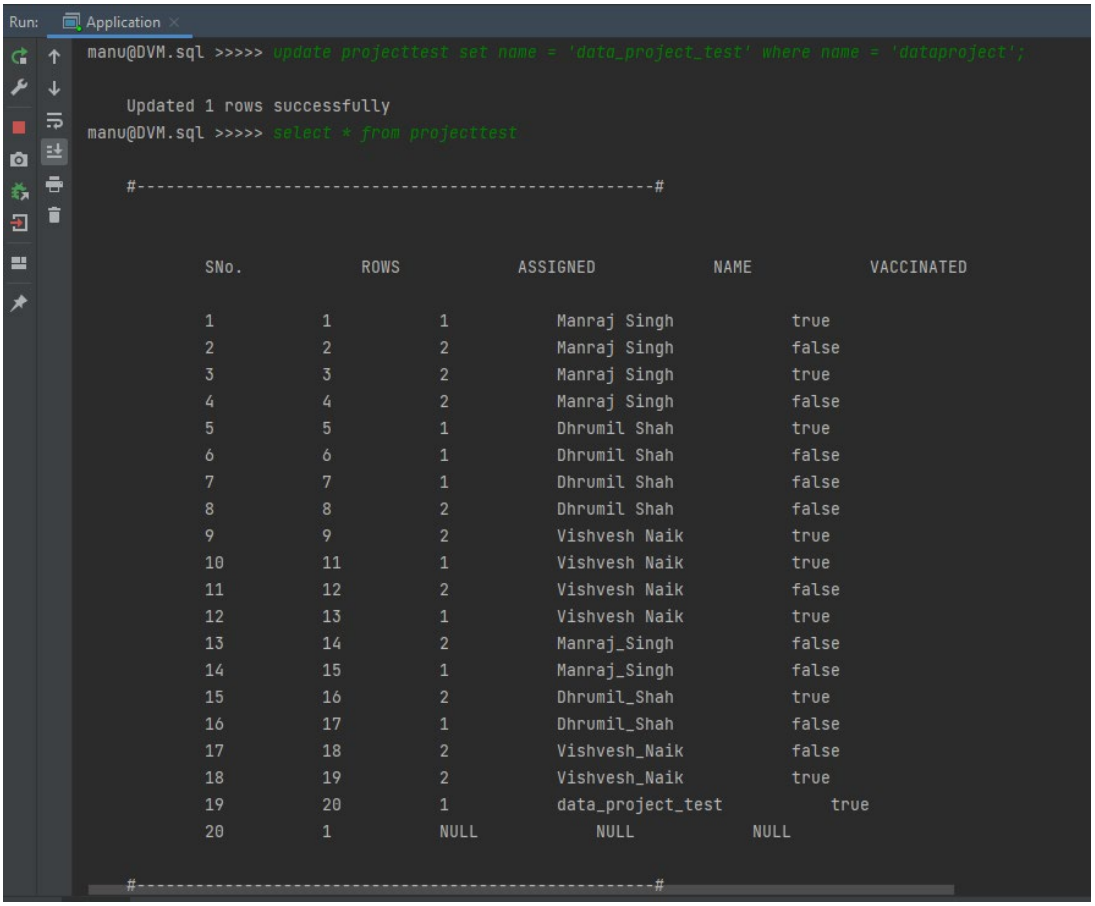

Figure 44. Test Case for a valid UPDATE query 2.

All Tables Query:

- This shows all the tables in the database.

