CS1003 Assignment 2: Databases

Overview:

In this assignment I was asked to create a Relational Database System (Which adopts SQLite as its querying language and database engine). To populate this database and also retrieve data from it, we were also tasked with the usage of the JDBC Interface, which would connect directly to the database to carry out the aforementioned operations.

The database is built around **Movies**, which contain characteristics such as "a title, a release date, running time, genre, the director, actors, plot, some ratings (like Rotten Tomatoes, IMDB etc.) and the awards they have won. " It will also use **Actors** as its main building pillar, which will contain attributes such as **"**a name, the awards they have won, what films they have been in, their birthday etc**.**

This practical was divided in 5 stages:

1. **The Design of an ER Diagram:** Used to visually represent the tables and relationships that form our databases.
2. **The Creation of the Relational Schema:** Which is used to add the constraints and indications about the different columns and keys that form the tables. Also establishes how the data is structured. This was stored in a .sql File in the form of a script.
3. **Creation of a Java Program to Create the Relational Schema:** By using a JDBC Connection and the already created Relational Schema File, the program would instantiate a database with the constraints that are in the relational schema file.
4. **Population of the Database:** To populate the database, I was asked to read through a series of files (which were created by me) and add entries to the database using information directly extracted from the files.
5. **Create a Java Program to Query:** The JDBC Connection would allow for the execution of a series of SQL Queries in order to retrieve data from the table.

My program is able to successfully complete all 5 tasks. This submission has an ER Diagram which accurately represents my database. Additionally, it contains a database that complies with the given specification and allows for its users to retrieve data from it. Finally, it also handles all of the related exceptions and problems that could occur during its usage, such as failure to create the database, failure to retrieve data from a table (because the query does not give any data back) or the usage of unrecognised files that are not compliant with my code, just to mention some examples.

Design:

Through the creation of my ER Diagram, I realised that there were several **many-to-many** relationships, which would make me create several tables that would work as intermediaries between the main tables. These tables are **Cast (**Actors and Movies)**,**

MoviesWithRatings, AwardWinningActor, AwardWinningDirector and AwardWinningMovie, which as their name might explain, contain the unique primary keys of each one of the tables which it mentions. This allows me to easily query for information, since I can simply use INNER JOINS and SQL Code to directly retrieve what I need and accurately depict real-life scenarios, such as an actor being in multiple movies, a director being in several movies and winning several awards, etc

Another one of the relevant decisions of my software was the usage of JSON Files for the storage of the information that populates the database. Although I believe that the same could have been achieved with other formats (Mainly CSV as an alternative), the limitations imposed by the lack of libraries led me to use this format, as I believe it was easier to parse and store data. Also, the fact that I have previously worked with this file type, influenced this decision.

One of my greatest concerns was the checking of the Relational Schema, which is the most important part of the code, as it is the foundation of everything that comes after. To eliminate this issue, I decided to implement a DDLTester class, which contains an ArrayList of Strings, which contain the valid relational schema commands. This is used during runtime, when it directly compares this to a list which is formed from the DDLScript file (or whichever file is designated as the DDLScript). If any of the commands do not match or the order is not met, then the operation is cancelled. By enforcing this, it makes it impossible to bypass the schema validation and thus, avoids all possible problems that could arise.

In order to accurately check the schemas of the JSON Files, my program uses streams to turn the correct documents into a list of lines. When doing the parsing, the code compares the file directly to this list in order to make sure that the inputted files **perfectly match** (in both content and order). That was the only way that I could thought of to actually test out the files that are going to be parsed. Ideally, this would have been achieved using custom schemas and libraries who supported a comparison between file schemas or a validation system that could allow to tell the difference between 2 JSON Files

Through the usage of streams and loops, every individual JSON File is turned into an array of objects. My parser essentially is composed of 4 methods depending on the datatype of the value in the key-value pair and by applying a series of methods on a **BufferedReader** stream, it stores every single value of a specified key into a list, which is then used to create the corresponding object. This process allows for a very easy access of each one of the eventual columns of the table, which in turn greatly simplifies the final process of population, as it is simply extracting and using the values of objects through looping. The parser essentially is composed of 4 methods depending on the datatype of the value in the key-value pair.

I would also like to highlight how the population system works on my program. By using a series of advanced for loops, prepared statements and standard accessor methods of each one of the necessary classes, the population of the tables is a straightforward process. Additionally, I also decided to initialise the intermediate tables directly with their main tables as well. This was done in order to reduce the number of calls done to the parser, as it directly uses the array that filled the main table as many times as intermediate tables require population. This increases performance and efficiency.

Sourced Code or Files:

I had to source code from certain websites in order to be able to achieve certain objectives. Although it is explained in the appropriate method documentation, I will also explain it here:

- **Testing of Printing**: Provided by Baeldung(2024), this allowed me to test the code that was being printed to the console.
- **Checking if a Result Set was Empty**: I wanted to do this, but I was unaware if there was a way to do so without printing. Thankfully, StackOverflow (2010) provided a way for this.
- **JDBC Driver Load**: This was sourced by the lecturers in the JDBC Examples of Week 6
- **Files**: It has to be mentioned that all of the additional files that are used in the code were created by me. This includes all of the JSON Files and the SQL Scripts that my code uses. As for the information in the JSON Files, I was found by using Google and researching in websites such as IMDb and Wikipedia.
- The rest of the code was created by me (unless it is claimed otherwise in the documentation)

Testing:

The main objective of my testing was to ensure that the conditions were adequate for the parsing process and the population process, while also ensuring that all of the necessary elements (The JSON Files and the SQL Scripts) are compliant with the schemas that I have elaborated. This approach makes the testing of the elements inside of the database, such as the correct datatypes, constraints and foreign key usage unnecessary, as the code performs perfectly when the conditions are given. Nonetheless, all of the processes were tested accordingly in order to ensure that the programs are up to standard.

In order to appropriately test all of the different case scenarios, this submission includes JUnit tests that aim to demonstrate the solidity of my code, as well as different other tests that although I could not run through JUnit, were essential in the pursuit of the improvement of my codebase. These tests are:

What is being tested	Name of the test method	Pre-conditions	Expected Outcome	Actual Outcome	Evidence
Normal creation of the Database with its constraints	Done by the initialiseDB() method. Tested by the test1() method of the DatabaseTests JUnit Class	Deletion of the database file.	Creation of the Database with its correct constrains.	Tables are correctly initialised	The passing score of test1() of my JUnit Class Please check figure 1 for the Diagram of the IntelliJ Database Plugin
Passing of non-existent file into as the SQL Script for the initialiseDBMethod	Done by the initialiseDB() method. Tested by the test2() method of the DatabaseTests	The passed file does not exist	Prints out appropriate message	"The DDL Script cannot be read, or it cannot be detected. Please check the DDL Script"	Passing score of the test2() method of my JUnit Class
Supplying a random existent SQL File to test the DDL Tester	Controlled by the test3() method of the JUnit Class	The creation and passing of a SQL File that does not contain anything	Prints out the appropriate message	"The DDL Script schema has not been validated. Therefore, the table was not created"	Passing score of the test3() method of the JUnit Class
Tests the JSON Tester for the Actors JSON validation	Controlled by the test4() method of the JUnit Class	The creation and passing of an empty file	Expects the JSONSchemaException	The exception is correctly thrown	Passing score of the test4() method of the JUnit Class
Tests the JSON Tester for the Awards JSON validation	Controlled by the test7() method of the JUnit Class	The creation and passing of an empty file	Expects the JSONSchemaException	The exception is correctly thrown	Passing score of the test7() method of the JUnit Class
Tests the JSON Tester for the Movies JSON validation	Controlled by the test8() method of the JUnit Class	The creation and passing of an empty file	Expects the JSONSchemaException	The exception is correctly thrown	Passing score of the test8() method of the JUnit Class
Tests the JSON Tester for the Directors JSON validation	Controlled by the test9() method of the JUnit Class	The creation and passing of an empty file	Expects the JSONSchemaException	The exception is correctly thrown	Passing score of the test9() method of the JUnit Class
Tests the JSON Tester for the Ratings JSON validation	Controlled by the test6() method of the JUnit Class	The creation and passing of an empty file	Expects the JSONSchemaException	The exception is correctly thrown	Passing score of the test6() method of the JUnit Class

Tests the JSON Tester for the Genre JSON validation	Controlled by the test10() method of the JUnit Class	The creation and passing of an empty file	Expects the JSONSchemaException	The exception is correctly thrown	Passing score of the test10() method of the JUnit Class
Printing of correct message if the SQL Query returns no values	Controlled by test11() of the JUnit Class	Creation of a query which will returns no values, and this query will be passed to the printer method	Expects the appropriate message	“Based on the inputted data, there are no matches. Please try again”	Passing score of the test11() method of the JUnit Class
Analyses what would happen if you passed numbers in string parameters of the 2nd query. Also, it tests for a query with no retrieval at the same time	Controlled by test12() of the JUnit Class	Creation of the tables and its population followed by executing Query 2 by passing a random number (as a string) as the parameter	Expects the appropriate message	“Based on the inputted data, there are no matches. Please try again”	Passing score of the test12() method of the JUnit Class
Tests for wrongful values as string parameters in the third query. Also tests for no retrieval at the same time	Controlled by test13() of the JUnit Class	Creation of the tables and its population followed by executing Query 3 by passing a random number (as a string) as the parameter	Expects the appropriate message	“Based on the inputted data, there are no matches. Please try again”	Passing score of the test13() method of the JUnit Class
Tests out if the fourth query prints out the correct message if there are no matches	Controlled by test14() of the JUnit Class	Creation of the tables and its population followed by executing Query 4 by passing a random string as the parameter	Expects the appropriate message	“Based on the inputted data, there are no matches. Please try again”	Passing score of the test14() method of the JUnit Class
Tests out if the fifth query prints out the correct message if there are no matches	Controlled by test15() of the JUnit Class	Creation of the tables and its population followed by executing Query 5 by passing a random a string and random integer as the parameters	Expects the appropriate message	“Based on the inputted data, there are no matches. Please try again”	Passing score of the test15() method of the JUnit Class
Evaluates what happens in you pass null values	Controlled by test16() of the JUnit Class	Creation of the tables and its population followed by executing Query	Expects the appropriate message	“Based on the inputted data, there are no matches.	Passing score of the test16() method JUnit Class

