Enterprise Computing

ASSIGNMENT 1

JACEK JANCZURA

MART. NR: 404975

Important info and credentials

Github repo: https://github.com/jjanczur/ATMSystem

AWS account ID: 701266085448

DB type: MySQL

DB credentials:

- **DB_URL**=jdbc:mysql://ec-atm-assignment1.cpmccfa2pqqj.eu-central-1.rds.amazonaws.com:3306/innodb
- **USER**=Ecatm
- **PASS**=EnterpriceComputing1

(Note that the pass is with „c” not „s”)

To run ATM System execute jar by command: java -jar ATMSystem-1.0-SNAPSHOT-shaded.jar

System requirenments

1. Relational database
2. ATM Java Client
    a. Login using her account number and corresponding pin code
    b. Deposit funds
    c. Withdraw funds
    d. Display account balance
3. Concurrent Access
    a. Many ATM transactions happen in parallel
    b. The account balance must not be negative

System requirenments

1. Relational database
2. ATM Java Client
   a. Login using her account number and corresponding pin code
   b. Deposit funds
   c. Withdraw funds
   d. Display account balance
3. Concurrent Access
   a. Many ATM transactions happen in parallel
   b. The account balance must not be negative

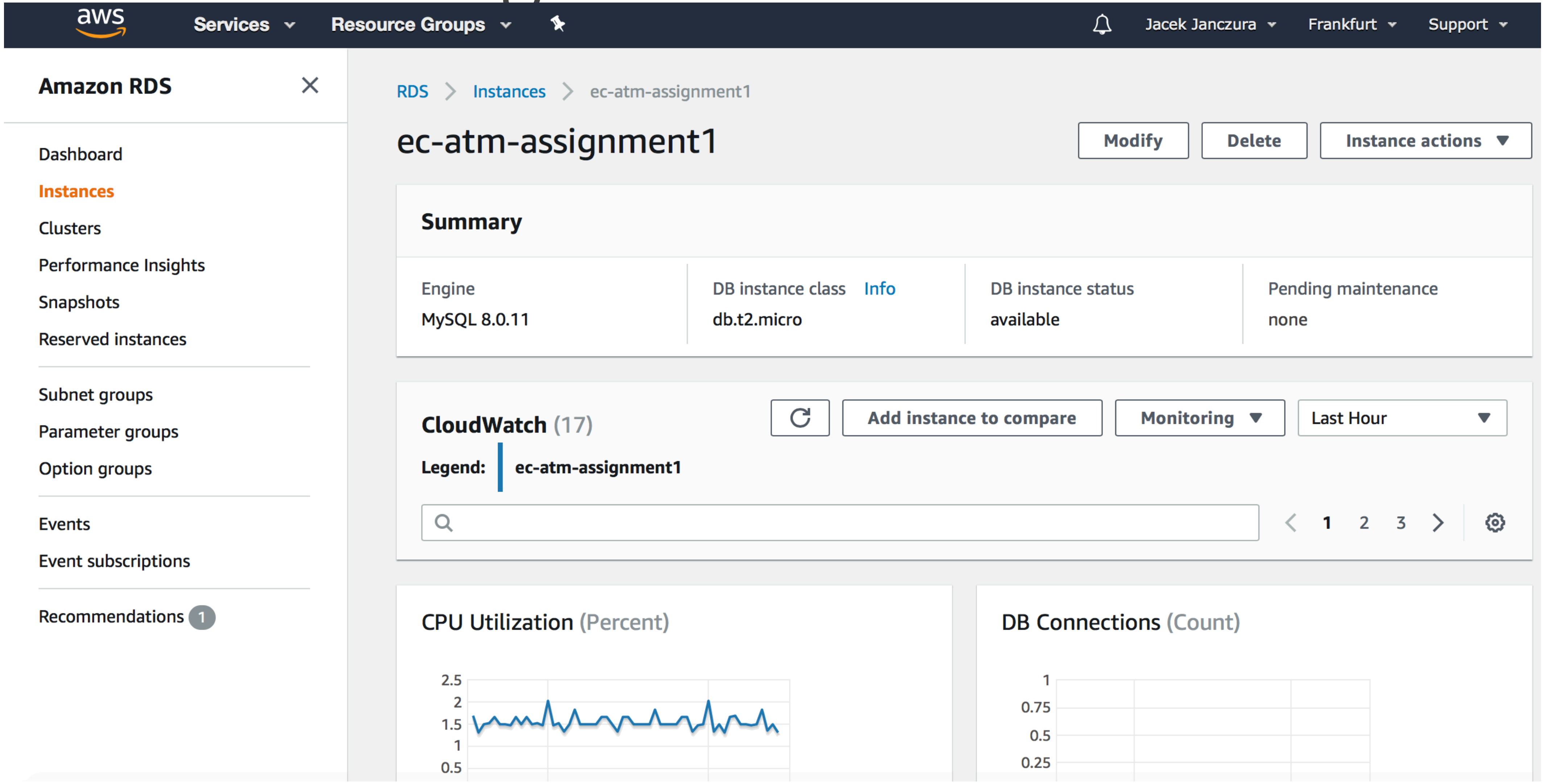
1. Relational database

1. Create DB query:
    a. CREATE TABLE IF NOT EXISTS account (account_id INT, pinCode INT, balance decimal(10,2) unsigned, UNIQUE (account_id)
    b. CHECK (balance > 0) could be added to balance column. Unfortunately CHECK clause is parsed but ignored by all storage engines.[1] So there is no point in adding CHECK clause because it does nothing - as the manual states
    c. System requirement 3.b. - type of balance column is unsigned in combination with sql_mode = 'STRICT_TRANS_TABLES' it results in SQLException. When ATM tries to update balance with the negative value an exception is thrown with the message "Data truncation: Out of range value for column 'balance' at row n"

       SET @@GLOBAL.sql_mode = 'STRICT_TRANS_TABLES'; - this mode should not be set globally
       SET @@SESSION.sql_mode = 'STRICT_TRANS_TABLES'; - this command is executed for each session
2. Inserting values:
    a. INSERT INTO account VALUES(1, 1111, 50), (2, 2222, 50), (3,3333,50)

[1] https://dev.mysql.com/doc/refman/8.0/en/create-table.html

1. AWS management console



ec-atm-assignment1

Summary

Engine	DB instance class	DB instance status	Pending maintenance
MySQL 8.0.11	db.t2.micro	available	none

CloudWatch (17)

Legend: ec-atm-assignment1

CPU Utilization (Percent)

DB Connections (Count)