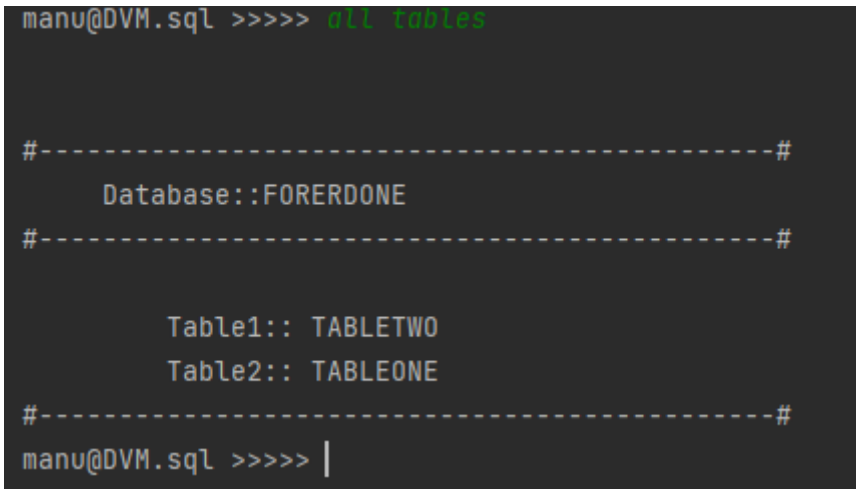

Figure 45. Test Case for a valid ALL TABLES query.

Rollback Query:

- Rollback will undo all the changes just till the last commit.

```
         SNo.          ROWS          ASSIGNED          NAME          VACCINATED

         1          1          1          Manraj Singh          true
         2          2          2          Manraj Singh          false
         3          3          2          Manraj Singh          true

   #-------------------------------------------------#
vishvesh@DVM.sql >>>>> commit

   Commit Successful!
   Object written successfully!!!!
vishvesh@DVM.sql >>>>> insert into projecttest (rows, assigned, name, vaccinated) values (20, 1, 'dataproject', true);
vishvesh@DVM.sql >>>>> rollback

   Rollback Successful!
   Reading object was successful
Reading object was successful
vishvesh@DVM.sql >>>>> select * from projecttest

   #-------------------------------------------------#


         SNo.          ROWS          ASSIGNED          NAME          VACCINATED

         1          1          1          Manraj Singh          true
         2          2          2          Manraj Singh          false
         3          3          2          Manraj Singh          true

   #-------------------------------------------------#
```

Figure 46. Test Case for a valid ROLLBACK command.

Primary Key and Foreign Key:

- Primary key should not be kept empty.
- Primary key cannot be a duplicate entry.
- The query of Primary and Foreign keys is given in the second picture.

```
manu@DVM.sql >>>>> describe newdataset

#-----------------------------------------------#
     Database::FIRST        ----->      TableName:: NEWDATASET
#-----------------------------------------------#

        *Primary Key:       ROWS        ----->      Integer
        *Foreign Key:       ASSIGNED        ----->      Integer
                            NAME        ----->      String
                            VACCINATED      ----->      Boolean
#-----------------------------------------------#
manu@DVM.sql >>>>>
```

Figure 47. Test Case for Primary and Foreign Keys 1.

```
manu@DVM.sql >>>>> create table cityId (id integer, cityName string, PRIMARY id);
Object written successfully!!!!
manu@DVM.sql >>>>> insert into cityId (id, cityName) values (1, 'Ahmedabad');
manu@DVM.sql >>>>> insert into cityId (id, cityName) values (1, 'Halifax');
Primary Key Cannot be duplicate
manu@DVM.sql >>>>>
```

Figure 48. Test Case for Primary and Foreign Keys 2.

```
manu@DVM.sql >>>>> create table name (id integer, name string, cityId int, vaccinated boolean, PRIMARY id, FOREIGN cityId);
Object written successfully!!!!
manu@DVM.sql >>>>> insert into name (id, name, cityId, vaccinated) values (1, 'Dhrumil', 3, true);
manu@DVM.sql >>>>> insert into name (id, name, cityId, vaccinated) values (2, 'Manraj', 2, true);
manu@DVM.sql >>>>> insert into name (id, name, cityId, vaccinated) values (3, 'Vishvesh', 1, true);
manu@DVM.sql >>>>> insert into name (id, name, cityId, vaccinated) values (4, 'Dhruv', 2, true);
manu@DVM.sql >>>>>

    Table name already present
manu@DVM.sql >>>>> create table cityId (id integer, cityName string, PRIMARY id);
Object written successfully!!!!
manu@DVM.sql >>>>> insert into cityId (id, cityName) values (1, 'Ahmedabad');
manu@DVM.sql >>>>> insert into cityId (id, cityName) values (1, 'Halifax');
Primary Key Cannot be duplicate
manu@DVM.sql >>>>> insert into cityId (id, cityName) values (2, 'Halifax');
manu@DVM.sql >>>>> insert into cityId (id, cityName) values (3, 'Mumbai');
manu@DVM.sql >>>>> commit;

    Commit Successful!
    Object written successfully!!!!
```

Figure 49. Test Case for Primary and Foreign Keys 3.

```
manu@DVM.sql >>>>> create table droptest (id int, name string, gender boolean, description string);

    Primary Key Cannot be NULL
```

Figure 50. Test Case for Primary and Foreign Keys 4.

