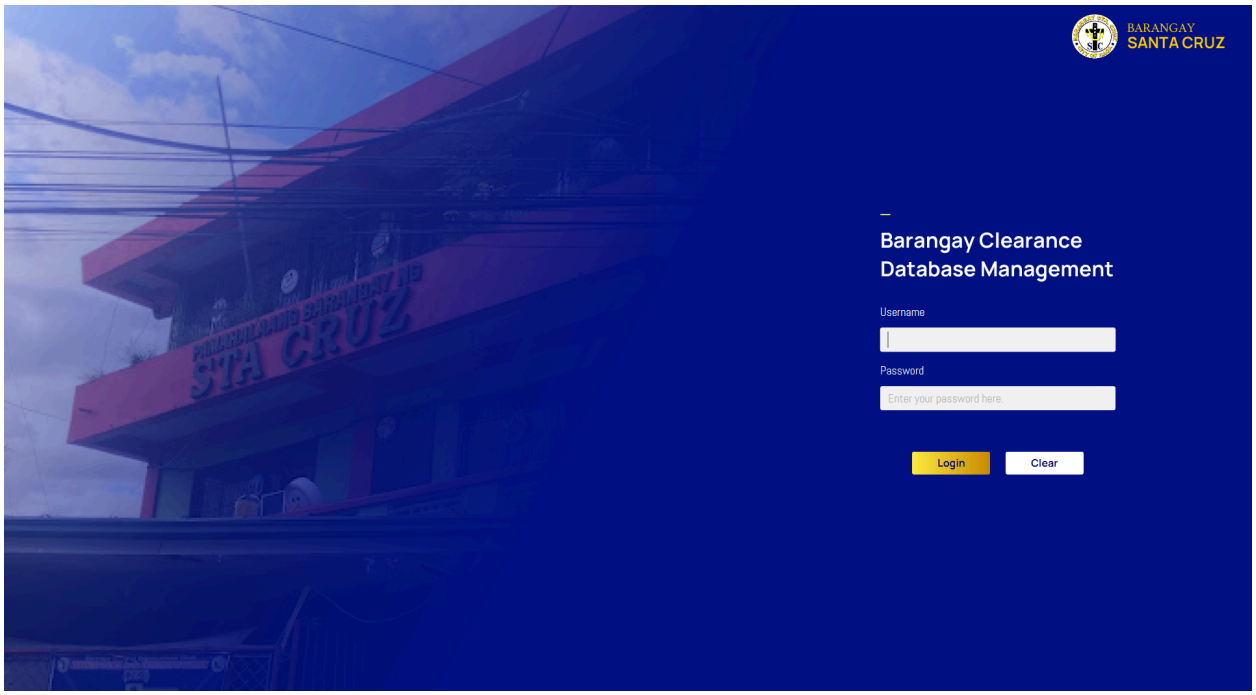


Business Clearance Database Management System for Barangay Sta. Cruz

Object Oriented Programming Final Project Documentation

—

Adrian Joshua M. Reapor

University of Nueva Caceres

Overview

This is the documentation of my final project submission for the Object Oriented Programming (OOP) course, which is to create a DBMS for a selected community.

The community my group has selected to help is *Barangay Sta. Cruz*. However, as the person that does not share the *Database Management I* course with the rest of my group, I am submitting this project separately for my OOP project, as my requirements for the Database section are different.

Technology Stack

This project uses the following technologies:

- **Java 11**
  - The programming language used in this project.
- **JavaFX 13**
  - The software platform used to build the graphics interface of the project.
- **SceneBuilder 23.0.1**
  - The visual layouting software used for building the JavaFX graphical interface.
- **ControlFX**
  - Imported JavaFX library for additional UI Control components
- **MySQL Server 8.0**
  - The relational database management system used in the project.
- **Visual Studio Code 1.96.2 (VSCode)**
  - The IDE used to program the project
- **Maven (as extension of VSCode)**
  - The build automation tool used to build and serve the Java project with JavaFX in VsCode.
- **Codeium (as extension of VSCode)**
  - The AI Code assistant platform used for troubleshooting and boilerplating.

Modules

The modules this project contains are the following:

1. **User Management Module** - A user login interface that restricts the access of the following module.
2. **Data Entry Module (Business Clearance)** - A CRUD (Create, Read, Update, Delete) interface for a Business Clearance database, based on the Business Clearance Form provided by Barangay Sta. Cruz.

Project Architecture

1. Model (Database Tables)

User Account Table

The structure of the model of the **User Management Module** was provided during the OOP laboratory lectures, as shown below. The attributes *username, password, lastname, firstname* and *middlename* are self-explanatory. Meanwhile, the *user_id* attribute is the automatically generated primary key that identifies the individual **user account** tuple, while the *status* attribute indicates whether the account is visible (value is **1**) in the database, or if the tuple was *"deleted"* (value is **0**) and is thus invisible to queries.

user_account		
PK	user_id	integer
	username	varchar(20)
	password	varchar(20)
	lastname	varchar(25)
	firstname	varchar(25)
	middlename	varchar(25)
	status	integer

Figure 1: Table Diagram of the **User Account** model from the OOP Lab Class

From this diagram we could easily make a table that satisfies the value constraints. The query used to initialize such a table is shown in **Figure 2**. Note that the '*IF NOT EXISTS*' keyword so that the query can be run even after the table has already been created.

User Account Table

Meanwhile, the structure of the model for the **Data Entry Module** would be dependent on the forms provided by our community, which is Barangay Sta. Cruz. While the barangay has not given us direct physical forms, they instead have provided us another barangay website which they use as basis for their forms[1]. From which I selected the **Barangay Business Clearance Form** for my Data Entry Module.

```
CREATE TABLE IF NOT EXISTS `user_account` (
    `user_id` INT NOT NULL AUTO_INCREMENT,
    `username` VARCHAR(20) NOT NULL,
    `password` VARCHAR(20) NOT NULL,
    `lastname` VARCHAR(25) NOT NULL,
    `firstname` VARCHAR(25) NOT NULL,
    `middlename` VARCHAR(25) NOT NULL,
    `status` INT NOT NULL DEFAULT 1,
    PRIMARY KEY (`user_id`),
    UNIQUE INDEX `username_UNIQUE` (`username` ASC) VISIBLE
);
```

Figure 2: MySQL Query to create the **user_account** table

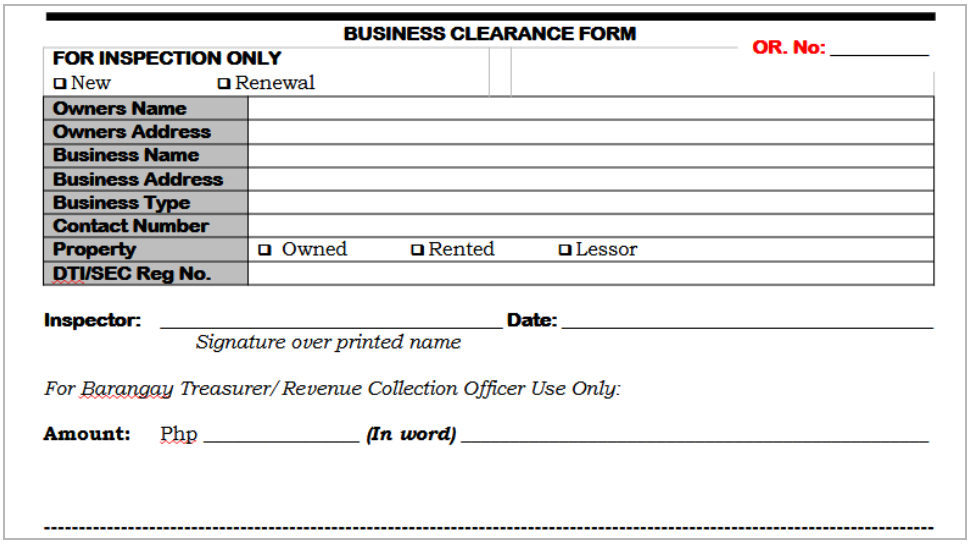
The **Barangay Business Clearance Form** module has an online form[2], and a document that a visitor could print and fill out on their own before submitting to the barangay for a personal application, as shown in **Figure 3**.

Mapping the inputs of the business clearance form to their equivalent attribute, we can create the diagram for the Business Clearance entity, as shown in **Figure 4**. We set a *transaction_id* attribute as the automatically generated primary key that identifies the individual **business clearance transaction** tuple, and a *status* key similar to the *status* attribute of the **User Account** table.