into the querying() method on query 5		5 by passing an array that has only the first value initialised, while the rest are null		Please try again"	
Evaluate what happens when you pass an array of random length in querying method when args[0] = 2	Controlled by test17() of the JUnit Class	Creation of the tables and its population followed by calling the querying() method and introducing the aforementioned array	Expects the appropriate message	"Incorrect number of arguments"	Passing score of the test17() method of the JUnit Class
Evaluate what happens when you pass an array of a random length in querying method when args[0] = 3	Controlled by test18() of the JUnit Class	Creation of the tables and its population followed by calling the querying() method and introducing the aforementioned array	Expects the appropriate message	"Incorrect number of arguments"	Passing score of the test18() method of the JUnit class
Evaluate what happens when you pass an array random length in querying method when args[0] = 4	Controlled by test19() of the JUnit Class	Creation of the tables and its population followed by calling the querying() method and introducing the aforementioned array	Expects the appropriate message	"Incorrect number of arguments"	Passing score of the test19() method of the JUnit class
Evaluate what happens when you pass an array random length in querying method when args[0] = 5	Controlled by test20() of the JUnit Class	Creation of the tables and its population followed by calling the querying() method and introducing the aforementioned array	Expects the appropriate message	"Incorrect number of arguments"	Passing score of the test20() method of the JUnit class
Evaluate what happens when you pass an array random length in querying method when args[0] = 6	Controlled by test21() of the JUnit Class	Creation of the tables and its population followed by calling the querying() method and introducing the	Expects the appropriate message	"Incorrect number of arguments"	Passing score of the test21() method of the JUnit class

		aforementioned array			
Evaluate what happens when you pass an array random length in querying method when args[0] = 1	Controlled by test22() of the JUnit Class	Creation of the tables and its population followed by calling the querying() method and introducing the aforementioned array	Expects the appropriate message	"Incorrect number of arguments"	Passing score of the test22() method of the JUnit class
Evaluate what happens when you pass null values in a query that requires a string	Controlled by test 24() of the JUnit class	Creation of the database tables and its population with an array with a length 3 in which the first argument is equal to 2 and the second argument is null	Expects the appropriate message	"Based on the inputted data, there are no matches. Please try again."	Passing score of the test24() of the JUnit Class
Makes sure that it throws the correct exception if the first parameter of the args array is a string integer value	Controlled by test25() of the JUnit Class	Creation of the database tables and its population with an array of length 2 with its first argument is equal to a random string integer value	Expects the correct exception. In this case the NumberFormatException	The appearance of the exception, as expected	Passing score of the test25() of the JUnit Class
Makes sure that it throws the correct exception if the first parameter of the args array is a double value	Controlled by test26() of the JUnit Class	Creation of the database tables and its population with an array of length 2 with its first argument is equal to a random double value	Expects the correct exception, which in this case is the NumberFormatException	The appearance of the exception, as expected	Passing score of the test26() of the JUnit Class
Makes sure that the database is actually being cleared using the deleted Method	Controlled by Test27() of the JUnit Class	Creation of the database and then immediate calling of the deleteDB() method	Boolean true value in the assertTrue method.	The test passes, which means assertTrue() has a true boolean value	Passing score of the test27() of the JUnit Class Please check Figure 2 for further testing evidence.
Avoids querying when the table has not been populated	Controlled by test28() of the JUnit Class	Creation of the database and of an array that has length 2 and its	Expects the correct printing of the message	"The table has not been populated.	Passing score of the test 28()

		first value is 2 and its second value is "The Matrix (1999)". After that, it launches the querying method with the array as a parameter		Please try again".	of the JUnit Class
Makes sure that the table cannot get populated more than once	Controlled by test29 of the JUnit Class	Creation of the database and its population. The population() method is run twice for the purpose of this test	Expects the correct printing of the message	"The table was already populated. Nothing happened"	Passing score of test29() of the JUnit Class
Tests for appropriate table population.	Controlled by test30 of the JUnit Class	Creation of the database and its population and then calling of the emptyDBChecker method, who ensures that the table is not empty	Expects false from the AssertFalse method in the test	The test passes, which means that the method produced a false Boolean value	Passing score of test30() of the JUnit Class Please check figures for the IntelliJ Database Plugin Screenshot with extra information

Attached below is further evidence of testing:

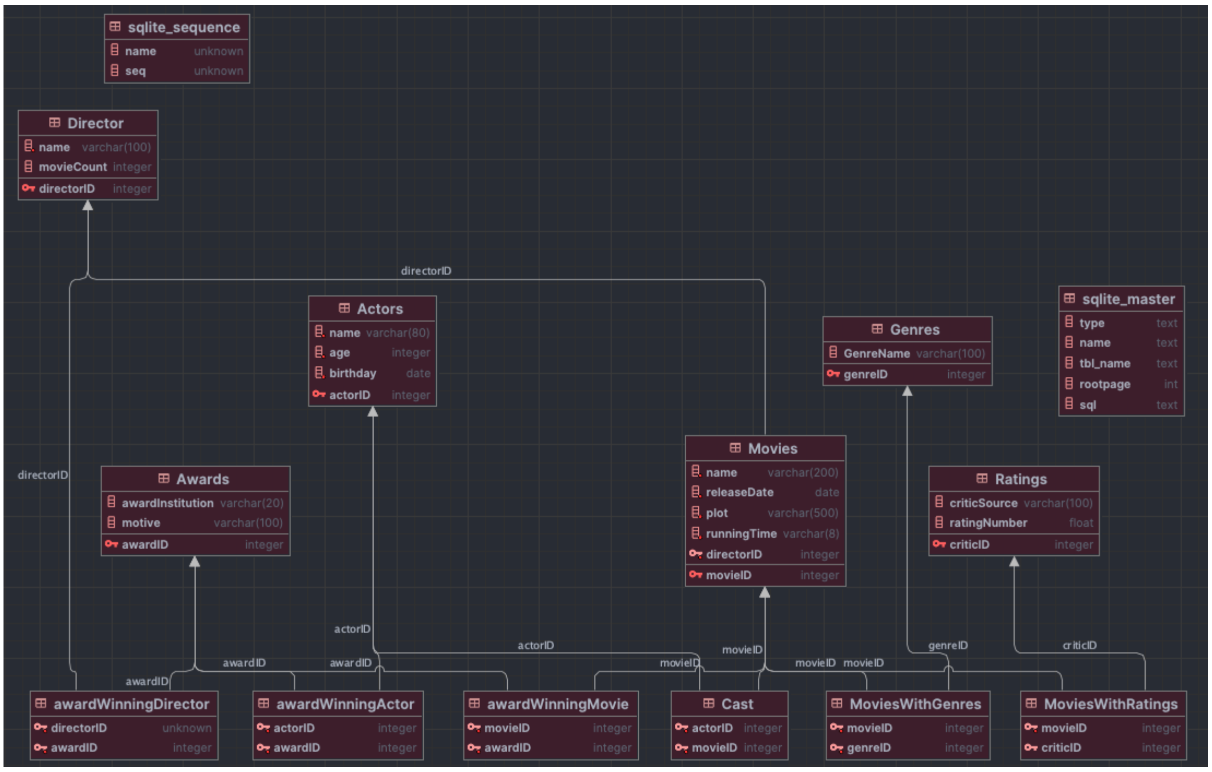


Figure 1: Demonstrates the correct functioning of the initialiseDB() method.

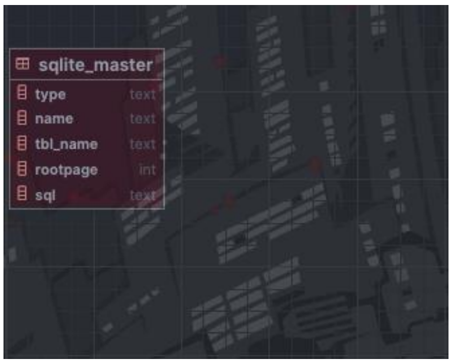
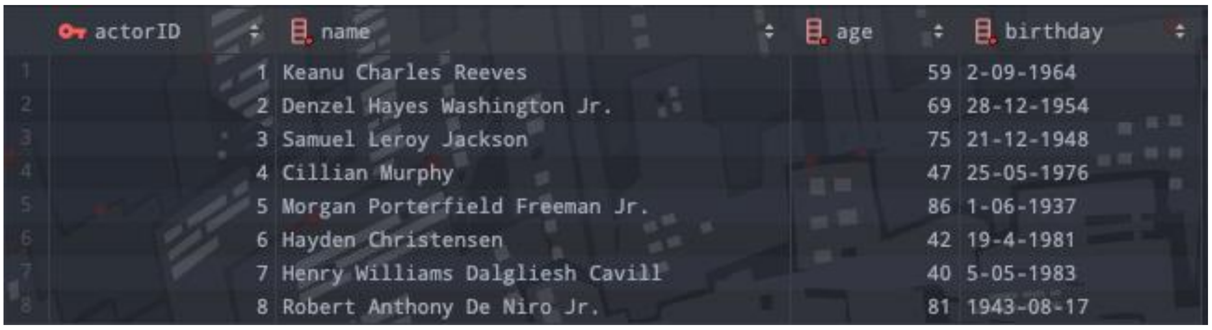
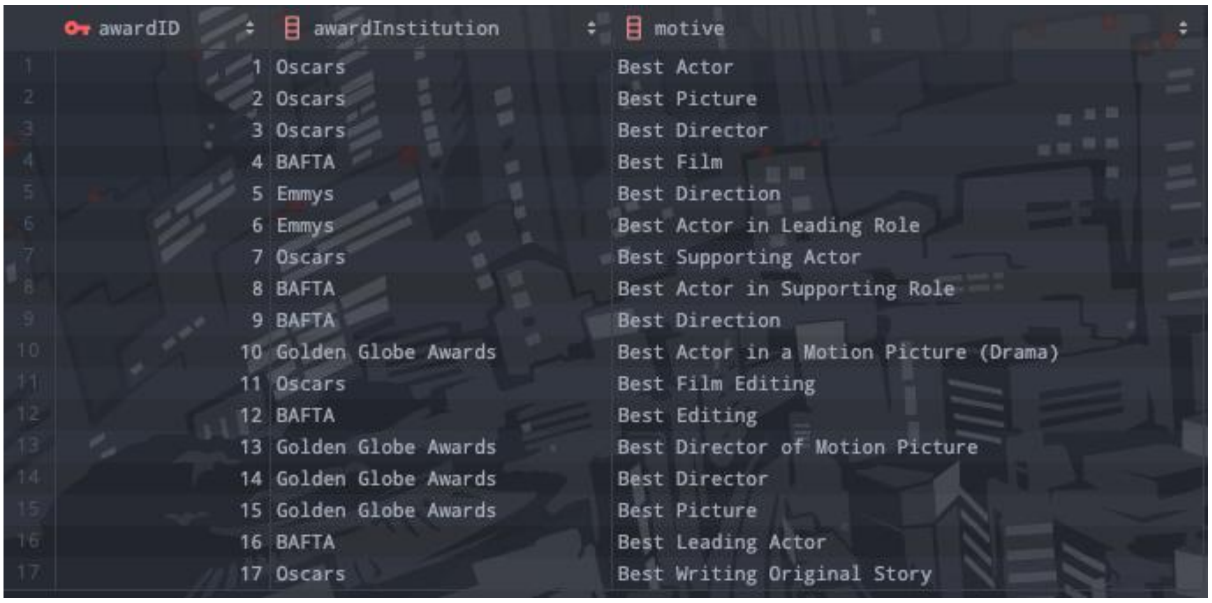


Figure 2: Database after running the deleteDB() method.