ER Diagram:

- ER Diagram is a visual representation of all the relation in the database only if they are any.

```
#-----------------------------------------------#
     Database::ERDTEST      ----->      TableName:: CITYID
#-----------------------------------------------#

        *Primary Key:       ID      ----->      Integer
                            CITYNAME        ----->      String
#-----------------------------------------------#

#-----------------------------------------------#
     Database::ERDTEST      ----->      TableName:: NAME
#-----------------------------------------------#

        *Primary Key:       ID      ----->      Integer
                            NAME        ----->      String
        *Foreign Key:       CITYID      ----->      Integer
                            VACCINATED      ----->      Boolean
#-----------------------------------------------#
```

Figure 51. Test Case for ER Diagram.

SQL Dump:

- Sqldump save command will save the database when in use; otherwise, it will throw an error. This command will save a dump in our directory and create a separate file in the ./FilesOfData/SQL_Dump.

```
manu@DVM.sql >>>>> show databases
All Database Tables:
###--------------------------------####

NEWDB
COUNTRYERD
ERDTEST
FIRST
DROPTEST

manu@DVM.sql >>>>> use droptest
manu@DVM.sql >>>>> sqldump save
Object written successfully!!!!


        SQL Dump Written Successfully!!!!
```

Figure 52. Test Case for SQL DUMP 1.

```
manu@DVM.sql >>>>> sqldump save

    Select Database First!!!

    Enter Correct Command
manu@DVM.sql >>>>> use first
manu@DVM.sql >>>>> sqldump save
File Written Successfully!!!!


        SQL Dump Written Successfully!!!!
```

Figure 53. Test Case for SQL DUMP 2.

- Sqldump get, this command will fetch all the dumps saved and add it to the directory to our all-database file to read and use from that file.

```
manu@DVM.sql >>>>> sqldump get

#-----------------------------------------------#

    List of Files:


#-----------------------------------------------#

        1 FIRST

#-----------------------------------------------#


    Enter the index of file to add: 1
Reading data was successful
File Written Successfully!!!!


        SQL Dump Read Successfully!!!!

manu@DVM.sql >>>>>
```

Figure 54. Test Case for SQL DUMP 3.

Persistent Storage:

- This is the example of the custom storage file format, in this file, we are saving the dump are in the text file with all the database name, table, attributes and their values.