The *DTI/SEC Reg. No.* is also encoded as the *registration_number* attribute instead. We also instead stored the *contact_number*, *registration_number*, and *official_receipt_number*

[1] https://barangaybagbag.com/services/
[2] https://barangaybagbag.com/barangay-clearance-form/

attributes as **varchar** instead of **integer** due to the nominal functionality of these attributes: their values are used only for identification purposes, are not used for mathematical computation, and are even at the risk of having values larger than the maximum integer value. Lastly, the amount is stored in **decimal** with **2** decimal digits to represent the Peso currency.



BUSINESS CLEARANCE FORM OR. No: _____

FOR INSPECTION ONLY
☐ New ☐ Renewal

Owners Name	
Owners Address	
Business Name	
Business Address	
Business Type	
Contact Number	
Property	☐ Owned ☐ Rented ☐ Lessor
DTI/SEC Reg No.	

Inspector: _____ **Date:** _____
Signature over printed name

For Barangay Treasurer/ Revenue Collection Officer Use Only:

Amount: Php _____ ***(In word)*** _____

Figure 3: The **Business Clearance Form** from the Barangay provided Website

business_clearance_transaction		
PK	transaction_id	integer
	inspection_type	enum('new','renewal')
	owner	varchar(75)
	owner_address	varchar(255)
	business_name	varchar(255)
	business_address	varchar(255)
	business_type	varchar(25)
	contact_number	varchar(20)
	property_type	enum('owned', 'rented','lessor')
	registration_number	varchar(25)
	inspector	varchar(75)
	inspection_date	date
	amount	decimal(10,2)
	official_receipt_number	varchar(25)
	status	integer

Figure 4: Table Diagram of a **Business Clearance Transaction** entity based from the Business Clearance Form

The MySQL query that can be used to make this table is also shown in **Figure 5**.

```
CREATE TABLE IF NOT EXISTS `business_clearance_transaction` (
    `transaction_id` INT NOT NULL AUTO_INCREMENT,
    `inspection_type` ENUM('new', 'renewal') NOT NULL,
    `owner` VARCHAR(255) NOT NULL,
    `owner_address` VARCHAR(75) NOT NULL,
    `business_name` VARCHAR(255) NOT NULL,
    `business_address` VARCHAR(255) NOT NULL,
    `business_type` VARCHAR(25) NOT NULL,
    `contact_number` VARCHAR(20) NOT NULL,
    `property_type` ENUM('owned', 'rented', 'lessor') NOT NULL,
    `registration_number` VARCHAR(25) NOT NULL,
    `inspector` VARCHAR(75) NOT NULL,
    `inspection_date` DATE NOT NULL,
    `amount` DECIMAL(10,2) NOT NULL,
    `official_receipt_number` VARCHAR(25) NOT NULL,
    `status` INT NOT NULL DEFAULT 1,
    PRIMARY KEY (`transaction_id`)
);
```

Figure 5: MySQL Query to create the **business_clearance_transaction** table

2. Model (Java Classes)

To turn these ER Diagrams to their equivalent Java models, we can imagine that each table represents an equivalent Java Class, with each column representing a class property.

The UML Diagram of the Database is shown in **Figure 6**.

While the conversion of the **User Account** table is straightforward, the **Business Clearance Transaction** table is a complex entity with values that do not directly map with Java's primitive types. The ENUM values of the *inspection_type* and *property_type* attributes are represented by actual Java Enums. The DATE value is represented by the *java.time.LocalDate* class as it can store a Date value. And the *amount* attribute uses the java class BigDecimal instead of *double* or *float* to preserve the precision.

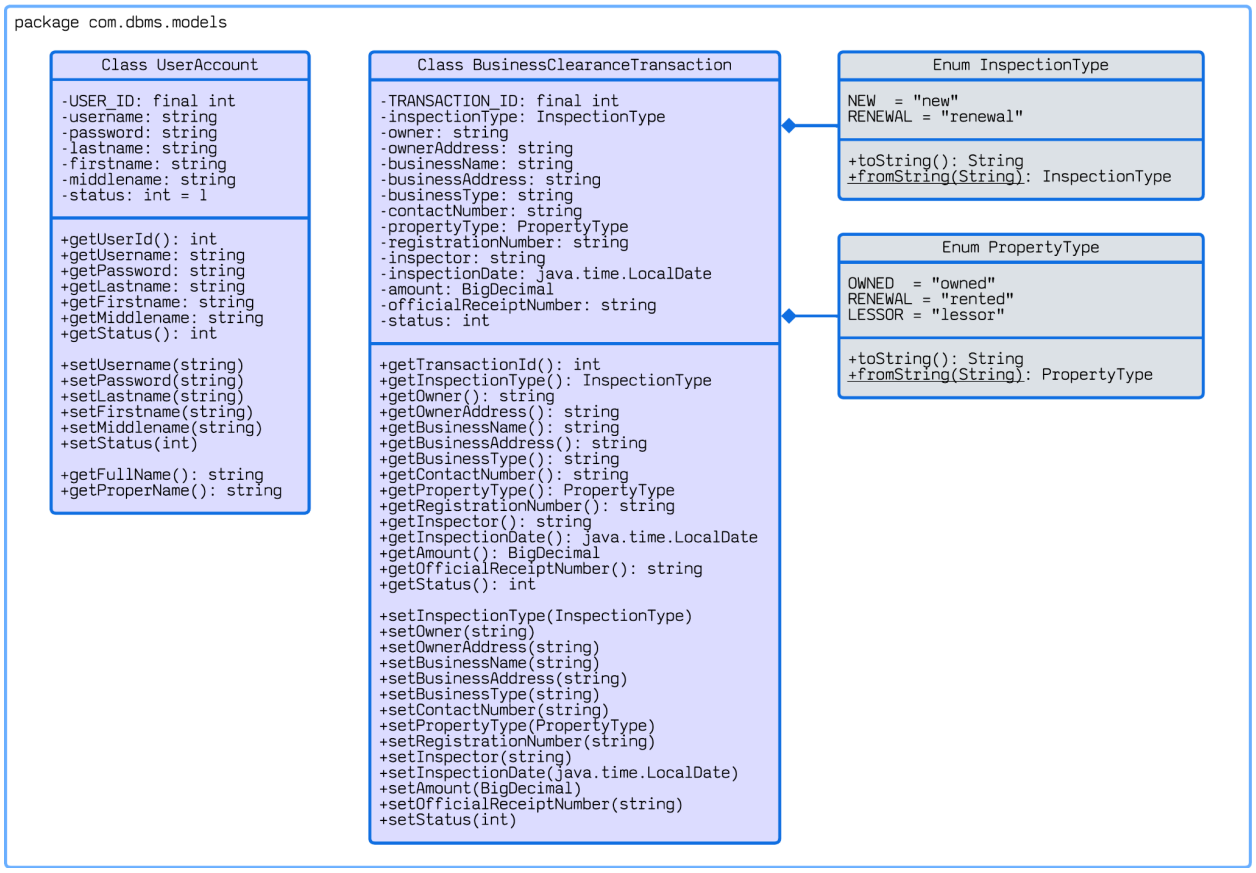


Figure 6: UML Diagrams of the **user_account** and **business_clearance_transaction** tables

3. Views

User Login View

The landing view that would first be seen is the User Login view shown in **Figure 7.** It is a straightforward login page composed by **Username** and **Password** fields. The authentication is directly compared to contents in the **user_account** table, so while the database initialization could rebuild the table after it is dropped, you might not be able to login through this view.

User Management

A workaround used is simply switching the first scene to load to **User Management View** in **Figure 8**. From here you can see the table of all the active accounts in the database, and general database management functions. You can add new data, edit or delete existing data, or even do a search on selected columns.