1. MySQL Workbench

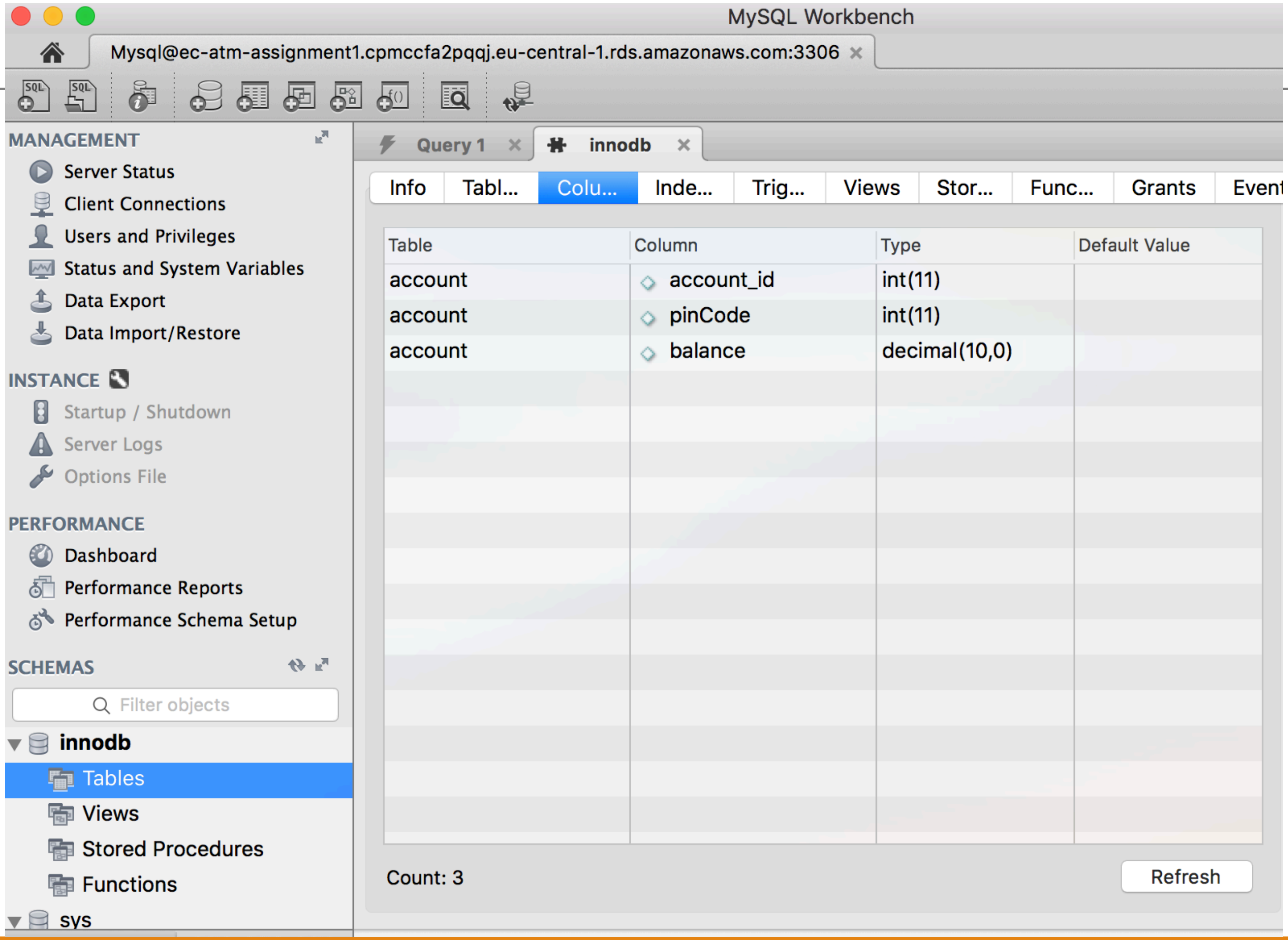


Table	Column	Type	Default Value
account	account_id	int(11)	
account	pinCode	int(11)	
account	balance	decimal(10,0)	

Count: 3

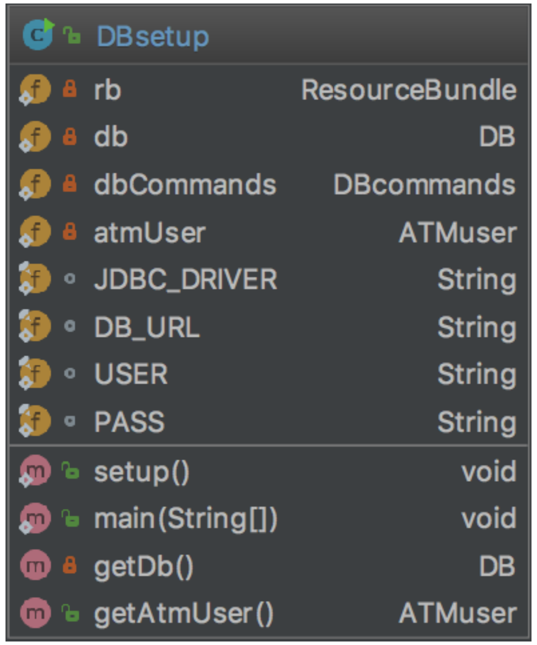
System requirenments

1. Relational database
2. ATM Java Client
    a. Login using her account number and corresponding pin code
    b. Deposit funds
    c. Withdraw funds
    d. Display account balance
3. Concurrent Access
    a. Many ATM transactions happen in parallel
    b. The account balance must not be negative

2. Java Client

