Question 1:

We close the connection to shut down any database resources that the connection is maintaining. Examples of resources like these would be handlers and cursors. If the connection was not closed, these feature would continue to run on the server and will be costly if left alone. There are a limited number of connections and leaving unused connection open is expensive in terms of efficiency for the database.

Question 2:

The input essentially launched an injection attack on our database. We dropped the Registration procedure from our database because part of the input was interpreted as additional commands in the query. This occurred since the input was not validated.

Question 3:

**public** ArrayList<SodaByRestaurant> getSodasByRestaurants(String rest, String soda, String price,

**boolean** useGreaterThanEqual) {

System.***out***.println(

**this**.dbService.getConnection() + " " + rest + " " + soda + " " + price + " " + useGreaterThanEqual);

**try** {

Statement stmt = **this**.dbService.getConnection().createStatement();

String query = "SELECT Restaurant, Soda, Manufacturer, RestaurantContact, Price \nFROM SodasByRestaurant\n";

**if** (price != **null** && price.length() != 0) {

PreparedStatement checkPrice = **null**;

query += "WHERE Price=?\n";

**try** {

**this**.dbService.getConnection().setAutoCommit(**false**);

checkPrice = **this**.dbService.getConnection().prepareStatement(query);

checkPrice.setString(1, price);

checkPrice.executeUpdate();

**this**.dbService.getConnection().commit();

} **catch** (SQLException e) {

e.printStackTrace();

**if** (**this**.dbService.getConnection() != **null**) {

**try** {

System.***err***.print("Invalid Input.");

**this**.dbService.getConnection().rollback();

} **catch** (SQLException excep) {

excep.printStackTrace();

;

}

}

} **finally** {

**if** (checkPrice != **null**) {

checkPrice.close();

}

**this**.dbService.getConnection().setAutoCommit(**true**);

}

}

ResultSet rs = stmt.executeQuery(query);

**return** parseResults(rs);

} **catch** (SQLException ex) {

JOptionPane.*showMessageDialog*(**null**, "Failed to retrieve sodas by restaurant.");

ex.printStackTrace();

**return** **new** ArrayList<SodaByRestaurant>();

}

}

Question 4:

Exception: Faulty input. Try again.com.microsoft.sqlserver.jdbc.SQLServerException: The server failed to resume the transaction. Desc:3e00000001.

Reason: The register procedure remained in the database on the second try. The output was an exception that said, “Faulty input, try again”. We bypassed the injection attempt by converting the total input into a string which could then not be interpreted as part of the query.

Question 5:

**public** ArrayList<SodaByRestaurant> getSodasByRestaurants(String rest, String soda, String price, boolean useGreaterThanEqual) {

Double parsePrice=**null**;

**if**(price.length()==0) {

price=**null**;

}**else** {

parsePrice = Double.*valueOf*(price);

}

**int** i=1;

String query = buildParameterizedSqlStatementString(rest, soda, parsePrice, useGreaterThanEqual);

**try** {

PreparedStatement statement = **null**;

**try** {

**this**.dbService.getConnection().setAutoCommit(**false**);

statement = **this**.dbService.getConnection().prepareStatement(query);

**if**(rest!=**null**) {

statement.setString(i, rest);

i++;

}

**if**(soda!=**null**) {

statement.setString(i, soda);

i++;

}

**if** (saferPrice != **null** && price.length() != 0) {

statement.setDouble(i, parsePrice);

}

ResultSet rs=statement.executeQuery();

**this**.dbService.getConnection().commit();

**return** parseResults(rs);

} **catch** (SQLException e) {

e.printStackTrace();

**if** (**this**.dbService.getConnection() != **null**) {

**try** {

System.***err***.print("Invalid Input. Try again.");

**this**.dbService.getConnection().rollback();

} **catch** (SQLException excep) {

excep.printStackTrace();

;

}

}

} **finally** {

**if** (statement != **null**) {

statement.close();

}

**this**.dbService.getConnection().setAutoCommit(**true**);

}

}**catch**(SQLException ex){

JOptionPane.*showMessageDialog*(**null**, "Failed to retrieve sodas by restaurant.");

ex.printStackTrace();

**return** **new** ArrayList<SodaByRestaurant>();

}

**return** **new** ArrayList<SodaByRestaurant>();

}

Question 6:

The procedure remained in the database. By parsing the price, we validated the input and made it so there cannot be an injection attack since the input can only be a double. Strings or other characters that could not be parsed into a Double will cause a NumberFormatException that would not let the query begin to execute and thus would not let it even send anything to the database. This is because price would be a local variable in our Java GUI and would have the error of an invalid input handled locally before it reaches the database.

Question 7: For the functionality that we need for the GUI, db\_owner has unnecessary privileges that give the user authorities that they don’t need. The db\_owner can drop the database, tables, and stored procedures which is not something that we really want to happen to this project especially from the GUI. As such, it makes sense to remove such privilege. On the other hand, we do want to be able to modify the database. This is where the **db\_datawriter** is a necessary privilege. They can add, remove, or change data in the database. In this way, we can still update the database, but don’t risk losing tables We also want to be able to view the data in the database which is the reason for **db\_datareader.**

Question 8:

No, the **db\_datawriter has the ability to remove data from the tables which means that they can still remove information the tables in the database. Restricting permissions** limits the amount of damage an individual can wreak on a database. The **db\_datawriter can’t drop the database and only has permission to modify the data in the database. Thus, they cannot destroy the structure of the database, but they can still mess it up a lot if they messed with the data.**

Question 9:

Because passwords are generally sensitive information. If they weren’t salted and hashed, anyone with access to the database would have access to every user’s password. It is a huge security/privacy risk that can be fixed by making passwords more difficult to decode. Even if someone managed to find a weak point in the system and gain access to user information, they would be hard pressed to decode a salted and hashed password to gain access to the individuals account.

Question 10:

We don’t use the same salt for security issues. If everyone had the same salt, one individual might figure out the salt and now have the password of every account on the project. Using different salts eliminates this risk and secures every individual’s account and prevents other forms of attacks like rainbow table attacks.

Question 11:

Because we don’t want to give specific errors which might give an individual who is trying to get into an account more information about specific accounts. From this insight, They can then utilize the information to eliminate possibilities that keep accounts secure.

Question 12:

For data security. Even if an individual manages to gain access to a server, they wont have direct access to the database if they don’t have the encryption key.