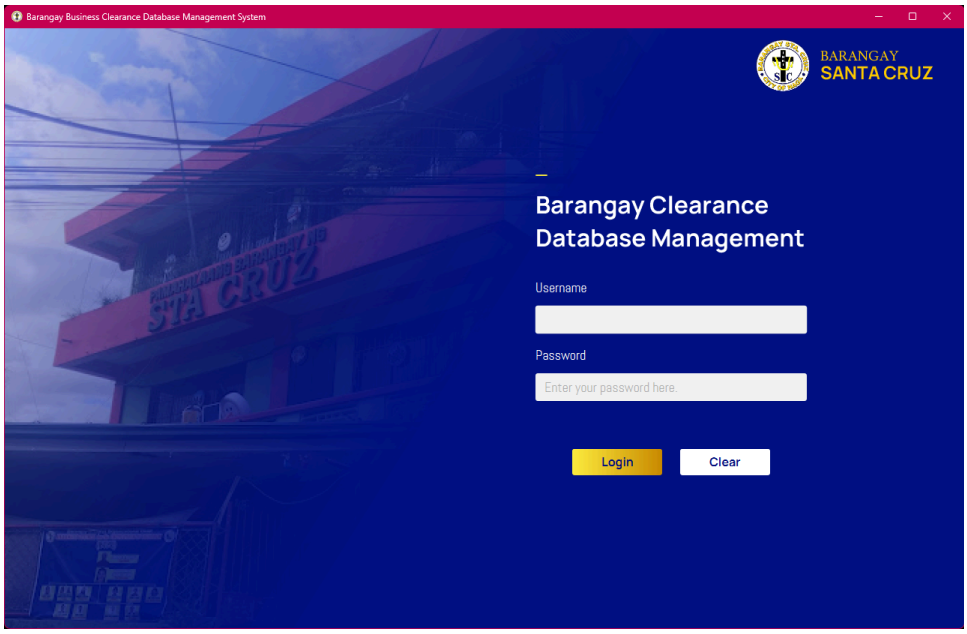


Figure 7: **User Login View**

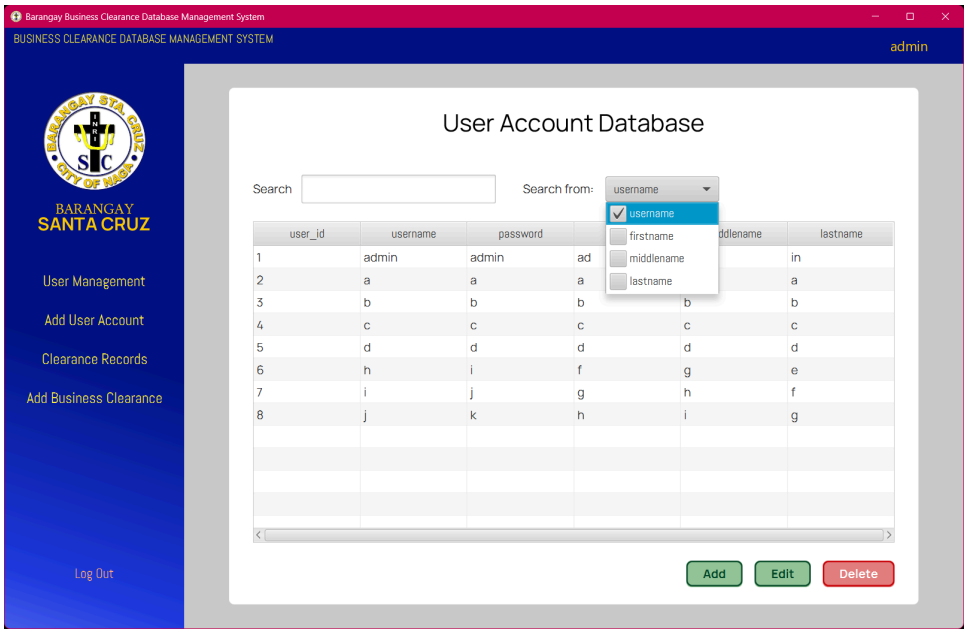


Figure 8: **Main View** and **User Management Views** together

Well, **Figure 8** is not necessarily just the **Main View**, for it is actually a combination of the **Main View** and **User Management** views showing at once. This is because the **Main View** only consists of the navigation bar at the left and the title header, while the **User Management** scene is dynamically loaded into the Center of the Border Pane in the **Main View.**

These two views when separated are shown in **Figures 9** and **10**. As you can see, the username at the header is dynamically loaded from who logged in, as indicated by the **admin** name in Figure 8.

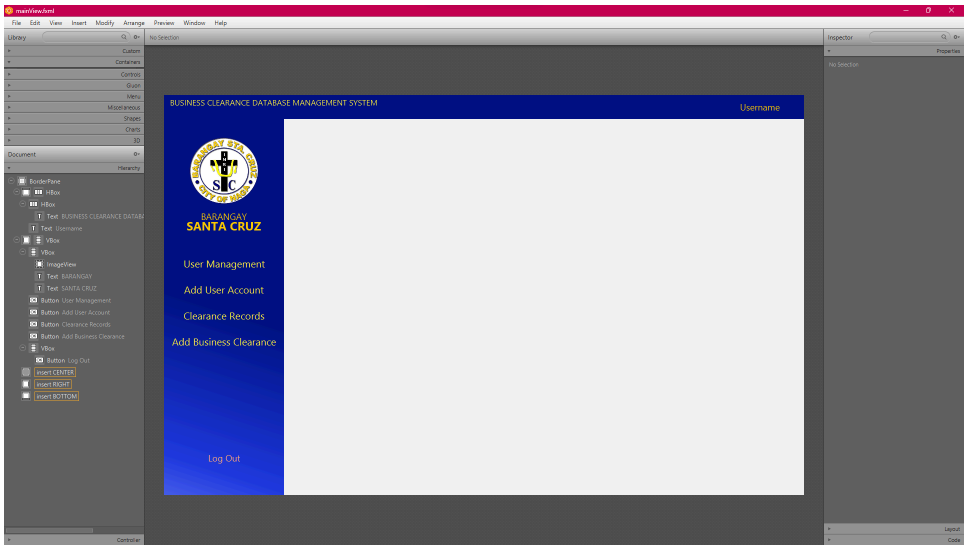


Figure 9: **Main View** in SceneBuilder

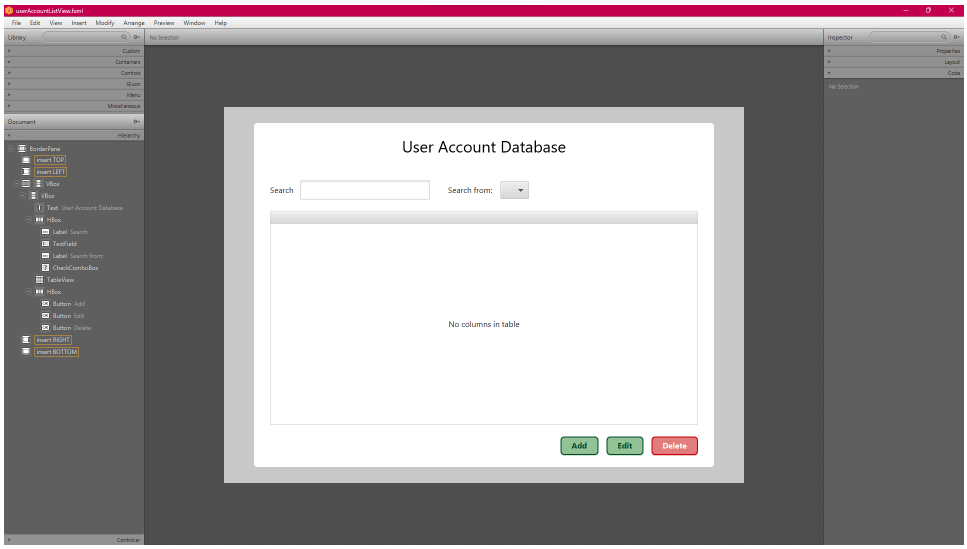


Figure 10: **User Management View** in SceneBuilder

User Account Entry View

Either by pressing the *Add User Account* item in the navigational menu, or either the *Add* or *Edit* buttons from the **User Management View**, one can open the **User Account Entry** view shown in **Figures 11-13.**