The Java Client consists a few classes:

- DBsetup – Class that is responsible for setting up the connection with DB.
  - It reads the DBconfig.properties - file where credentials are stored (file added to .gitignore)
  - Runs a main function where user menu display is called
  - Creates the connection to DB and passes the connection into object of the class DB



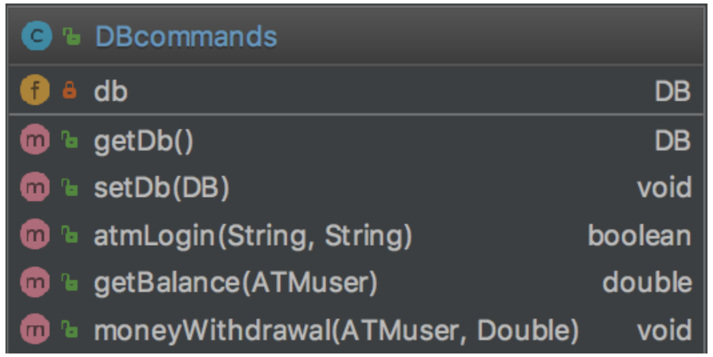
DBsetup	
rb	ResourceBundle
db	DB
dbCommands	DBcommands
atmUser	ATMuser
JDBC_DRIVER	String
DB_URL	String
USER	String
PASS	String
setup()	void
main(String[])	void
getDb()	DB
getAtmUser()	ATMuser