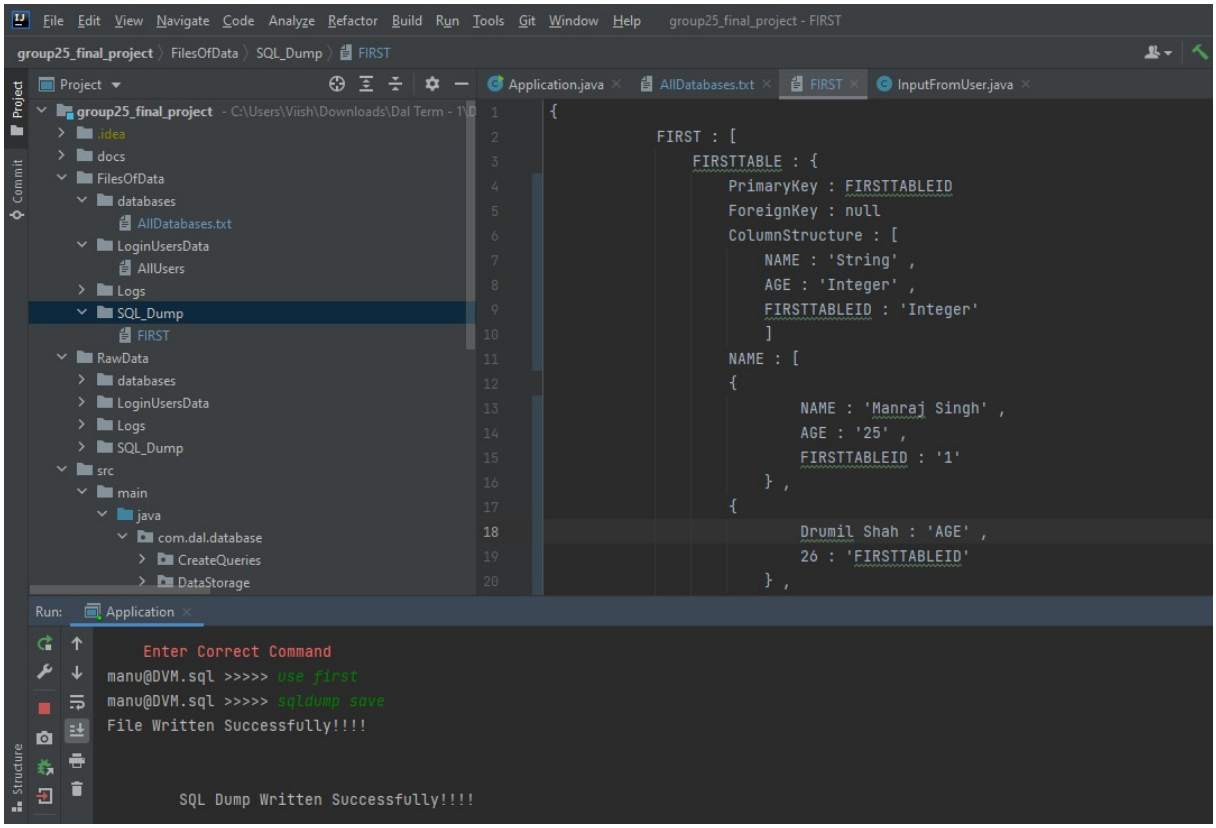

Figure 55. Test Case for Persistent Storage of the data 1.

- AllDatabases.txt is the main file where we write and read everything.