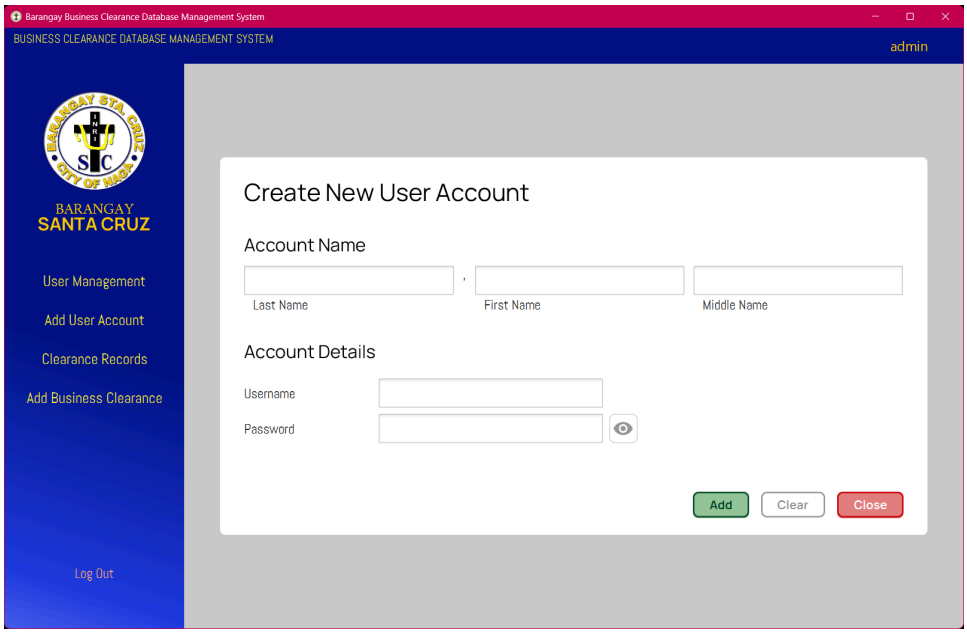


Figure 11: Adding a new account using **User Account Entry**

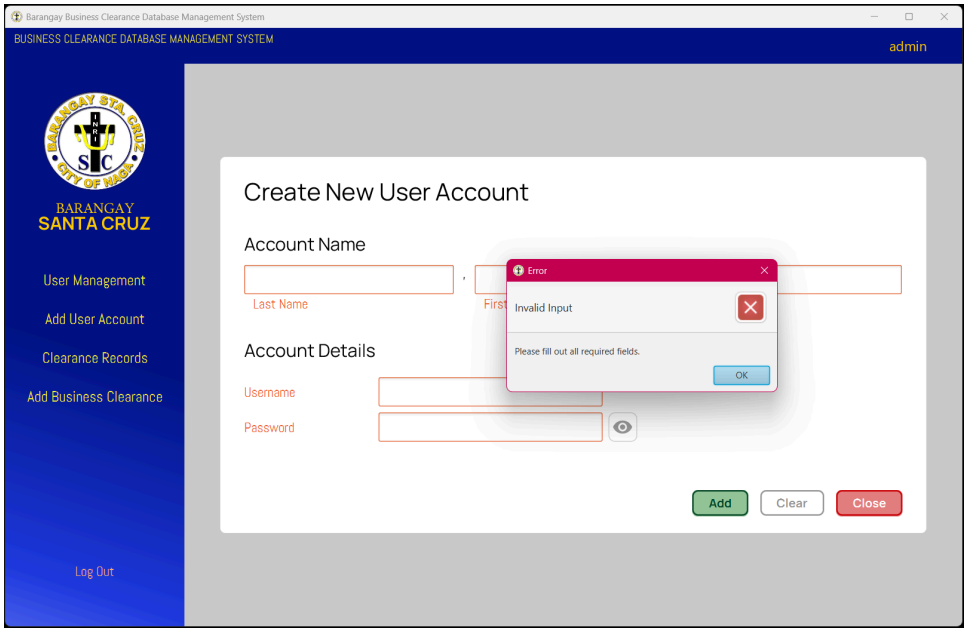


Figure 12: Required fields have to be filled

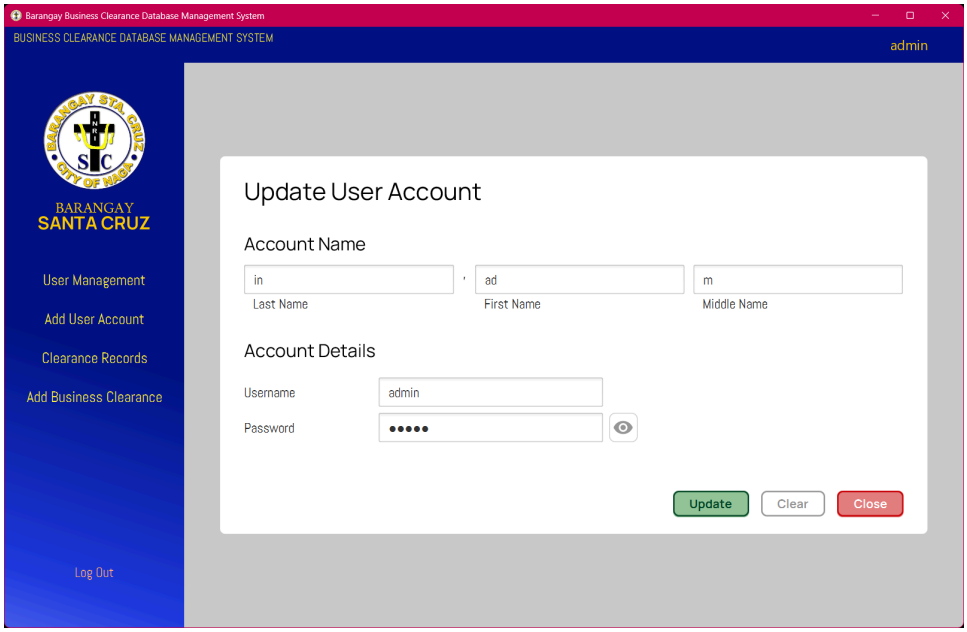


Figure 13: Loading an existing account in **User Account Entry**

Business Clearance Transaction List View

This view is similar to the **User Account** list view as shown in **Figures 14-15**, if a bit more occupied due to how much more data there is needed for a business clearance transaction.

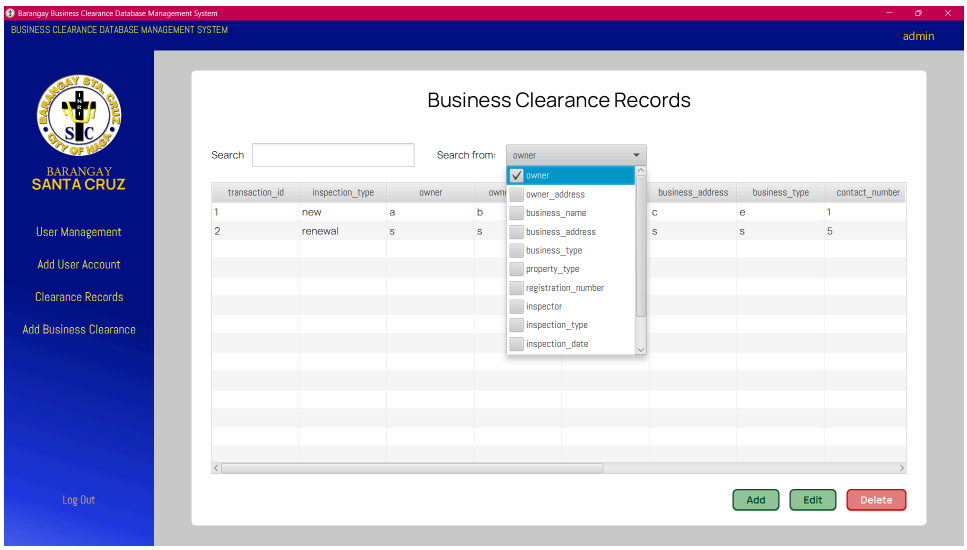


Figure 14: Loading an existing account in **User Account Entry**

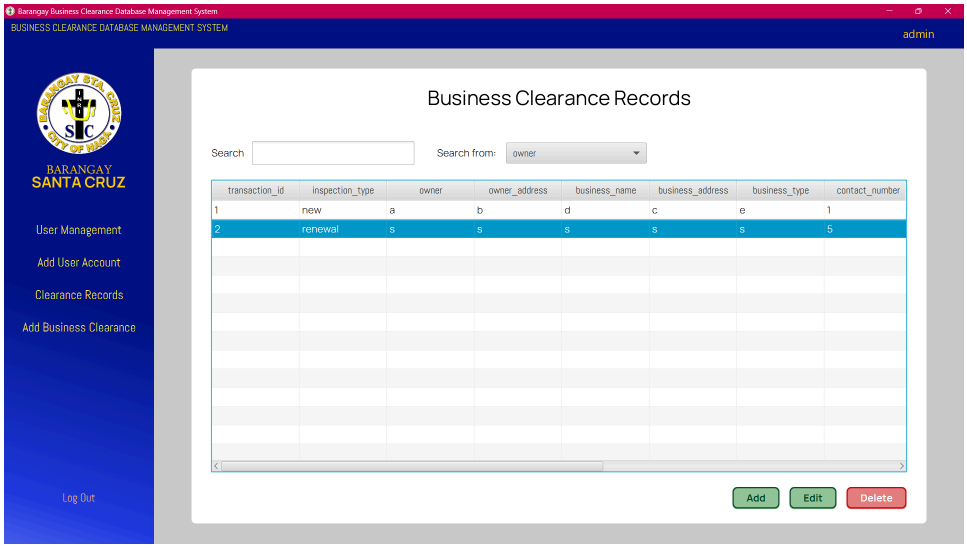


Figure 15: Selecting individual transactions from the table

Business Clearance Transaction Entry View

Expanding from the styling of the **User Account Entry View**, this view is quite lengthy that it is wrapped inside a *Scroll Pane*, as shown in **Figures 16-17**.