2. Java Client

The Java Client consists a few classes:

- DBcommands– Class that executes whole queries to DB. Methods from this class are used by ATMuser. All of the methodes are SQLInjection safe due to the correct usage of PreparedStatement
  - atmLogin() – method that executes select query to find out if account id and pin are correct. If they are the method returns true otherwise false
  - getBalance() – method that executes select query to find the balance of the user account.
  - moneyWithdrawal() - *Methode to perform withdrawal operation.*
    Demarcation in 2PC[1] is used to assure that balance after withdrawal is > 0.
    If not transaction is rolled back.
    1. Check if user is correctly logged in.
    2. Perform sql update query - money withdrawal
    3. if the balance is < 0 - rollback else commit

    ** **@param** atmUser Object that represents user*
    ** **@param** amount Amount of money to withdrawal from users bank account*
    ** **@throws** SQLException Problem with sql statement execution*
    ** **@throws** ATMaccountException Exception it inform about obligatory login*
  - Money deposit is withdrawal but with opposite value so the same method can be used



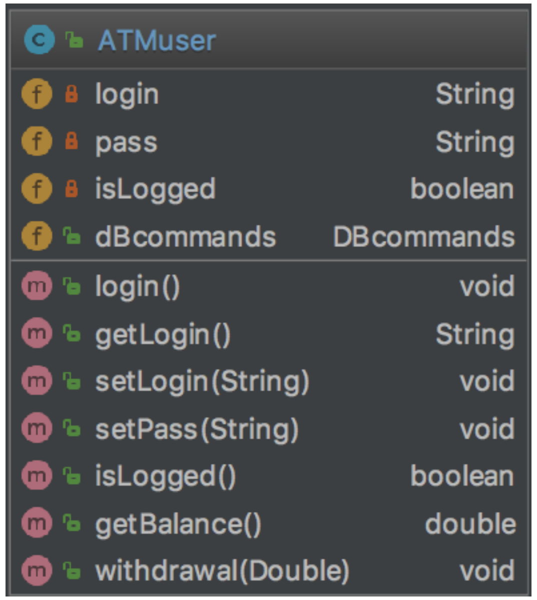
DBcommands	
db	DB
getDb()	DB
setDb(DB)	void
atmLogin(String, String)	boolean
getBalance(ATMuser)	double
moneyWithdrawal(ATMuser, Double)	void

[1] https://dev.mysql.com/doc/refman/8.0/en/xa-states.html

2. Java Client

The Java Client consists a few classes:

- ATMusers - class that represents the user which is using the ATM. This class executes the commands from Dbcommands, stores the login and password and information if the user that is using the ATM is correctly logged in.
- At first I wanted it to extend Dbcommands but now I've decided to pass in a constructor Dbcommands object that will execute queries against one DB in one DB connection

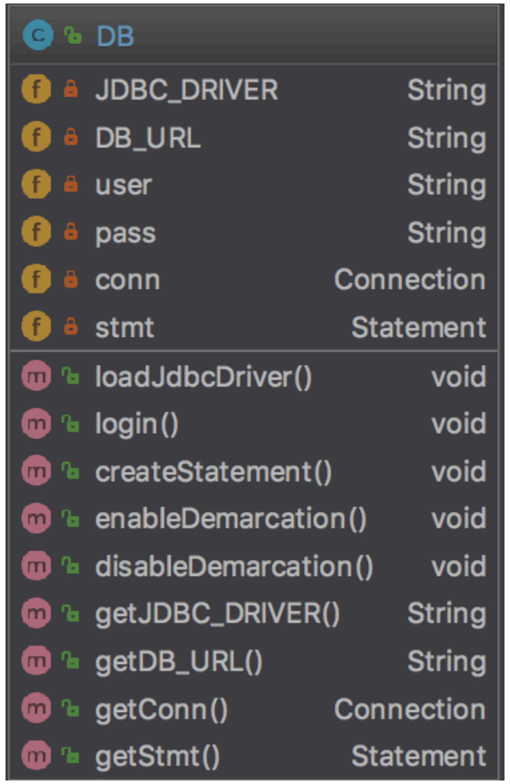


```
ATMuser
login                 String
pass                  String
isLogged              boolean
dBcommands            DBcommands
login()               void
getLogin()            String
setLogin(String)      void
setPass(String)       void
isLogged()            boolean
getBalance()          double
withdrawal(Double)    void
```