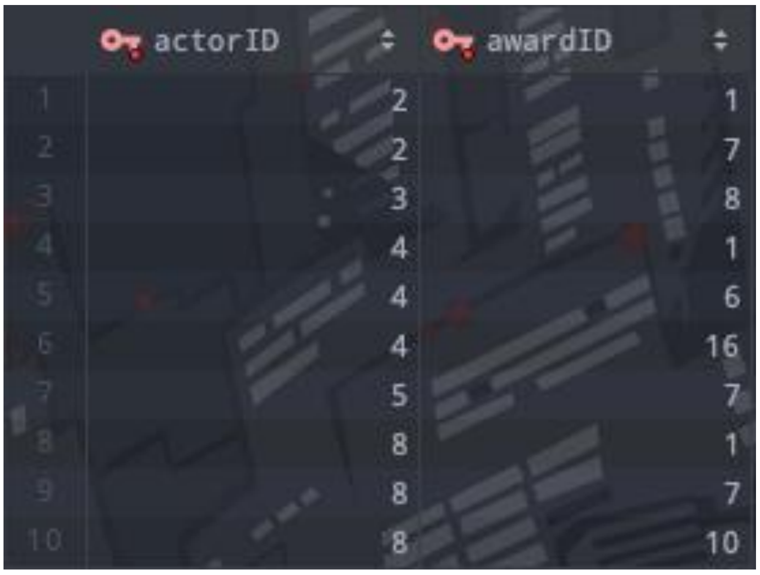
	actorID	name	age	birthday
1	1	Keanu Charles Reeves	59	2-09-1964
2	2	Denzel Hayes Washington Jr.	69	28-12-1954
3	3	Samuel Leroy Jackson	75	21-12-1948
4	4	Cillian Murphy	47	25-05-1976
5	5	Morgan Porterfield Freeman Jr.	86	1-06-1937
6	6	Hayden Christensen	42	19-4-1981
7	7	Henry Williams Dalgliesh Cavill	40	5-05-1983
8	8	Robert Anthony De Niro Jr.	81	1943-08-17

Figure 3: Actors table after running the population() method.



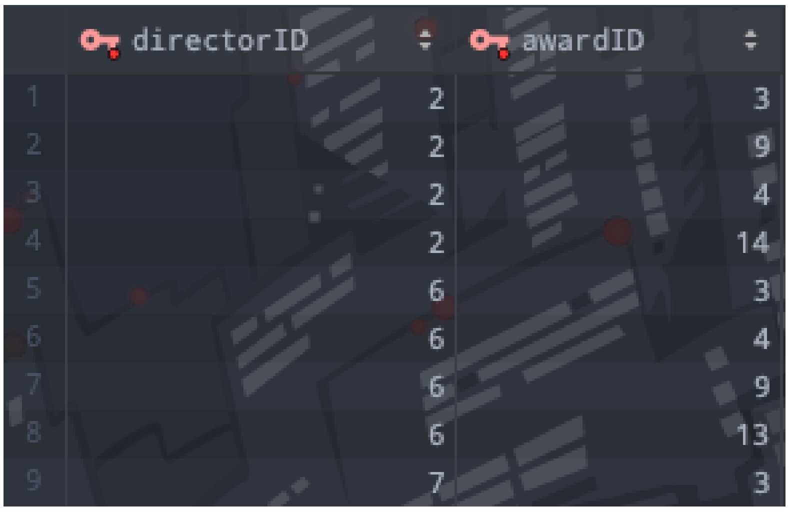
awardID	awardInstitution	motive
1	Oscars	Best Actor
2	Oscars	Best Picture
3	Oscars	Best Director
4	BAFTA	Best Film
5	Emmys	Best Direction
6	Emmys	Best Actor in Leading Role
7	Oscars	Best Supporting Actor
8	BAFTA	Best Actor in Supporting Role
9	BAFTA	Best Direction
10	Golden Globe Awards	Best Actor in a Motion Picture (Drama)
11	Oscars	Best Film Editing
12	BAFTA	Best Editing
13	Golden Globe Awards	Best Director of Motion Picture
14	Golden Globe Awards	Best Director
15	Golden Globe Awards	Best Picture
16	BAFTA	Best Leading Actor
17	Oscars	Best Writing Original Story

Figure 4 Awards table after running the populationDB method.



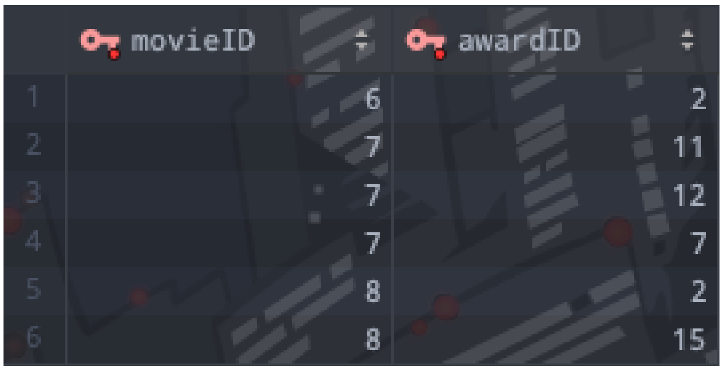
actorID	awardID
2	1
2	7
3	8
4	1
4	6
4	16
5	7
8	1
8	7
8	10

Figure 5 AwardWinningActor table after running the populationDB method.



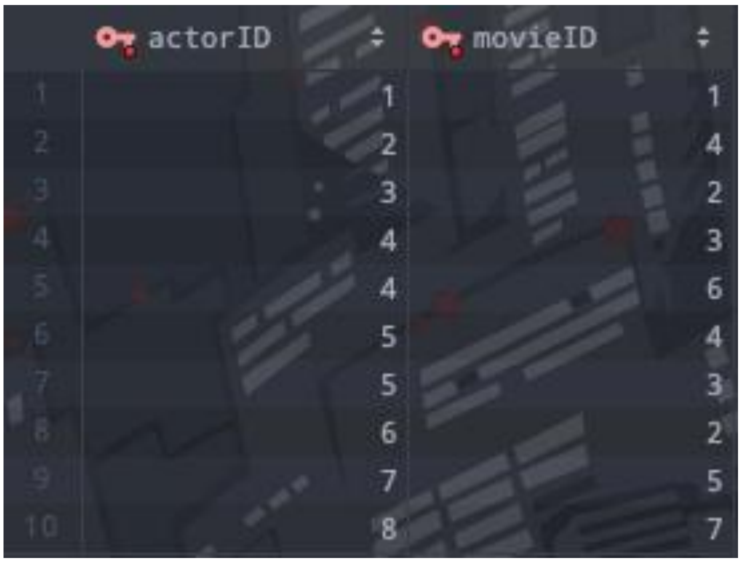
	directorID	awardID
1	2	3
2	2	9
3	2	4
4	2	14
5	6	3
6	6	4
7	6	9
8	6	13
9	7	3

Figure 6 awardWinningDirectors table after running the populationDB method.



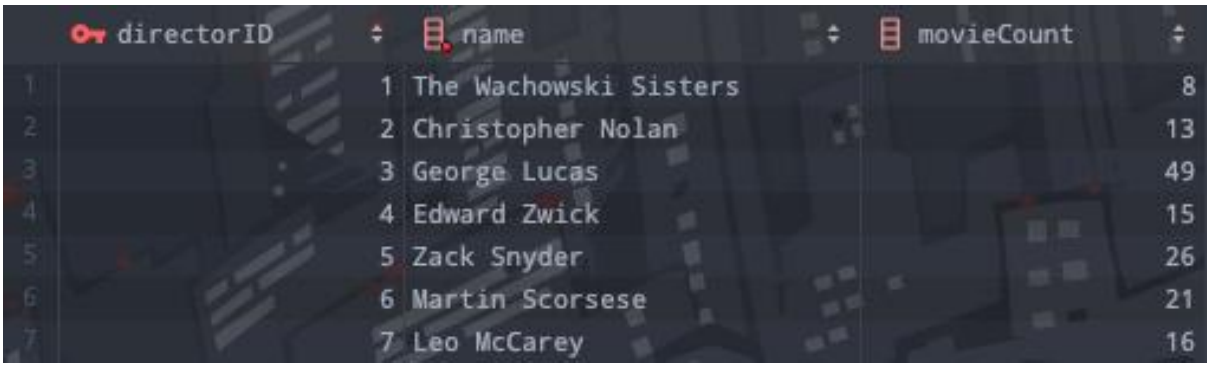
	movieID	awardID
1	6	2
2	7	11
3	7	12
4	7	7
5	8	2
6	8	15

Figure 7 awardWinningMovie table after running the populationDB method.