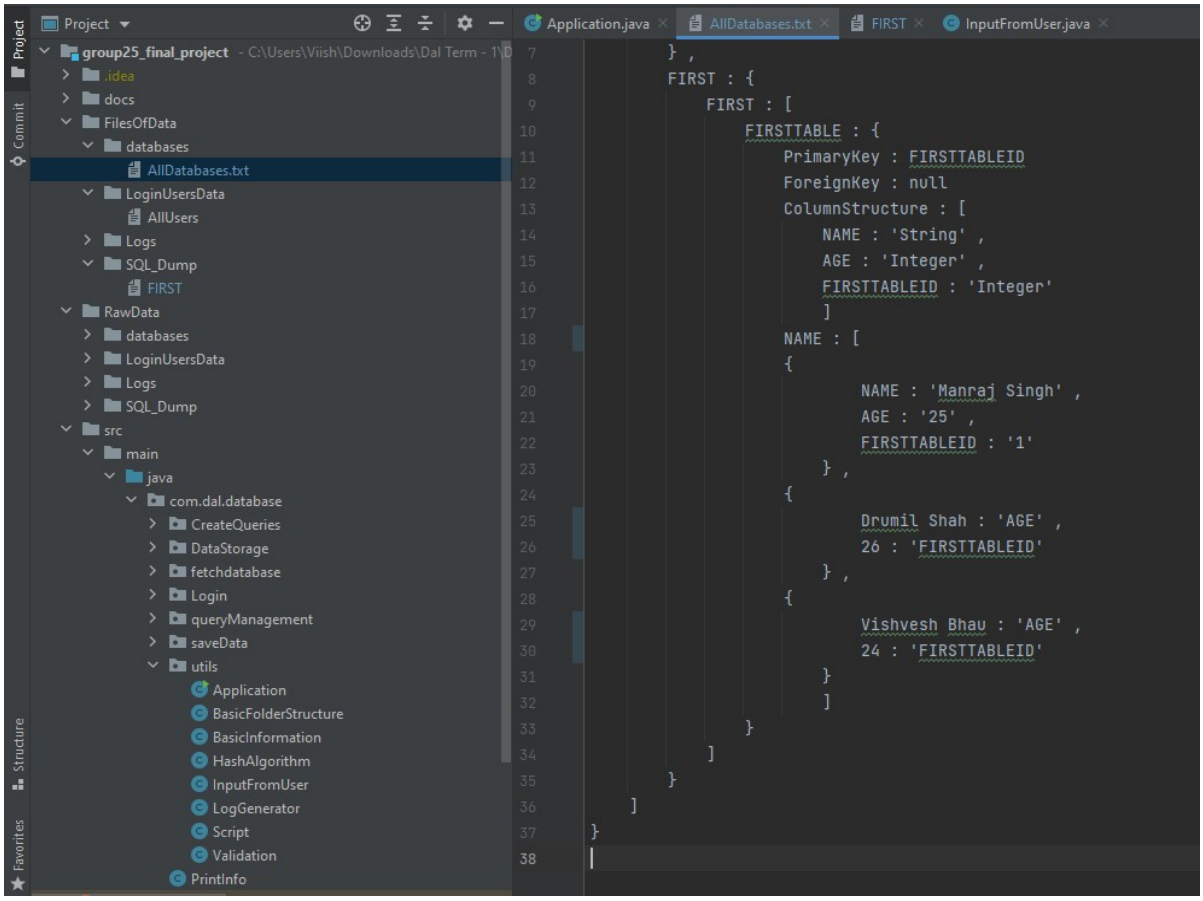

Figure 56. Test Case for Persistent Storage of the data 2.

Log Files:

Query Logs:

- This file contains the query related stuff, what query we've executed from the beginning of the application.

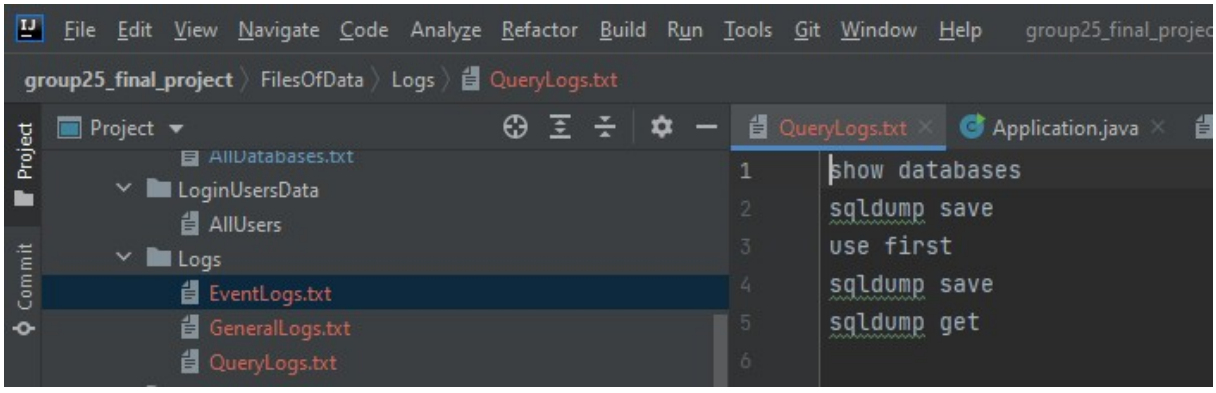

Figure 57. Test Case for Query Logs being generated in a QueryLogs.txt file.

General Log:

- This file contains all the general logs, it logs persist of our entered query and the returned values by the RDBMS.