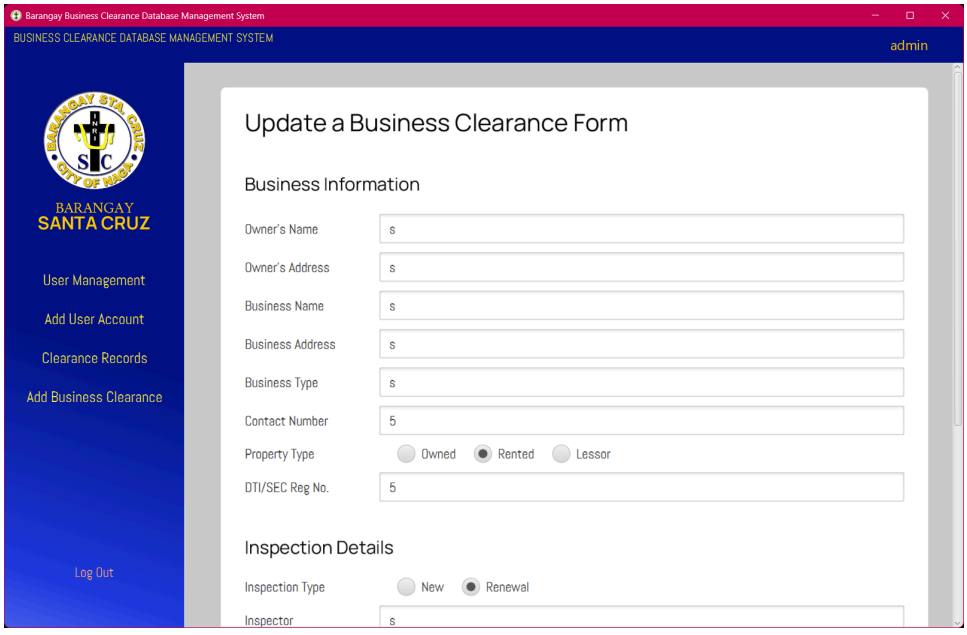


Figure 16: Top of the **User Account Entry View**, editing the selected transaction from Figure 15

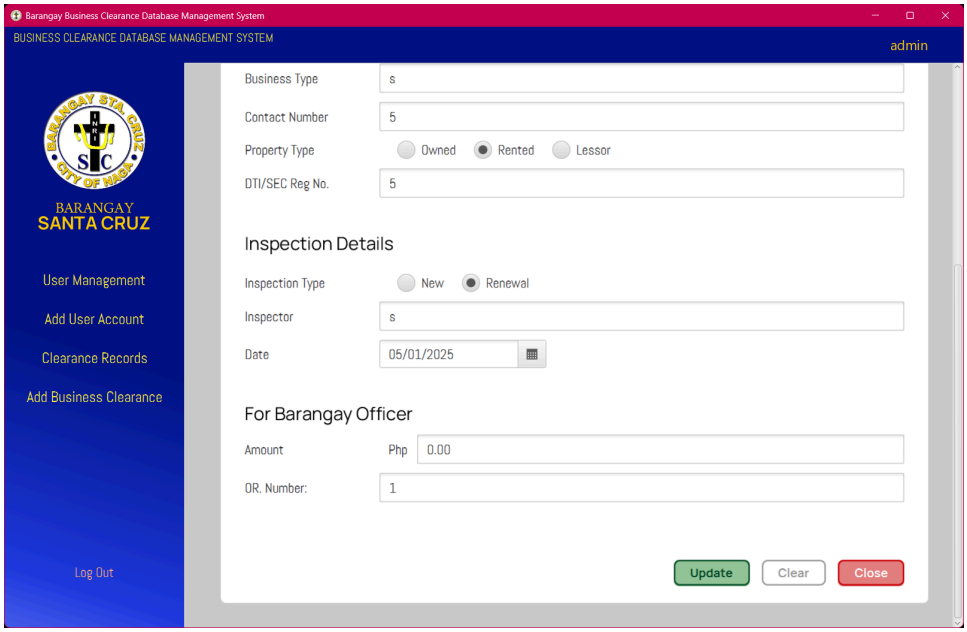


Figure 17: Bottom of the **User Account Entry View**

4. Database

Database Configuration

```
oop_dbms > src > main > java > com > dbms > database > DBConnection.java > ...
1   package com.dbms.database;
2
3   import java.sql.Connection;
4   import java.sql.DriverManager;
5   import java.sql.SQLException;
6
7   // Utility class to connect to the database
8   public class DBConnection {
9       static private final String DB_URL = "jdbc:mysql://localhost:3309/";
10      static private final String USER = "root";
11      static private final String PASS = "NotCat24";
12      static private final String DB_NAME = "oop_dbms";
13
14      // Returns a connection to the database
15      protected static Connection getConnection() throws SQLException {
16          return DriverManager.getConnection(DB_URL, USER, PASS);
17      }
18
19      // Returns the database name
20      protected static String getDBName() {
21          return DB_NAME;
22      }
23  }
24
```

Figure 18: **DBConnection Class**

Due to time constraints, the MySQL server is only hosted locally as of now with the configurations hardcoded in the database interface class **DBConnection** shown in **Figure 18**. These configurations may have to be modified to fit your local MySQL server configuration.

Database Interface

A summary of the database classes used to bridge the Controllers to the database are shown in Figure 19. The **DBConnection** class is universally used by the other database classes to connect to the database server.