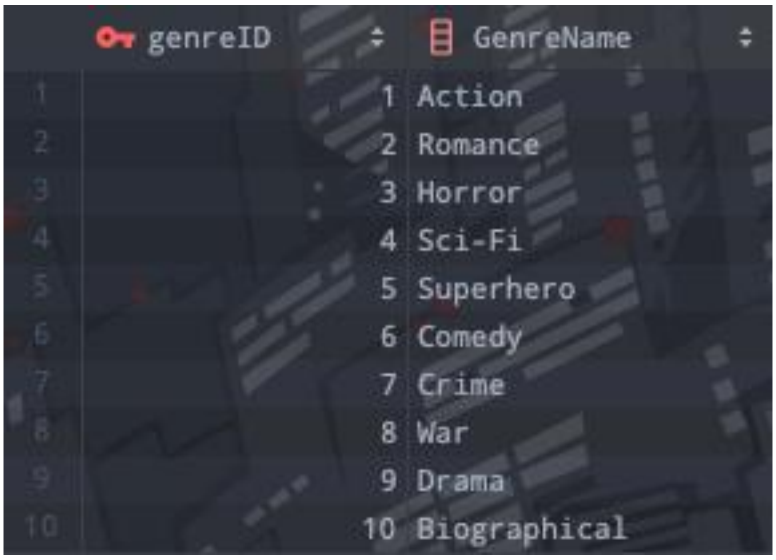
	actorID	movieID
1	1	1
2	2	4
3	3	2
4	4	3
5	4	6
6	5	4
7	5	3
8	6	2
9	7	5
10	8	7

Figure 8 Cast table after running the populationDB method.



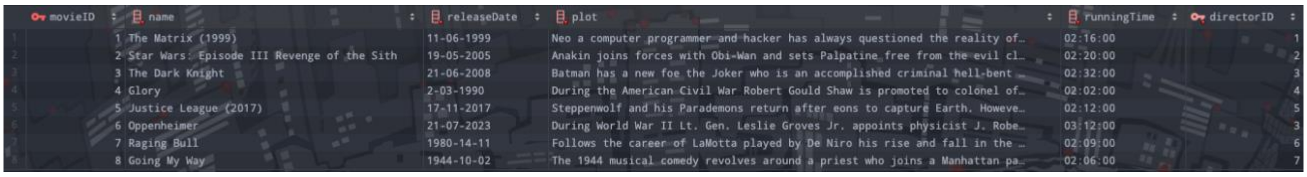
directorID	name	movieCount
1	The Wachowski Sisters	8
2	Christopher Nolan	13
3	George Lucas	49
4	Edward Zwick	15
5	Zack Snyder	26
6	Martin Scorsese	21
7	Leo McCarey	16

Figure 9 Director table after running the populationDB method.



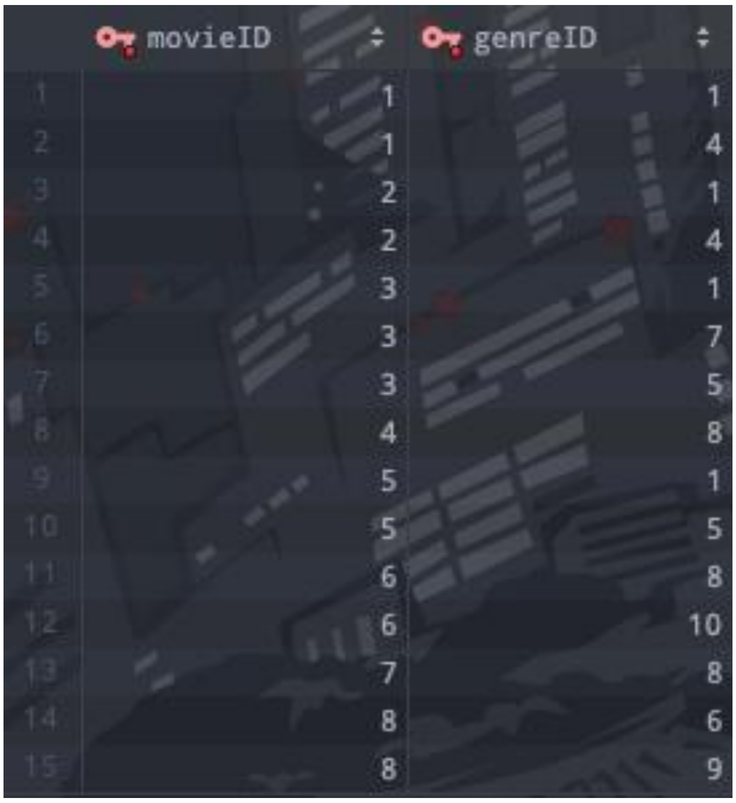
genreID	GenreName
1	Action
2	Romance
3	Horror
4	Sci-Fi
5	Superhero
6	Comedy
7	Crime
8	War
9	Drama
10	Biographical

Figure 10 Genre table after running the populationDB method.



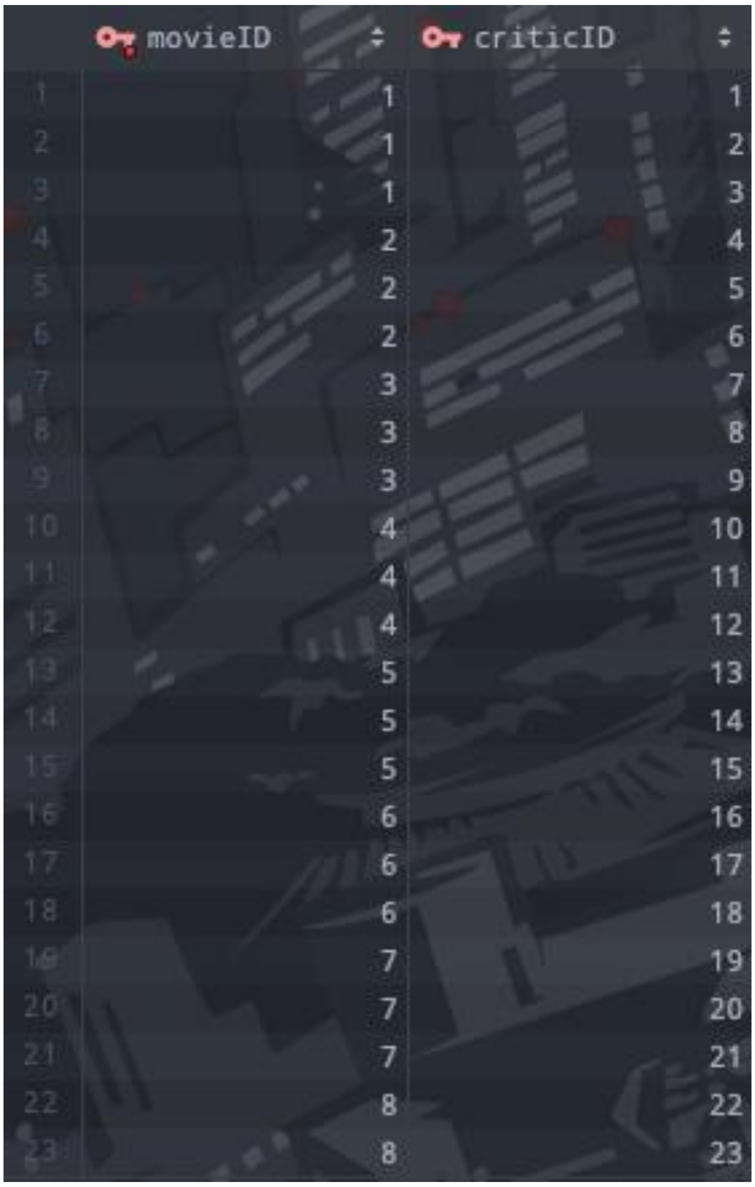
movieID	name	releaseDate	plot	runningTime	directorID
1	The Matrix (1999)	11-06-1999	Neo a computer programmer and hacker has always questioned the reality of…	02:16:00	1
2	Star Wars: Episode III Revenge of the Sith	19-05-2005	Anakin joins forces with Obi-Wan and sets Palpatine free from the evil cl…	02:20:00	2
3	The Dark Knight	21-06-2008	Batman has a new foe the Joker who is an accomplished criminal hell-bent …	02:32:00	3
4	Glory	2-03-1990	During the American Civil War Robert Gould Shaw is promoted to colonel of…	02:02:00	4
5	Justice League (2017)	17-11-2017	Steppenwolf and his Parademons return after eons to capture Earth. Howeve…	02:12:00	5
6	Oppenheimer	21-07-2023	During World War II Lt. Gen. Leslie Groves Jr. appoints physicist J. Robe…	03:12:00	3
7	Raging Bull	1980-14-11	Follows the career of LaMotta played by De Niro his rise and fall in the …	02:09:00	6
8	Going My Way	1944-10-02	The 1944 musical comedy revolves around a priest who joins a Manhattan pa…	02:06:00	7

Figure 11 Movies table after running the populationDB method.



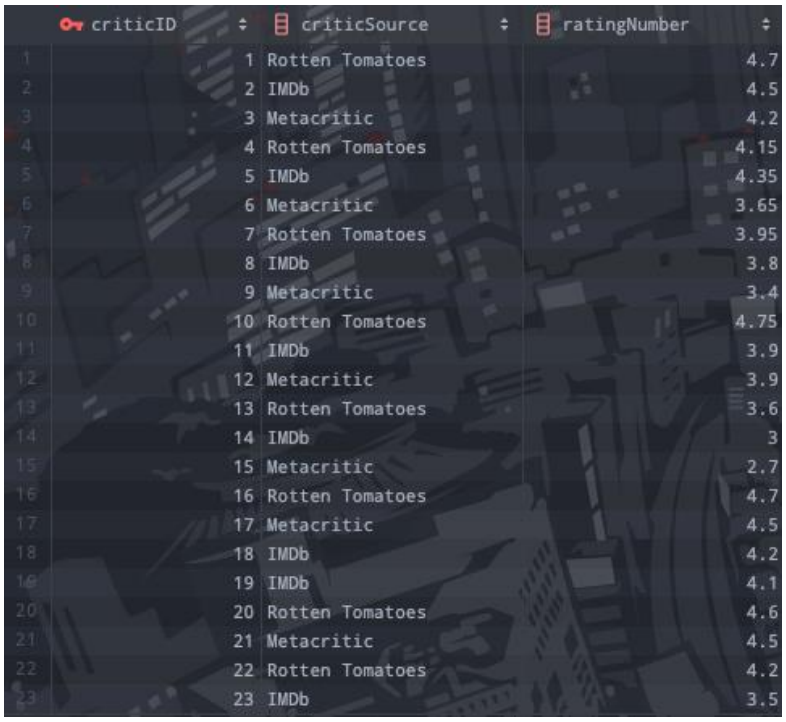
	movieID	genreID
1	1	1
2	1	4
3	2	1
4	2	4
5	3	1
6	3	7
7	3	5
8	4	8
9	5	1
10	5	5
11	6	8
12	6	10
13	7	8
14	8	6
15	8	9