2. Java Client

The Java Client consists a few classes:

- DB – Class which objects represent a connection to DB
- Object of this class can:
  - Enable and disable demarcation mechanism which is a base for keeping atomicity of each update transaction. Only fully correct transaction is committed to DB.
  - Setup a connection with DB
    - To keep high isolation of transaction even though the default transaction isolation level for MySQL db is REPEATABLE_READ[1] was changed to SERIALIZABLE. Due to this isolation level anomalies such as dirty read, non-repeatable read and phantom read are prevented
    - To force db to throw *SQLException* when client tries to update balance with negative values (even though positive balance is checked during each witdrawal otherwise transaction is rolledback) sql_mode = 'STRICT_TRANS_TABLES' is eabled



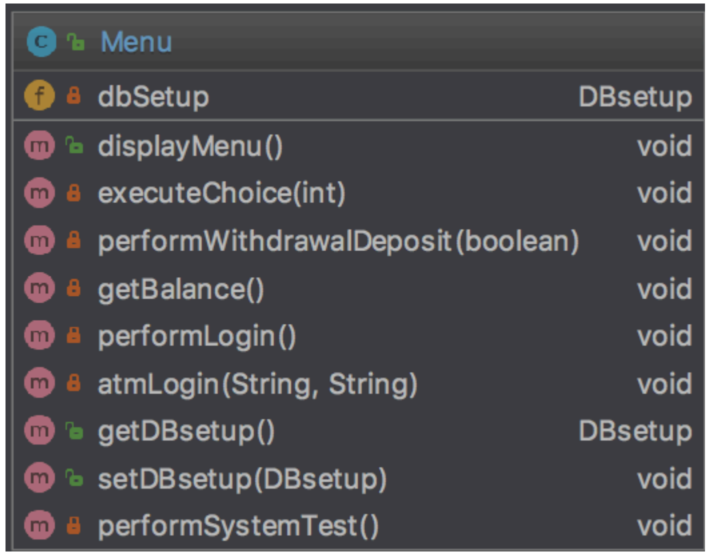
DB	
JDBC_DRIVER	String
DB_URL	String
user	String
pass	String
conn	Connection
stmt	Statement
loadJdbcDriver()	void
login()	void
createStatement()	void
enableDemarcation()	void
disableDemarcation()	void
getJDBC_DRIVER()	String
getDB_URL()	String
getConn()	Connection
getStmt()	Statement

[1] https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html

2. Java Client

The Java Client consists a few classes:

- Menu – Class that is responsible to correctly display user menu and react for user choices.
  - Class show to the user options that can be used in ATM.
  - For testing purpose hidden option 5 has been added - execute withdrawal without demarcation and balance check
  - It calls the methods to withdrawal, deposit, login and perform the tests
- ATMaccountExceptio – This exception is thrown when the user is not logged in and tries to perform any operation on the account or user tries to login with wrong credentials
- BalanceException – This exception is thrown on every attempt setting up the balance below zero



Menu	
dbSetup	DBsetup
displayMenu()	void
executeChoice(int)	void
performWithdrawalDeposit(boolean)	void
getBalance()	void
performLogin()	void
atmLogin(String, String)	void
getDBsetup()	DBsetup
setDBsetup(DBsetup)	void
performSystemTest()	void

2. Java Client

DB testing – The basic test done by user in “one ATM” –

1. Login to user 1 with pass 1111
    - withdrawal money,
    - deposit money,
    - withdrawal more than the balance
2. Repeat the same operation on the other user for example user 2 with pass 2222
3. trying to login with wrong credentials and repeat nr 1.