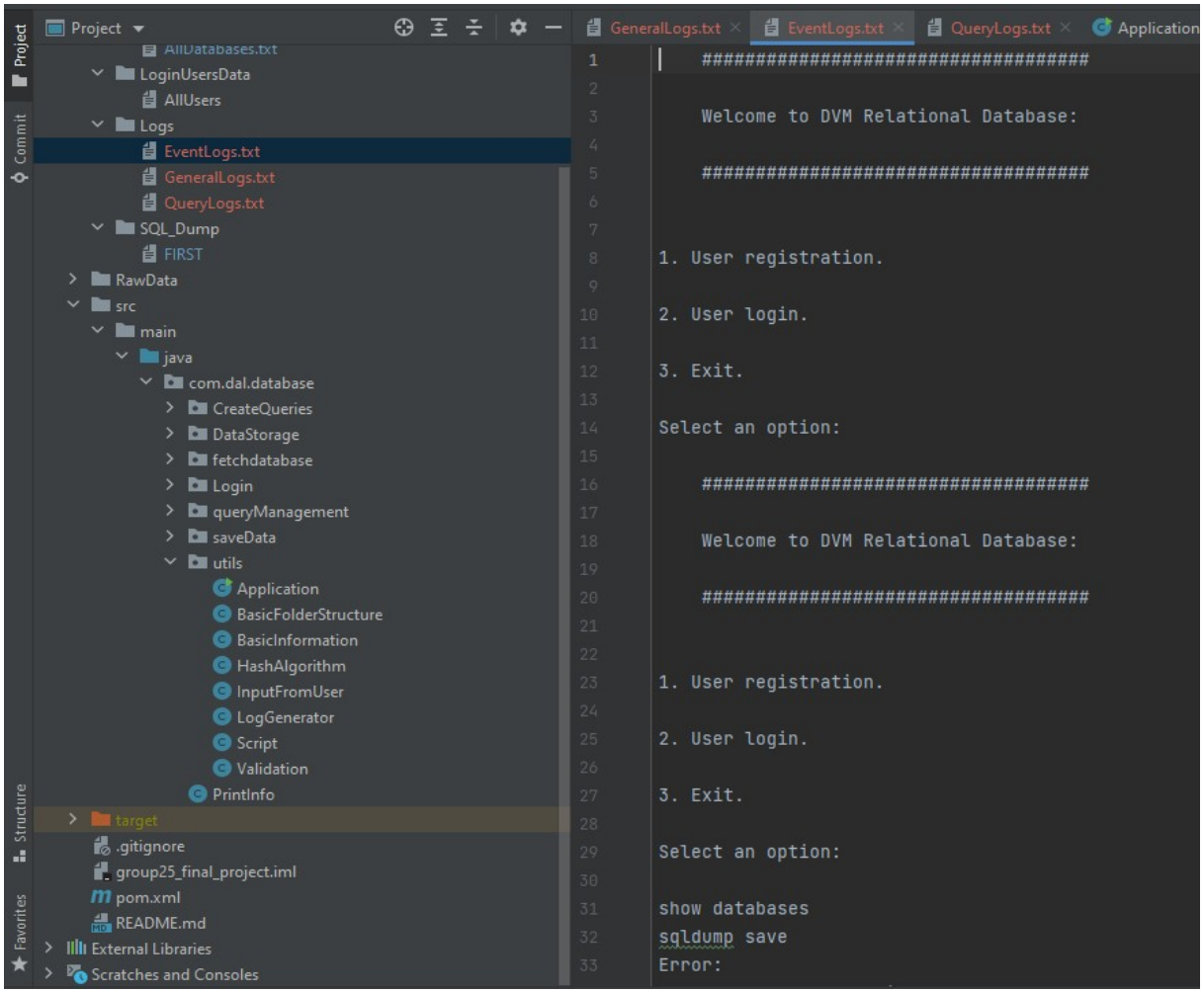

Figure 58. Test Case for General Logs being generated in a GeneralLogs.txt file.

Event Log:

- In this type of log, the only difference is that this file will also contain the errors that are returned by the RDBMS.