Figure 12 moviesWithGenre table after running the populationDB method.



	movieID	criticID
1	1	1
2	1	2
3	1	3
4	2	4
5	2	5
6	2	6
7	3	7
8	3	8
9	3	9
10	4	10
11	4	11
12	4	12
13	5	13
14	5	14
15	5	15
16	6	16
17	6	17
18	6	18
19	7	19
20	7	20
21	7	21
22	8	22
23	8	23

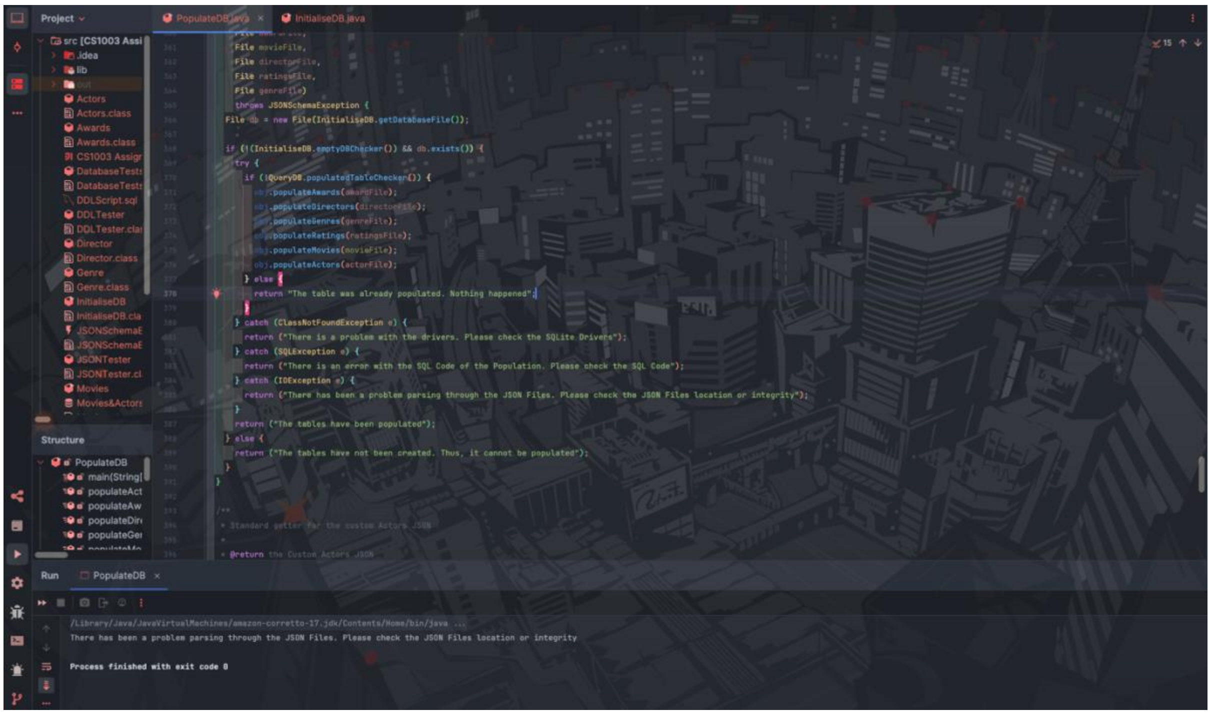
Figure 13 moviesWithRatings table after running the populationDB method.



criticID	criticSource	ratingNumber
1	Rotten Tomatoes	4.7
2	IMDb	4.5
3	Metacritic	4.2
4	Rotten Tomatoes	4.15
5	IMDb	4.35
6	Metacritic	3.65
7	Rotten Tomatoes	3.95
8	IMDb	3.8
9	Metacritic	3.4
10	Rotten Tomatoes	4.75
11	IMDb	3.9
12	Metacritic	3.9
13	Rotten Tomatoes	3.6
14	IMDb	3
15	Metacritic	2.7
16	Rotten Tomatoes	4.7
17	Metacritic	4.5
18	IMDb	4.2
19	IMDb	4.1
20	Rotten Tomatoes	4.6
21	Metacritic	4.5
22	Rotten Tomatoes	4.2
23	IMDb	3.5

Figure 14 Ratings table after running the populationDB method.

Aside from conventional testing methods, I decided to take my testing farther by directly moving or relocating files that are essential for the correct functioning of the test. This involved moving the Assignment 2 JSON Files folder, the DDLScript.sql and the TableClearDDL.sql someplace else. For the results of this test, please check the images below, as these tests could have not been applied using JUnit and therefore was done with no replicable code.

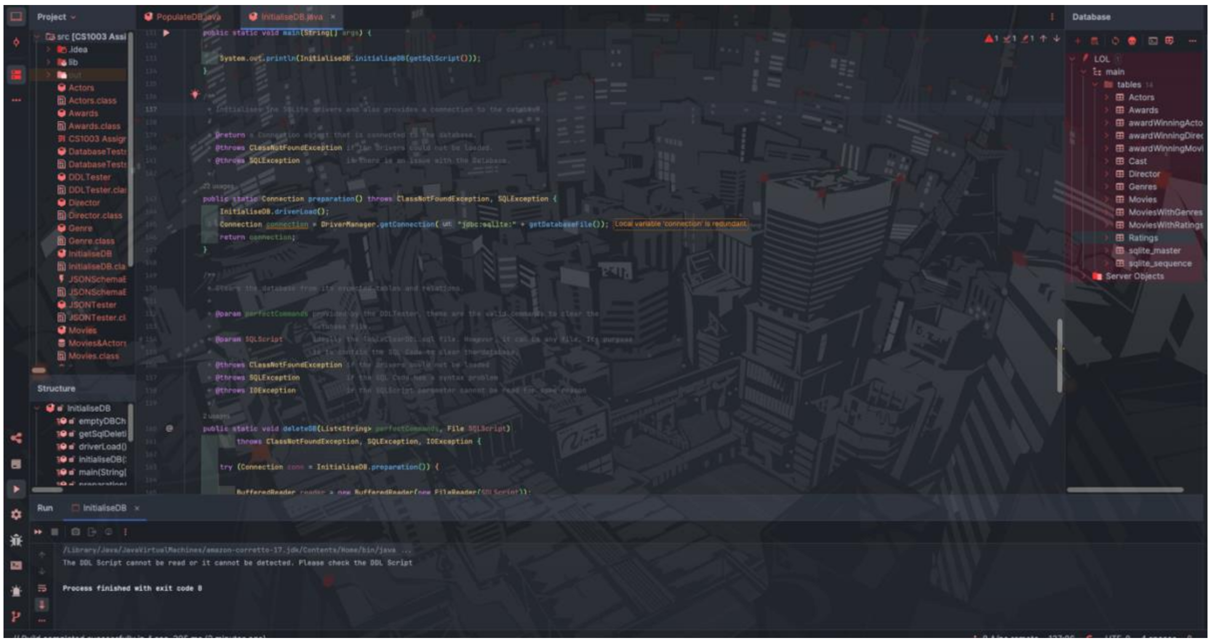


```java
    File movieFile,
    File directorFile,
    File ratingsFile,
    File genreFile)
    throws JSONSchemaException {
  File db = new File(InitialiseDB.getDatabaseFile());

  if (!(InitialiseDB.emptyDBChecker()) && db.exists()) {
    try {
      if (!QueryDB.populatedTableChecker()) {
        obj.populateAwards(awardFile);
        obj.populateDirectors(directorFile);
        obj.populateGenres(genreFile);
        obj.populateRatings(ratingsFile);
        obj.populateMovies(movieFile);
        obj.populateActors(actorFile);
      } else {
        return "The table was already populated. Nothing happened";
      }
    } catch (ClassNotFoundException e) {
      return ("There is a problem with the drivers. Please check the SQLite Drivers");
    } catch (SQLException e) {
      return ("There is an error with the SQL Code of the Population. Please check the SQL Code");
    } catch (IOException e) {
      return ("There has been a problem parsing through the JSON Files. Please check the JSON Files location or integrity");
    }
    return ("The tables have been populated");
  } else {
    return ("The tables have not been created. Thus, it cannot be populated");
  }
}

/**
 * Standard getter for the custom Actors JSON
 *
 * @return the Custom Actors JSON
```

```
/Library/Java/JavaVirtualMachines/amazon-corretto-17.jdk/Contents/Home/bin/java ...
There has been a problem parsing through the JSON Files. Please check the JSON Files location or integrity

Process finished with exit code 0
```

Figure 15 Correct message printing when the running the population() method when the Assignment 2 JSON Files are not available. Looking at the top corner, it is possible to notice the project structure, where the Assignment 2 JSON Folder is nowhere to be found.

In this case, there are no further tests to be applied without the Assignment 2 JSON Files, since the only class who actively uses the files in said folder is the PopulateDB Class. Next, we can observe the functioning of the program when the SQLScript.sql file is not on the correct folder (or has been deleted).