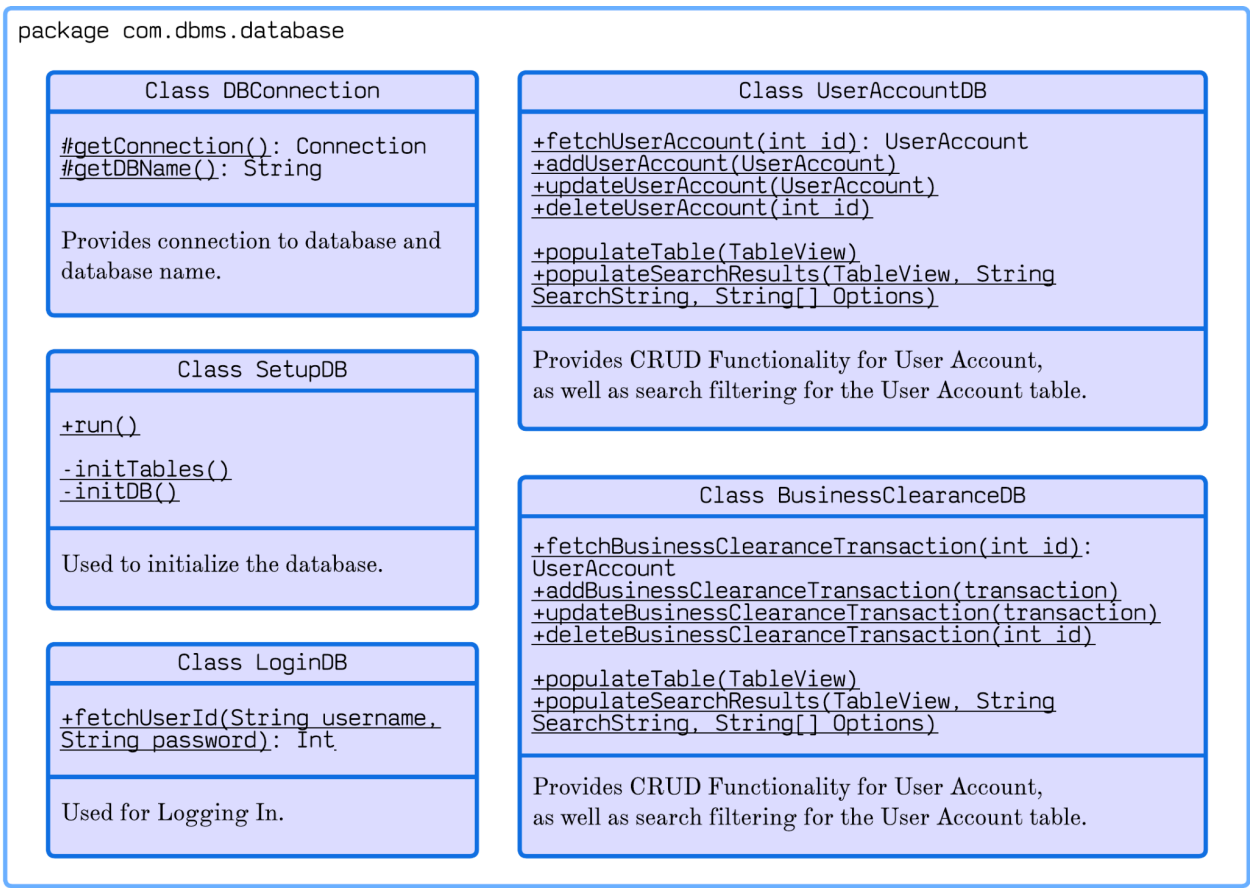


Figure 19: **Database Interface Classes**

The **SetupDB** is run to initialize the database for the session, creating the database and tables if they weren't initialized before.

The **LoginDB** is used to authenticate the login credentials from **User Login View.**

Meanwhile the **UserAccountDB** and **BusinessClearanceDB** classes manage all the CRUD functionality in the **data entry views** and the table management for the **list views**.

SetupDB.run()

As shown in **Figure 20**, the main setup step is ultimately just calling for the **initDB()**, the **createUserAccountTable()** and the **createBusinessClearanceTransactionTable()** helper methods.

The **initDB()** method is a simple *CREATE DATABASE* query with the *IF NOT EXISTS* so it would run even if the database already exists. This query calls for the database name configured in the **DBConnection** Class through the **getDBName()** method. Unfortunately, one cannot use **PreparedStatement.setString()** to securely insert the database name, so it is only appended through string concatenation.

Meanwhile the other helper methods **createUserAccountTable()** and **createBusinessClearanceTransactionTable()**, shown in **Figures 21** and **22**, are actually using the MySQL queries from **Figures 2** and **5** to create the respective tables to store the **User Account** and **Business Clearance Transaction** entities.

```
// Initializes the database and tables
public static void run() throws SQLException {
    initDB();
    initTables();
}

// Initializes the tables
private static void initTables() throws SQLException {
    createUserAccountTable();
    createBusinessClearanceTransactionTable();
}

// Initializes the database
private static void initDB() throws SQLException {
    String dbName = DBConnection.getDBName();
    String query = "CREATE DATABASE IF NOT EXISTS " + dbName + " ;";
    Connection conn = DBConnection.getConnection();
    PreparedStatement stmt = conn.prepareStatement(query);

    stmt.execute();
    conn.close();
}
```

Figure 20: Head of the contents of the **run()** method

```
// Initializes the `user_account` table
private static void createUserAccountTable() throws SQLException {
    String query = "CREATE TABLE IF NOT EXISTS `" + DBConnection.getDBName() + "`.
    `user_account` (\n" +
            "  `user_id` INT NOT NULL AUTO_INCREMENT,\n" +
            "  `username` VARCHAR(20) NOT NULL,\n" +
            "  `password` VARCHAR(20) NOT NULL,\n" +
            "  `lastname` VARCHAR(25) NOT NULL,\n" +
            "  `firstname` VARCHAR(25) NOT NULL,\n" +
            "  `middlename` VARCHAR(25) NOT NULL,\n" +
            "  `status` INT NOT NULL DEFAULT 1,\n" +
            "  PRIMARY KEY (`user_id`),\n" +
            "  UNIQUE INDEX `username_UNIQUE` (`username` ASC) VISIBLE\n" +
            ");";
    Connection conn = DBConnection.getConnection();
    PreparedStatement stmt = conn.prepareStatement(query);
    stmt.execute();
    conn.close();
}
```

Figure 21: Body of the **createUserAccountTable()** method

```
// Initializes the `business_clearance_transaction` table
private static void createBusinessClearanceTransactionTable() throws SQLException {
    String query = "CREATE TABLE IF NOT EXISTS `" + DBConnection.getDBName()
            + "`.`business_clearance_transaction` (\n" +
            "  `transaction_id` INT NOT NULL AUTO_INCREMENT,\n" +
            "  `inspection_type` ENUM('new', 'renewal') NOT NULL,\n" +
            "  `owner` VARCHAR(255) NOT NULL,\n" +
            "  `owner_address` VARCHAR(75) NOT NULL,\n" +
            "  `business_name` VARCHAR(255) NOT NULL,\n" +
            "  `business_address` VARCHAR(255) NOT NULL,\n" +
            "  `business_type` VARCHAR(25) NOT NULL,\n" +
            "  `contact_number` VARCHAR(20) NOT NULL,\n" +
            "  `property_type` ENUM('owned', 'rented', 'lessor') NOT NULL,\n" +
            "  `registration_number` VARCHAR(25) NOT NULL,\n" +
            "  `inspector` VARCHAR(75) NOT NULL,\n" +
            "  `inspection_date` DATE NOT NULL,\n" +
            "  `amount` DECIMAL(10,2) NOT NULL,\n" +
            "  `official_receipt_number` VARCHAR(25) NOT NULL,\n" +
            "  `status` INT NOT NULL DEFAULT 1,\n" +
            "  PRIMARY KEY (`transaction_id`)\n" +
            ");";
    Connection conn = DBConnection.getConnection();
    PreparedStatement stmt = conn.prepareStatement(query);

    stmt.execute();
    conn.close();
}
```

Figure 22: Body of the **createBusinessClearanceTransactionTable()** method

LoginDB.fetchUserId()

The account validation calls for a simple query that looks for the **user id** of the one active (status=1) **user_account** with the passed *username* and *password* values, as shown in **Figure 23**.

```java
// returns user_id, or -1 if not found
public static int fetchUserId(String username, String password) throws SQLException {
    int ret = -1;

    String query = "SELECT user_id FROM `" + DBConnection.getDBName() + "`.
    `user_account` WHERE username = '" + username
            + "' AND password = '" + password
            + "' AND status = 1";

    Connection conn = DBConnection.getConnection();
    PreparedStatement stmt = conn.prepareStatement(query);

    ResultSet rs = stmt.executeQuery();
    if (rs.next()) {
        ret = rs.getInt(columnLabel:"user_id");
    }
    conn.close();

    return ret;
}
```

Figure 23: Body of the **fetchUserId()** method

Entity Fetch Methods

The functionalities and methods between the database classes **UserAccountDB** and **BusinessClearanceDB** fairly parallel each other, with the latter only a bit more lengthy than the former due to the larger attribute count. So from here on, similar methods would be explored in pairs. We are also shortening the database to **userDB** and **txnDB** (*txn* for *transaction*) for this section.

```java
// Retrieves a user account from the database using the given user ID.
public static UserAccount fetchUserAccount(int userId) throws SQLException {

    UserAccount userAccount = null;

    String query = "SELECT * FROM `" + DBConnection.getDBName() + "`.`user_account` WHERE user_id = "
            + userId + ";";
    Connection conn = DBConnection.getConnection();
    PreparedStatement stmt = conn.prepareStatement(query);
    ResultSet rs = stmt.executeQuery();

    if (rs.next()) {
        int USER_ID = rs.getInt(columnLabel:"user_id");
        String username = rs.getString(columnLabel:"username");
        String password = rs.getString(columnLabel:"password");
        String lastname = rs.getString(columnLabel:"lastname");
        String firstname = rs.getString(columnLabel:"firstname");
        String middlename = rs.getString(columnLabel:"middlename");

        userAccount = new UserAccount(USER_ID, username, password, lastname, firstname, middlename);
    }
    conn.close();

    return userAccount;
}
```

Figure 24: Body of the **fetchUserAccount()** method

userDB.fetchUserAccount() and **txnDB.fetchBusinessClearanceTransaction()** methods, as shown in **Figures 24** and **25**, both take in their respective *id* attributes *userId* and *transactionId*, and creates a *SELECT WHERE* String *query* matching those *id* values. This query is then turned into a **PreparedStatement** *stmt* using the **Connection** *conn* provided by the **DBConnection** class.