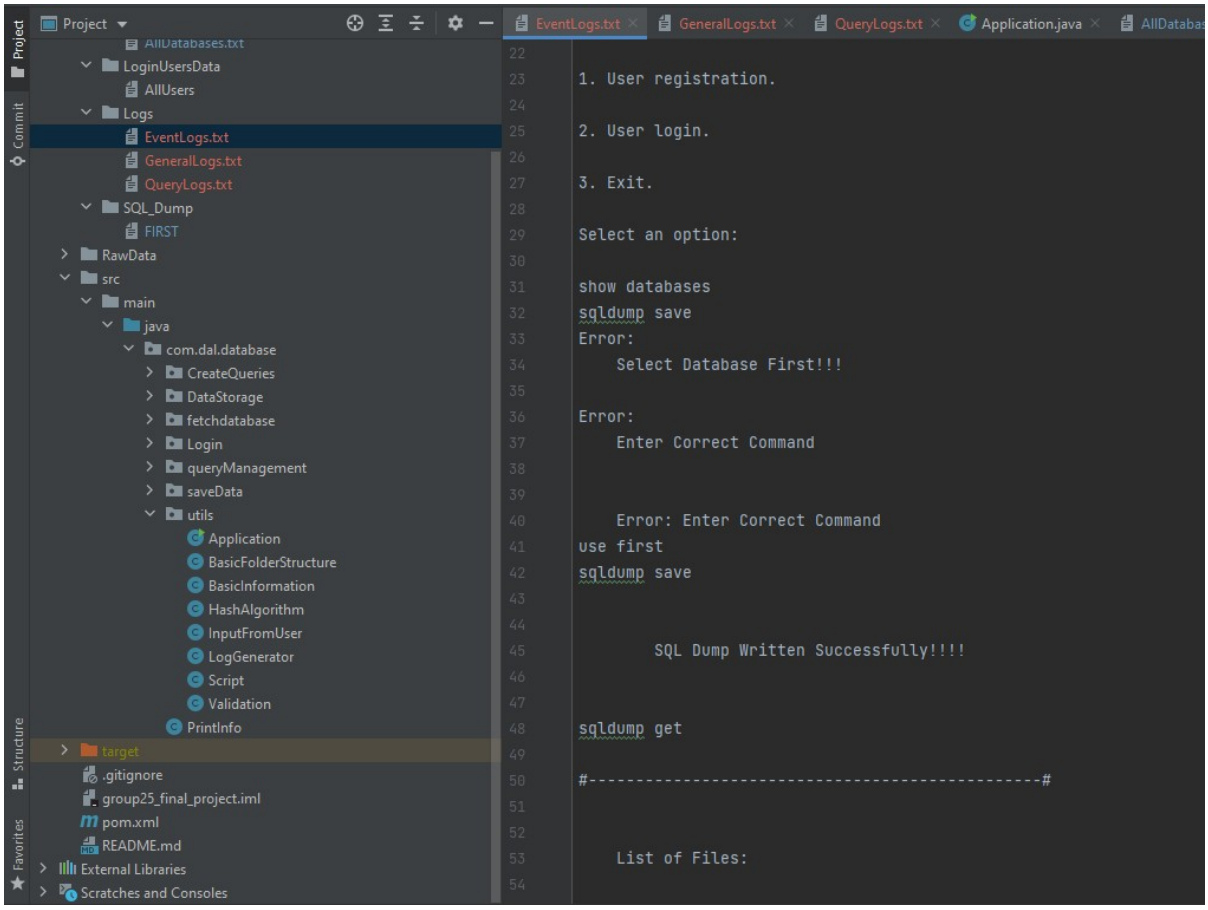

Figure 59. Test Case for Event Logs being generated in an EventLogs.txt file.

Limitations:

- The RDBMS created will only work in our environment locally, not in any other system.
- The system cannot handle multiple concurrencies at a single time. Moreover, the transactions will not interact with multiple tables at a time.
- No alias will be used in the query.
- The Queries will not have a GROUP BY clause.
- The nested queries or the subqueries will not work in our system. For example,

```
SELECT *
  FROM CUSTOMERS
  WHERE ID IN (SELECT ID
      FROM CUSTOMERS
      WHERE SALARY > 4500);
```

- The SQL dump that we are generating will only get exported and imported into our system, not in other software like SQL Workbench. This is because we had to create our own parser for the persistence storage.
- Similarly, the ERD is not generating a perfect structure or visually preferable image like SQL Workbench.

Future Scope:

In the future, we would like to rectify all these static methods that we are working with and make them completely dynamic. The system should be more user-friendly—the addition of SQL dump working in any environment and within SQL workbench. The ERD will be more visually friendly rather than only showing the relations of the tables and table names.

References:

[1] L. Duggan, "Welcome to the DevOps Platform era," GitLab, 2021. [Online]. Available: https://about.gitlab.com/.

[2] JetBrains, "IntelliJ IDEA: The Capable & Ergonomic Java IDE by JetBrains," JET BRAINS, 2000. [Online]. Available: https://www.jetbrains.com/idea/.