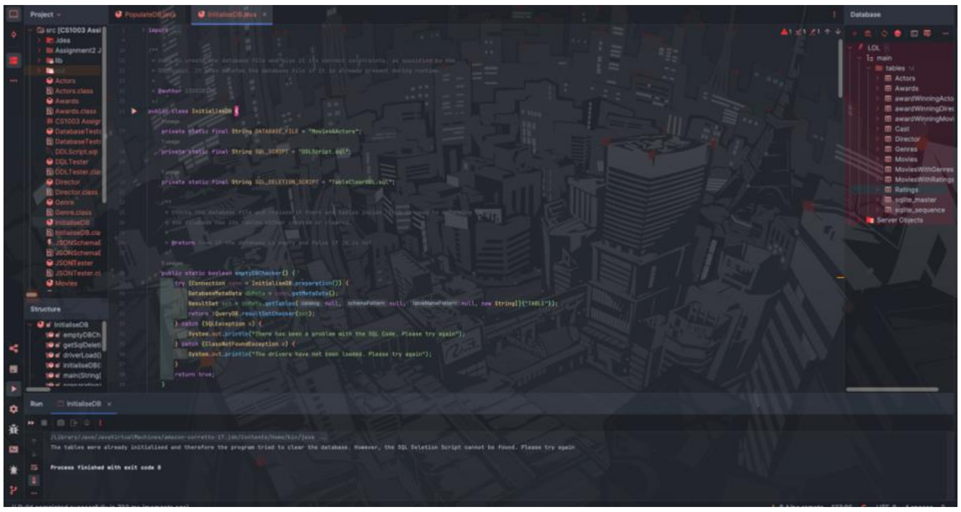


```java
public static void main(String[] args) {

    System.out.println(InitialiseDB.initialiseDB(getSqlScript()));
}

public static Connection preparation() throws ClassNotFoundException, SQLException {
    InitialiseDB.driverLoad();
    Connection connection = DriverManager.getConnection( url: "jdbc:sqlite:" + getDatabaseFile());
    return connection;
}

public static void deleteDB(List<String> perfectCommands, File SQLScript)
        throws ClassNotFoundException, SQLException, IOException {

    try (Connection conn = InitialiseDB.preparation()) {

        BufferedReader reader = new BufferedReader(new FileReader(SQLScript));
```

```
/Library/Java/JavaVirtualMachines/amazon-corretto-17.jdk/Contents/Home/bin/java ...
The DDL Script cannot be read or it cannot be detected. Please check the DDL Script

Process finished with exit code 0
```

Figure 16 Correct message printing when the running the initialiseDB method when the SQLScript.sql is not available. Looking at the top corner, it is possible to notice the project structure, where the SQLScript.sql file is nowhere to be found.

Also, in this case there are no further tests to be done, since this is the only class that uses the file. Below, we can observe the functioning of the program when the TableClearDDL.sql File is not on the correct folder (or has been deleted)



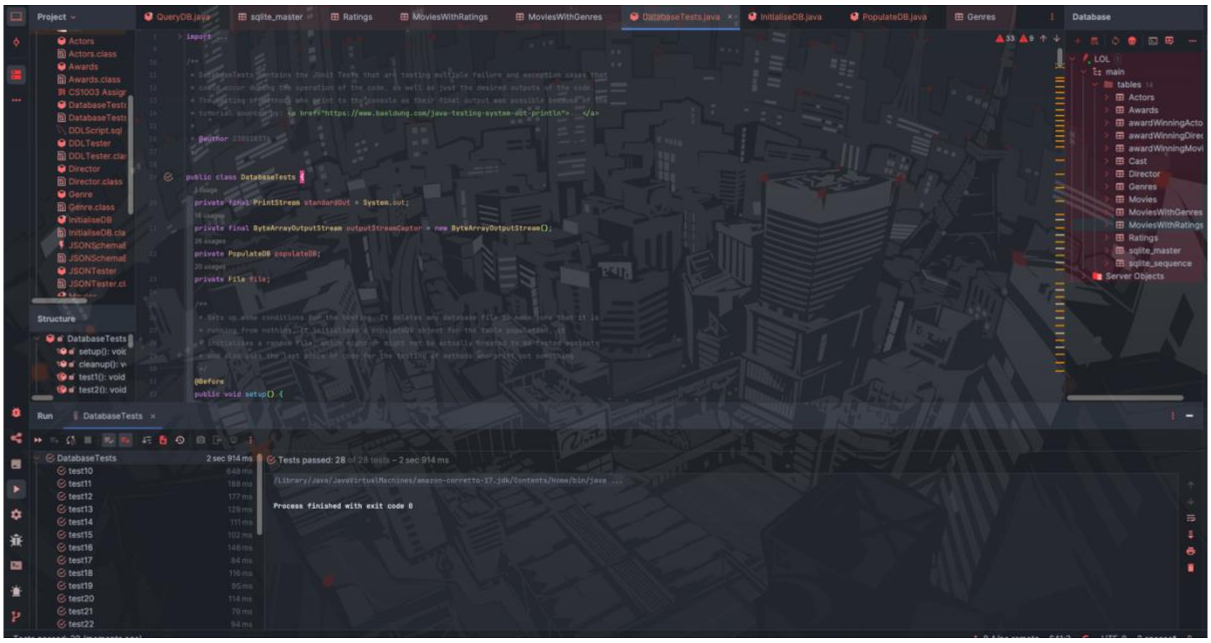
```java
public class InitialiseDB {
    private static final String DATABASE_FILE = "Movies&Actors";
    private static final String SQL_SCRIPT = "SQLScript.sql";

    private static final String SQL_DELETION_SCRIPT = "TableClearDDL.sql";

    public static boolean emptyDBChecker() {
        try (Connection conn = InitialiseDB.preparation()) {
            DatabaseMetaData dbMeta = conn.getMetaData();
            ResultSet set = dbMeta.getTables( catalog: null, schemaPattern: null, tableNamePattern: null, new String[]{"TABLE"});
            return !QueryDB.resultSetChecker(set);
        } catch (SQLException e) {
            System.out.println("There has been a problem with the SQL Code. Please try again");
        } catch (ClassNotFoundException e) {
            System.out.println("The drivers have not been loaded. Please try again");
        }
        return true;
    }
```

```
/Library/Java/JavaVirtualMachines/amazon-corretto-17.jdk/Contents/Home/bin/java ...
The tables were already initialised and therefore the program tried to clear the database. However, the SQL Deletion Script cannot be found. Please try again

Process finished with exit code 0
```

Figure 17 Correct message printing when the running the initialiseDB method when the TableClearDDl.sql file is not available. Looking at the top corner, it is possible to notice the project structure, where the TableClearDDL.sql file is nowhere to be found.

Just like the former files, there is no use testing for other classes or methods, since this is the only class who actively uses those files.

Below we can see the passing of all the aforementioned JUnit Tests, which serves as evidence of the statements presented in the table displayed at the top of this section:

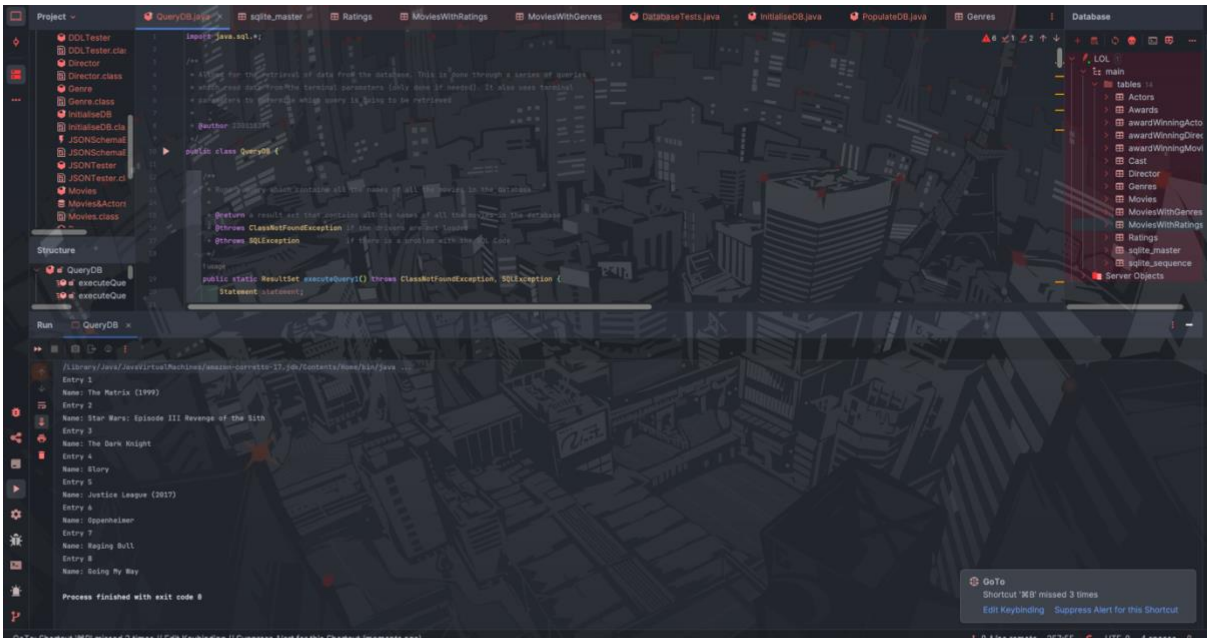


```
DatabaseTests   2 sec 914 ms
  test10        648 ms
  test11        188 ms
  test12        177 ms
  test13        129 ms
  test14        111 ms
  test15        102 ms
  test16        146 ms
  test17         84 ms
  test18        116 ms
  test19         95 ms
  test20        114 ms
  test21         70 ms
  test22         94 ms

Tests passed: 28 of 28 tests – 2 sec 914 ms

/Library/Java/JavaVirtualMachines/amazon-corretto-17.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

Figure 18 *Perfect Passing of all the JUnit Tests. For more information regarding the tests and the purpose of each, please check the DatabaseTests.html file contained in the Javadoc folder, which contains additional information for each test.*

I would also like to provide some examples of the QueryDB Class. Since these depend on the parameters, I am going to choose existing parameters to demonstrate functionality:

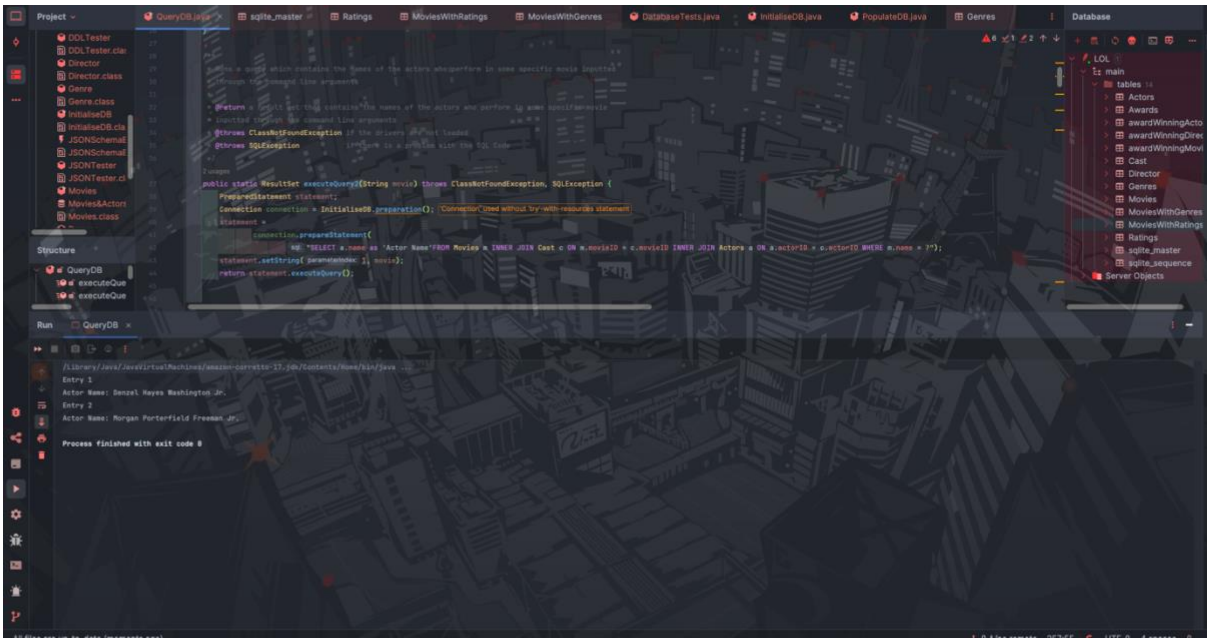
Query1:



```
/Library/Java/JavaVirtualMachines/amazon-corretto-17.jdk/Contents/Home/bin/java ...
Entry 1
Name: The Matrix (1999)
Entry 2
Name: Star Wars: Episode III Revenge of the Sith
Entry 3
Name: The Dark Knight
Entry 4
Name: Glory
Entry 5
Name: Justice League (2017)
Entry 6
Name: Oppenheimer
Entry 7
Name: Raging Bull
Entry 8
Name: Going My Way

Process finished with exit code 0
```

Figure 19 Example of Query1. This does not take parameters.

Query2:

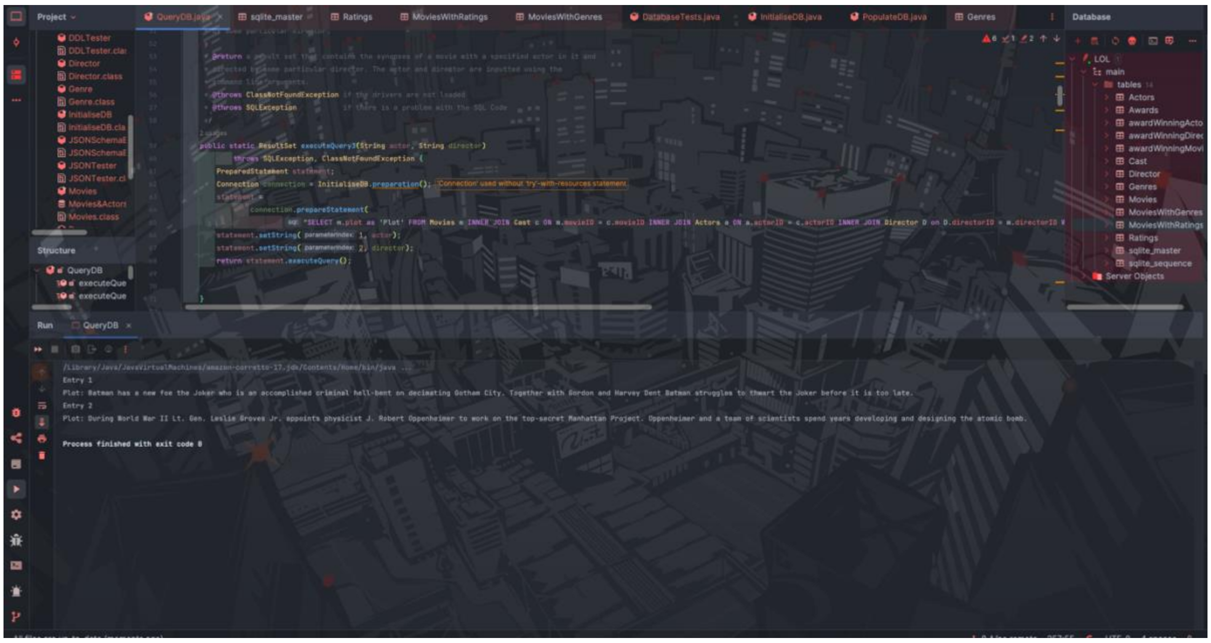


```java
 * ...s a query which contains the names of the actors who perform in some specific movie inputted
 * through the command line arguments
 *
 * @return a result set that contains the names of the actors who perform in some specific movie
 * inputted through the command line arguments
 * @throws ClassNotFoundException if the drivers are not loaded
 * @throws SQLException          if there is a problem with the SQL Code
 */
public static ResultSet executeQuery2(String movie) throws ClassNotFoundException, SQLException {
    PreparedStatement statement;
    Connection connection = InitialiseDB.preparation();
    statement =
            connection.prepareStatement(
                    sql: "SELECT a.name as 'Actor Name'FROM Movies m INNER JOIN Cast c ON m.movieID = c.movieID INNER JOIN Actors a ON a.actorID = c.actorID WHERE m.name = ?");
    statement.setString( parameterIndex: 1, movie);
    return statement.executeQuery();
```

```
/Library/Java/JavaVirtualMachines/amazon-corretto-17.jdk/Contents/Home/bin/java ...
Entry 1
Actor Name: Denzel Hayes Washington Jr.
Entry 2
Actor Name: Morgan Porterfield Freeman Jr.

Process finished with exit code 0
```

Figure 20 Example of Query2. This took the parameter Glory, as the name of the movie

Query3:

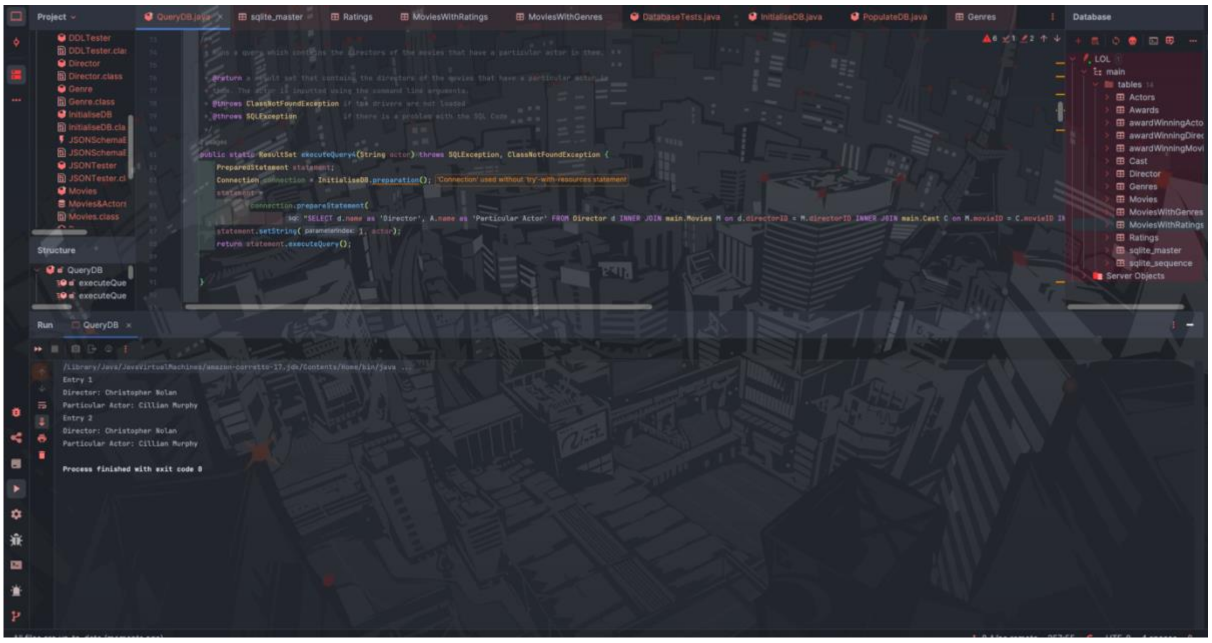


```java
 * @return a result set that contains the synopses of a movie with a specified actor in it and
 * directed by some particular director. The actor and director are inputted using the
 * command line arguments.
 * @throws ClassNotFoundException if the drivers are not loaded
 * @throws SQLException          if there is a problem with the SQL Code
 */
public static ResultSet executeQuery3(String actor, String director)
        throws SQLException, ClassNotFoundException {
    PreparedStatement statement;
    Connection connection = InitialiseDB.preparation();
    statement =
            connection.prepareStatement(
                    sql: "SELECT m.plot as 'Plot' FROM Movies m INNER JOIN Cast c ON m.movieID = c.movieID INNER JOIN Actors a ON a.actorID = c.actorID INNER JOIN Director D on D.directorID = m.directorID W
    statement.setString( parameterIndex: 1, actor);
    statement.setString( parameterIndex: 2, director);
    return statement.executeQuery();
}
```

```
/Library/Java/JavaVirtualMachines/amazon-corretto-17.jdk/Contents/Home/bin/java ...
Entry 1
Plot: Batman has a new foe the Joker who is an accomplished criminal hell-bent on decimating Gotham City. Together with Gordon and Harvey Dent Batman struggles to thwart the Joker before it is too late.
Entry 2
Plot: During World War II Lt. Gen. Leslie Groves Jr. appoints physicist J. Robert Oppenheimer to work on the top-secret Manhattan Project. Oppenheimer and a team of scientists spend years developing and designing the atomic bomb.