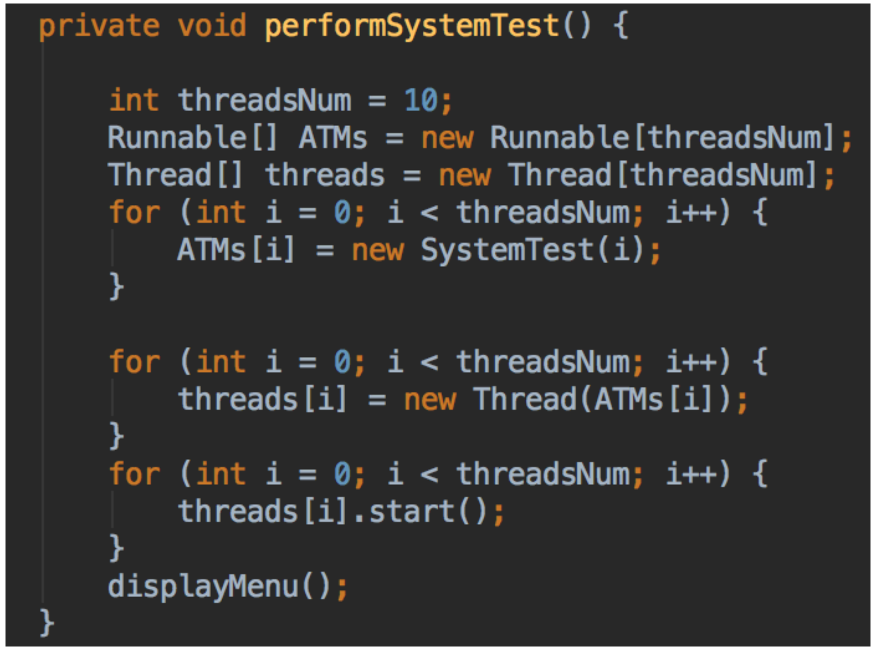
This type of easy testing checks if the basic methods and general architecture are working correctly but the real test is to login to different ATMs with the same credentials in the same time and trying to validate ACID principles in the system.

2. Java Client

DB testing – The real test done by user many ATMs in the same time

To talk about concurrent operations we need to create ATMs and run them in a different threads

1. Class SystemTest which implements interface runnable was created. The class creates own db connection, setup DB, passes it to DBcommands and creates user object which can execute queries against this db. This setup is equivalent to one ATM with one user.
2. User tries to withdrawal ammount lower than balance on his account
3. Test is performed by creating and running 10 threads. To each thread is assigned processor time and we can assume that the threads are working “concurently”.
4. Results in a file *DBTest.log*



```java
private void performSystemTest() {

    int threadsNum = 10;
    Runnable[] ATMs = new Runnable[threadsNum];
    Thread[] threads = new Thread[threadsNum];
    for (int i = 0; i < threadsNum; i++) {
        ATMs[i] = new SystemTest(i);
    }

    for (int i = 0; i < threadsNum; i++) {
        threads[i] = new Thread(ATMs[i]);
    }
    for (int i = 0; i < threadsNum; i++) {
        threads[i].start();
    }
    displayMenu();
}
```

System requirenments

1. Relational database
2. ATM Java Client
   a. Login using her account number and corresponding pin code
   b. Deposit funds
   c. Withdraw funds
   d. Display account balance
3. Concurrent Access
   a. Many ATM transactions happen in parallel
   b. The account balance must not be negative

Concurrent access and positive balance

When multiple users are updating the same data simultaneously may result in inconsistent data.

To keep ACID principles, positive balance and assure correct concurrent access a few steps were taken:

1. Server side:
   a. balance column is set to unsigned – as one of the requirements is to keep balance always positive
2. Client's side:
   a. For any UPDATE queries autocommit was turned off. In case of failure in any part of the transaction whole transaction was rolledback. Otherwise transaction is commited.
   b. One part of withdrawal transaction is balance check. If the balance is negative or if error is thrown by DB, whole withdrawal transaction is rolled back.
   c. To ensure full isolation *TRANSACTION_SERIALIZABLE* was turned on. Due to this anomalies such as dirty read, non-repeatable read and phantom read will not occur.
   d. To force DB to throw an error not just update balance column value with 0, SESSION.sql_mode = 'STRICT_TRANS_TABLES' is enabled as a session mode to each of the connections

Sum up

Double protection on a Java's clients side and DB's side protects positive balance. 2PC protocol is used[2], DB is on AWS with daily backup and transaction log[1] what makes the DB durable. All transaction are isolated due to full isolation – SERIALIZABLE mode, DB is consistent between many ATMs.

Taking into consideration all previously mentioned facts I can assume that my DB system fulfills all the requirements.

[1] https://dev.mysql.com/doc/refman/8.0/en/binary-log.html
[2] https://dev.mysql.com/doc/refman/8.0/en/xa.html