Executing the *stmt* returns a **ResultSet** as the database's response to the query. The ResultSet would be extracted for its information, then creating the instance of the either **UserAccount** and **BusinessClearanceTransaction** that would be returned.

```
// fetch business clearance transaction
public static BusinessClearanceTransaction fetchBusinessClearanceTransaction(int transactionId)
        throws SQLException {
    BusinessClearanceTransaction transaction = null;
    String query = "SELECT * FROM `" + DBConnection.getDBName()
            + "`.`business_clearance_transaction` WHERE transaction_id = "
            + transactionId + ";";

    Connection conn = DBConnection.getConnection();
    PreparedStatement stmt = conn.prepareStatement(query);

    ResultSet rs = stmt.executeQuery();

    if (rs.next()) {
        int TRANSACTION_ID = rs.getInt(columnLabel:"transaction_id");
        InspectionType inspectionType = InspectionType.fromString(rs.getString(columnLabel:"inspection_type"));
        String owner = rs.getString(columnLabel:"owner");
        String ownerAddress = rs.getString(columnLabel:"owner_address");
        String businessName = rs.getString(columnLabel:"business_name");
        String businessAddress = rs.getString(columnLabel:"business_address");
        String businessType = rs.getString(columnLabel:"business_type");
        String contactNumber = rs.getString(columnLabel:"contact_number");
        PropertyType propertyType = PropertyType.fromString(rs.getString(columnLabel:"property_type"));
        String registrationNumber = rs.getString(columnLabel:"registration_number");
        String inspector = rs.getString(columnLabel:"inspector");
        LocalDate inspectionDate = rs.getDate(columnLabel:"inspection_date").toLocalDate();
        BigDecimal amount = rs.getBigDecimal(columnLabel:"amount");
        String officialReceiptNumber = rs.getString(columnLabel:"official_receipt_number");

        transaction = new BusinessClearanceTransaction(TRANSACTION_ID, inspectionType, owner,
                ownerAddress, businessAddress, businessName, businessType, contactNumber,
                propertyType,
                registrationNumber, inspector, inspectionDate, amount, officialReceiptNumber);
    }
    conn.close();

    return transaction;
}
```

Figure 25: Body of the **fetchBusinessClearanceTransaction()** method

Add Entity Methods

The **userDB.addUserAccount()** and **txnDB.addBusinessClearanceTransaction()** are simply the reverse of the Entity Fetching methods, but this time we get to use the **PreparedStatement**'s **setString()** method. As shown in **Figures 26** and **27**, the **setString()** allows for a more secure value substitution into the query string than direct string concatenation.

The conversion of the attributes from the **User Account** entity is straightforward as all of them are already strings. Meanwhile the **BusinessClearanceTransaction** have enum types (parameter indices of 1 and 8), LocalDate (11) and BigDecimal (12) that have to be converted to string. Remember from **Figure 6** that the enum types **InspectionType** and **PropertyType** have a **toString()** method which returns their equivalent string values.

```
// Adds a new user account to the database.
public static void addUserAccount(UserAccount userAccount) throws SQLException {
    String query = "INSERT INTO `" + DBConnection.getDBName()
            + "`.`user_account` (`username`, `password`, `lastname`, `firstname`,
            `middlename`) VALUES (?, ?, ?, ?, ?)";

    Connection conn = DBConnection.getConnection();
    PreparedStatement stmt = conn.prepareStatement(query);

    stmt.setString(parameterIndex:1, userAccount.getUsername());
    stmt.setString(parameterIndex:2, userAccount.getPassword());
    stmt.setString(parameterIndex:3, userAccount.getLastname());
    stmt.setString(parameterIndex:4, userAccount.getFirstname());
    stmt.setString(parameterIndex:5, userAccount.getMiddlename());

    stmt.execute();
    conn.close();
}
```

Figure 26: Body of the **addUserAccount()** method

```
// add business clearance transaction
public static void addBusinessClearanceTransaction(BusinessClearanceTransaction
transaction)
        throws SQLException {
    String query = "INSERT INTO `" + DBConnection.getDBName()
            + "`.`business_clearance_transaction` ("
            + "`inspection_type`, `owner`, `owner_address`, `business_address`,
            `business_name`, `business_type`, `contact_number`, "
            + "`property_type`, `registration_number`, `inspector`,
            `inspection_date`, `amount`, `official_receipt_number`"
            + ") VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?);";

    Connection conn = DBConnection.getConnection();
    PreparedStatement stmt = conn.prepareStatement(query);

    stmt.setString(parameterIndex:1, transaction.getInspectionType().toString());
    stmt.setString(parameterIndex:2, transaction.getOwner());
    stmt.setString(parameterIndex:3, transaction.getOwnerAddress());
    stmt.setString(parameterIndex:4, transaction.getBusinessAddress());
    stmt.setString(parameterIndex:5, transaction.getBusinessName());
    stmt.setString(parameterIndex:6, transaction.getBusinessType());
    stmt.setString(parameterIndex:7, transaction.getContactNumber());
    stmt.setString(parameterIndex:8, transaction.getPropertyType().toString());
    stmt.setString(parameterIndex:9, transaction.getRegistrationNumber());
    stmt.setString(parameterIndex:10, transaction.getInspector());
    stmt.setString(parameterIndex:11, transaction.getInspectionDate().toString());
    stmt.setBigDecimal(parameterIndex:12, transaction.getAmount());
    stmt.setString(parameterIndex:13, transaction.getOfficialReceiptNumber());

    stmt.execute();
    conn.close();
}
```

Figure 27: Body of the **addBusinessClearanceTransaction()** method

Update Entity Methods

As what is shown in **Figures 28** and **29**, the update methods **userDB.updateUserAccount()** and **txnDB.updateBusinessClearanceTransaction()** are practically the same with the **Add Entity** methods, with the only difference that they are using an *UPDATE* query rather than *INSERT*, and passing the condition as the one that matches either the *id* values *userId* or *transactionId* of the passed identity.

```
// Updates an existing user account in the database.
public static void updateUserAccount(UserAccount userAccount) throws SQLException {
    String query = "UPDATE `" + DBConnection.getDBName()
            + "`.`user_account` SET `username` = ?, `password` = ?, `lastname` = ?,
            `firstname` = ?, `middlename` = ? WHERE `user_account`.`user_id` = ?";

    Connection conn = DBConnection.getConnection();
    PreparedStatement stmt = conn.prepareStatement(query);

    stmt.setString(parameterIndex:1, userAccount.getUsername());
    stmt.setString(parameterIndex:2, userAccount.getPassword());
    stmt.setString(parameterIndex:3, userAccount.getLastname());
    stmt.setString(parameterIndex:4, userAccount.getFirstname());
    stmt.setString(parameterIndex:5, userAccount.getMiddlename());
    stmt.setInt(parameterIndex:6, userAccount.getUserId());

    stmt.execute();
    conn.close();
}
```