Process finished with exit code 0
```

Figure 21 Example of Query3. This took the parameters Cillian Murphy and Christopher Nolan as the respective names.

Query4:

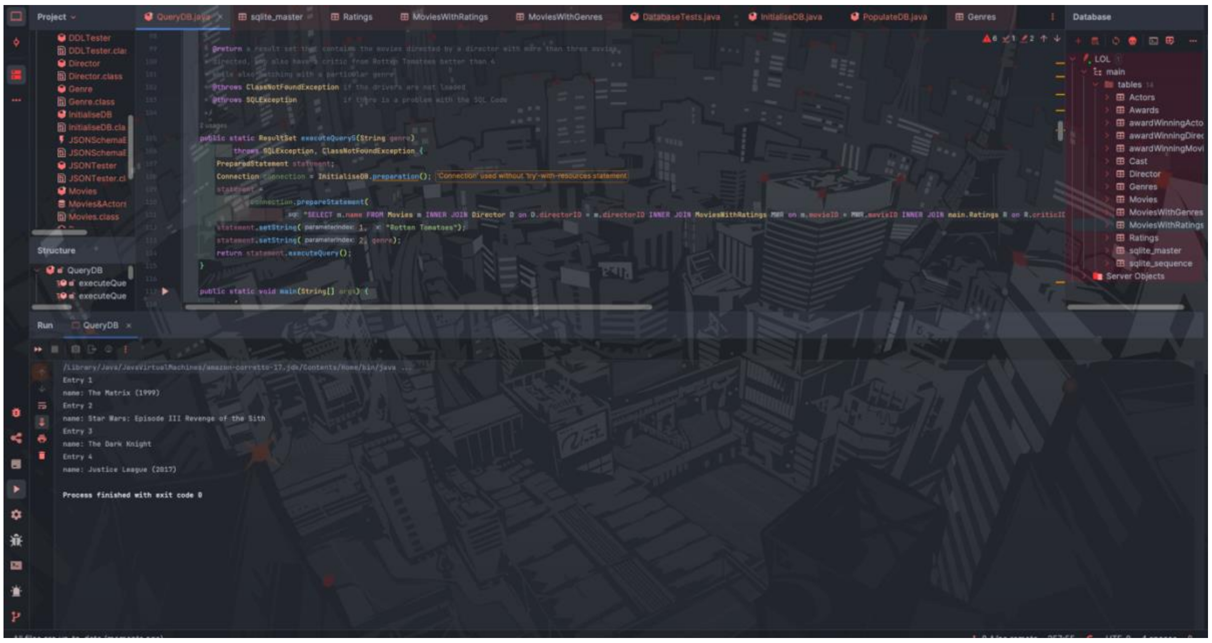


```java
 * Runs a query which contains the directors of the movies that have a particular actor in them.
 *
 * @return a result set that contains the directors of the movies that have a particular actor in
 * them. The actor is inputted using the command line arguments.
 * @throws ClassNotFoundException if the drivers are not loaded
 * @throws SQLException           if there is a problem with the SQL Code
 */
public static ResultSet executeQuery4(String actor) throws SQLException, ClassNotFoundException {
    PreparedStatement statement;
    Connection connection = InitialiseDB.preparation();
    statement =
            connection.prepareStatement(
                    sql: "SELECT d.name as 'Director', A.name as 'Particular Actor' FROM Director d INNER JOIN main.Movies M on d.directorID = M.directorID INNER JOIN main.Cast C on M.movieID = C.movieID IN
    statement.setString( parameterIndex: 1, actor);
    return statement.executeQuery();

}
```

```
/Library/Java/JavaVirtualMachines/amazon-corretto-17.jdk/Contents/Home/bin/java ...
Entry 1
Director: Christopher Nolan
Particular Actor: Cillian Murphy
Entry 2
Director: Christopher Nolan
Particular Actor: Cillian Murphy

Process finished with exit code 0
```

Figure 22 *Example of Query4. This took the parameter Cillian Murphy*

Query5:

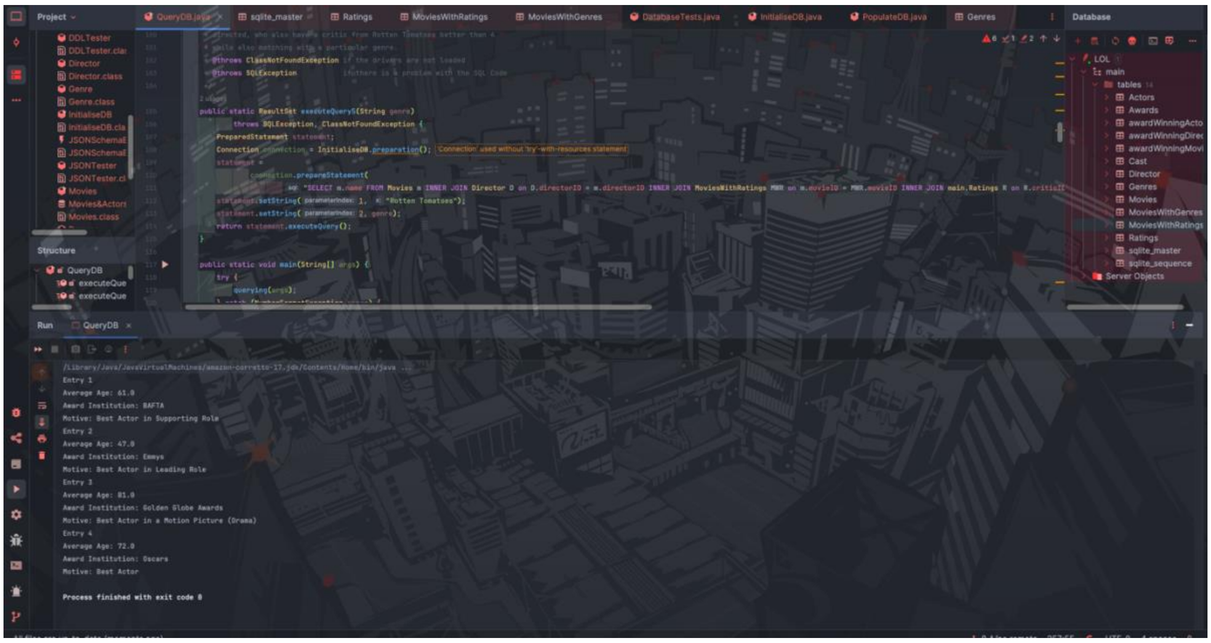


```java
 * @return a result set that contains the movies directed by a director with more than three movies
 * directed, and also have a critic from Rotten Tomatoes better than 6
 * while also matching with a particular genre
 * @throws ClassNotFoundException if the drivers are not loaded
 * @throws SQLException           if there is a problem with the SQL Code
 */
public static ResultSet executeQuery5(String genre)
        throws SQLException, ClassNotFoundException {
    PreparedStatement statement;
    Connection connection = InitialiseDB.preparation();
    statement =
            connection.prepareStatement(
                    sql: "SELECT m.name FROM Movies m INNER JOIN Director D on D.directorID = m.directorID INNER JOIN MoviesWithRatings MWR on m.movieID = MWR.movieID INNER JOIN main.Ratings R on R.criticID
    statement.setString( parameterIndex: 1, x: "Rotten Tomatoes");
    statement.setString( parameterIndex: 2, genre);
    return statement.executeQuery();
}

public static void main(String[] args) {
```

```
/Library/Java/JavaVirtualMachines/amazon-corretto-17.jdk/Contents/Home/bin/java ...
Entry 1
name: The Matrix (1999)
Entry 2
name: Star Wars: Episode III Revenge of the Sith
Entry 3
name: The Dark Knight
Entry 4
name: Justice League (2017)

Process finished with exit code 0
```

Figure 23 *Example of Query5. This took the parameter Action.*

Query6:



```java
 * directed, who also have a critic from Rotten Tomatoes better than 4
 * while also matching with a particular genre.
 * @throws ClassNotFoundException if the drivers are not loaded
 * @throws SQLException          ifthere is a problem with the SQL Code

public static ResultSet executeQuery5(String genre)
        throws SQLException, ClassNotFoundException {
    PreparedStatement statement;
    Connection connection = InitialiseDB.preparation();
    statement =
            connection.prepareStatement(
                    sql: "SELECT m.name FROM Movies m INNER JOIN Director D on D.directorID = m.directorID INNER JOIN MoviesWithRatings MWR on m.movieID = MWR.movieID INNER JOIN main.Ratings R on R.criticI
    statement.setString( parameterIndex: 1, x: "Rotten Tomatoes");
    statement.setString( parameterIndex: 2, genre);
    return statement.executeQuery();
}

public static void main(String[] args) {
    try {
        querying(args);
```

```
/Library/Java/JavaVirtualMachines/amazon-corretto-17.jdk/Contents/Home/bin/java ...
Entry 1
Average Age: 61.0
Award Institution: BAFTA
Motive: Best Actor in Supporting Role
Entry 2
Average Age: 47.0
Award Institution: Emmys
Motive: Best Actor in Leading Role
Entry 3
Average Age: 81.0
Award Institution: Golden Globe Awards
Motive: Best Actor in a Motion Picture (Drama)
Entry 4
Average Age: 72.0
Award Institution: Oscars
Motive: Best Actor

Process finished with exit code 0
```

Figure 24 Example of Query6. This took no parameters.

Evaluation:

My program successfully completes all of the 5 required tasks described in the specification. Additionally, it handles many possible exception (or failure) cases, which greatly improves user experience, as it directly specifies error causes such as missing files, improper schemas or even SQL Code Errors inside of the scripts or Prepared Statements.

I also believe that my database design allows for further ampliation and the addition of many multiple entries that cover a great plethora of cases and situations without much modification. This denotes the usefulness and reusability of my software.

Conclusion:

My program successfully:

- Contains a database that stores all of the necessary attributes described in the specification. It also uses primary keys, foreign keys and intermediate tables in order to represent their relationships as accurately as possible.
- Turns a series of custom JSON Files into arrays of objects, which contain all of the necessary information to appropriately populate the tables and establish the relationships between them.
- Uses JDBC in order to effectively implement the relational schema, populate the tables that are inside of the database and finally to allow the users to query for data inside of the database system.
- Handles a great majority of the possible errors and failures that can occur during its operation, pointing the user the specific cause of the error or the specific missing file.

I found the parsing without libraries the most difficult task, as I had to come up with a way to parse through a file with no previous code. The rest was done simply by using basic JDBC and SQL Code. I am sure that if I had libraries and .jar support, this part would have been trivial.

Given more time, I would have liked to add more entries to my database and build a better codebase to support a comparation of JSON Schemas (by using a custom JSON Schema that I would have built). Aside from that I cannot really come up with possible improvements to the code. Although, I could also consider using other formats such as a .csv to try to have all of the data in a single line, which could prove to be more efficient.

One of the few limitations of my program is related to the fact that it cannot carry out any CRUD Operations, except for the retrieval of very specific prepared SQL Code. However, this is correspondent to the specification that was given, so it cannot be really counted as a limitation per se.

References:

- StackOverflow(2010) https://stackoverflow.com/questions/867194/java-resultset-how-to-check-if-there-are-any-results
- Baeldung(2024) https://www.baeldung.com/java-testing-system-out-println