Figure 28: Body of the **updateUserAccount**() method

```
// update business clearance transaction
public static void updateBusinessClearanceTransaction(BusinessClearanceTransaction transaction)
        throws SQLException {
    String query = "UPDATE `" + DBConnection.getDBName()
            + "`.`business_clearance_transaction` SET "
            + "`inspection_type` = ?, `owner` = ?, `owner_address` = ?, `business_address` = ?, "
            + "`business_name` = ?, `business_type` = ?, `contact_number` = ?, `property_type` = ?, "
            + "`registration_number` = ?, `inspector` = ?, `inspection_date` = ?, `amount` = ?, "
            + "`official_receipt_number` = ? WHERE `transaction_id` = ?;";

    Connection conn = DBConnection.getConnection();
    PreparedStatement stmt = conn.prepareStatement(query);

    stmt.setString(parameterIndex:1, transaction.getInspectionType().toString());
    stmt.setString(parameterIndex:2, transaction.getOwner());
    stmt.setString(parameterIndex:3, transaction.getOwnerAddress());
    stmt.setString(parameterIndex:4, transaction.getBusinessAddress());
    stmt.setString(parameterIndex:5, transaction.getBusinessName());
    stmt.setString(parameterIndex:6, transaction.getBusinessType());
    stmt.setString(parameterIndex:7, transaction.getContactNumber());
    stmt.setString(parameterIndex:8, transaction.getPropertyType().toString());
    stmt.setString(parameterIndex:9, transaction.getRegistrationNumber());
    stmt.setString(parameterIndex:10, transaction.getInspector());
    stmt.setString(parameterIndex:11, transaction.getInspectionDate().toString());
    stmt.setBigDecimal(parameterIndex:12, transaction.getAmount());
    stmt.setString(parameterIndex:13, transaction.getOfficialReceiptNumber());
    stmt.setInt(parameterIndex:14, transaction.getTransactionId());

    stmt.execute();
    conn.close();
}
```

Figure 28: Body of the **updateBusinessClearanceTransaction()** method

Delete Entity Methods

Deletion only requires updating the *status* attribute of the entity to *0*, the query strings of which are shown in **Figures 29** and **30**. We can then assume that the query is executed similarly to the previously covered methods.

```
// Deletes a user account from the database.
public static void deleteUserAccount(int userId) throws SQLException {
    // update status to 0 (deleted)
    String query = "UPDATE `" + DBConnection.getDBName()
            + "`.`user_account` SET `status` = 0 WHERE `user_account`.`user_id` = " + userId + ";";
```

Figure 29: Query String of the **deleteUserAccount**() method

```
// delete business clearance transaction
public static void deleteBusinessClearanceTransaction(int transactionId) throws SQLException {
    // set status to 0
    String query = "UPDATE `" + DBConnection.getDBName()
            + "`.`business_clearance_transaction` SET `status` = 0 WHERE `transaction_id` = "
            + transactionId + ";";
```

Figure 30: Query String of the **deleteBusinessClearanceTransaction**() method

Table Populating Methods

These are the methods used to populate the passed JavaFX node **TableView**. As shown in **Figures 31** and **32**, both methods begin by setting up a query string selecting all active entities, that is their *status* attribute is set to *1*, then passing the resulting **ResultSet** into a **PopulateTable.populateTable(..**

The **PopulateTable** is a custom made utility class which populates the passed **TableView** with the contents of the **ResultSet**, as shown in **Figure 33.** The passed **TableView** is then what is rendered in the **Business Clearance Transaction List View** and **User Account List View**.

```
// Populates a TableView with user accounts from the database.
public static void populateTable(TableView tableView) throws SQLException {
    String query = "SELECT user_id, username, password, firstname, middlename, lastname FROM `"
            + DBConnection.getDBName() + "`.`user_account` WHERE status = 1;";

    Connection conn = DBConnection.getConnection();
    PreparedStatement stmt = conn.prepareStatement(query);
    ResultSet rs = stmt.executeQuery();

    PopulateTable.populateTable(tableView, rs);
    conn.close();
}
```

Figure 31: Body of the **userDB.populateTable()** method

```
// populate TableView with full list
public static void populateTable(TableView tableView) throws SQLException {
    String query = "SELECT transaction_id, inspection_type, owner, owner_address, business_name, business_address, business_type,
    contact_number, property_type, registration_number, inspector, inspection_date, amount, official_receipt_number FROM `"
            + DBConnection.getDBName()
            + "`.`business_clearance_transaction` WHERE status = 1;";

    Connection conn = DBConnection.getConnection();
    PreparedStatement stmt = conn.prepareStatement(query);
    ResultSet rs = stmt.executeQuery();

    PopulateTable.populateTable(tableView, rs);
    conn.close();
}
```

Figure 32: Body of the **txnDB.populateTable()** method

```
public class PopulateTable {
    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static void populateTable(TableView tableView, ResultSet rs) throws
    SQLException {
        // Clear the table
        tableView.getItems().clear();
        tableView.getColumns().clear();

        ObservableList<ObservableList> data = tableView.getItems();

        for (int i = 0; i < rs.getMetaData().getColumnCount(); i++) {
            final int j = i;
            TableColumn col = new TableColumn(rs.getMetaData().getColumnName(i + 1));
            col.setCellValueFactory(
                    (Callback<TableColumn.CellDataFeatures<ObservableList, String>,
                    ObservableValue<String>>) param -> new SimpleStringProperty(
                            param.getValue().get(j).toString()));
            tableView.getColumns().add(col);
        }

        while (rs.next()) {
            ObservableList<String> row = FXCollections.observableArrayList();
            for (int i = 1; i <= rs.getMetaData().getColumnCount(); i++) {
                row.add(rs.getString(i));
            }
            data.add(row);
        }
    }
}
```

Figure 32: Body of the **PopulateTable.populateTable()** method

Methods for Populating with Search Filter

With the **PopulateTable** utility class, it would also be a matter of passing the right query to implement the search function. As shown in **Figures 8** and **14**, the **List Views** uses **ControlFX's CheckComboBox** to allow selection of specific columns that would be searched through.

These are parametrized in the methods explored in **Figures 33** and **34**. The **populateSearchResults()** method is what is publicly accessed which takes in the **TableView** to be populated, the **String** in the search field (*searchString)*, and an **ArrayList of Strings** *(searchOptions)* which store the specific columns to be searched through.

The helper method **searchQuery** creates the query by iterating through the *searchOptions* for *LIKE* statements, and retrieves the **ResultSet** which then would be used with **PopulateTable.populateTable()** to populate the initial **TableView**.

All these methods work together to produce a working table search functionality for the **List Views**.

```java
// Populates a TableView with search results from the database.
public static void populateSearchResults(TableView tableView, String searchString, ArrayList<String> searchOptions)
        throws SQLException {
    ResultSet rs = searchQuery(searchString, searchOptions);
    PopulateTable.populateTable(tableView, rs);
}

// Executes a search query on the user accounts in the database.
private static ResultSet searchQuery(String searchString, ArrayList<String> searchOptions) throws SQLException {
    String query = "SELECT user_id, username, password, firstname, middlename, lastname FROM `"
            + DBConnection.getDBName() + "`.`user_account` WHERE status = 1 AND (";
    for (int i = 0; i < searchOptions.size(); i++) {
        query += searchOptions.get(i) + " LIKE ? ";
        if (i < searchOptions.size() - 1) {
            query += "OR ";
        }
    }
    query += ");";

    Connection conn = DBConnection.getConnection();
    PreparedStatement stmt = conn.prepareStatement(query);
    for (int i = 0; i < searchOptions.size(); i++) {
        stmt.setString(i + 1, "%" + searchString + "%");
    }
    ResultSet rs = stmt.executeQuery();

    return rs;
}
```

Figure 33: **userDB.populateSearchResults()** and **userDB.searchQuery()**

```java
// populate TableView with search results
public static void populateSearchResults(TableView tableView, String searchString, ArrayList<String> searchOptions)
        throws SQLException {
    ResultSet rs = searchQuery(searchString, searchOptions);
    PopulateTable.populateTable(tableView, rs);
}

// customized search query
private static ResultSet searchQuery(String searchString, ArrayList<String> searchOptions) throws SQLException {
    String query = "SELECT transaction_id, inspection_type, owner, owner_address, business_name, business_address, business_type, contact_number,
    property_type, registration_number, inspector, inspection_date, amount, official_receipt_number FROM `"
            + DBConnection.getDBName()
            + "`.`business_clearance_transaction` WHERE status = 1 AND (";

    for (int i = 0; i < searchOptions.size(); i++) {
        query += searchOptions.get(i) + " LIKE ? ";
        if (i < searchOptions.size() - 1) {
            query += "OR ";
        }
    }

    query += ");";

    Connection conn = DBConnection.getConnection();
    PreparedStatement stmt = conn.prepareStatement(query);
    for (int i = 0; i < searchOptions.size(); i++) {
        stmt.setString(i + 1, "%" + searchString + "%");
    }
    ResultSet rs = stmt.executeQuery();

    return rs;
}
```

Figure 34: **txnDB.populateSearchResults()** and **txnDB.searchQuery()**

5. Data Flow

The full data flow between the explored components of the Barangay Clearance Database Management System is shown in **Figure 20**.

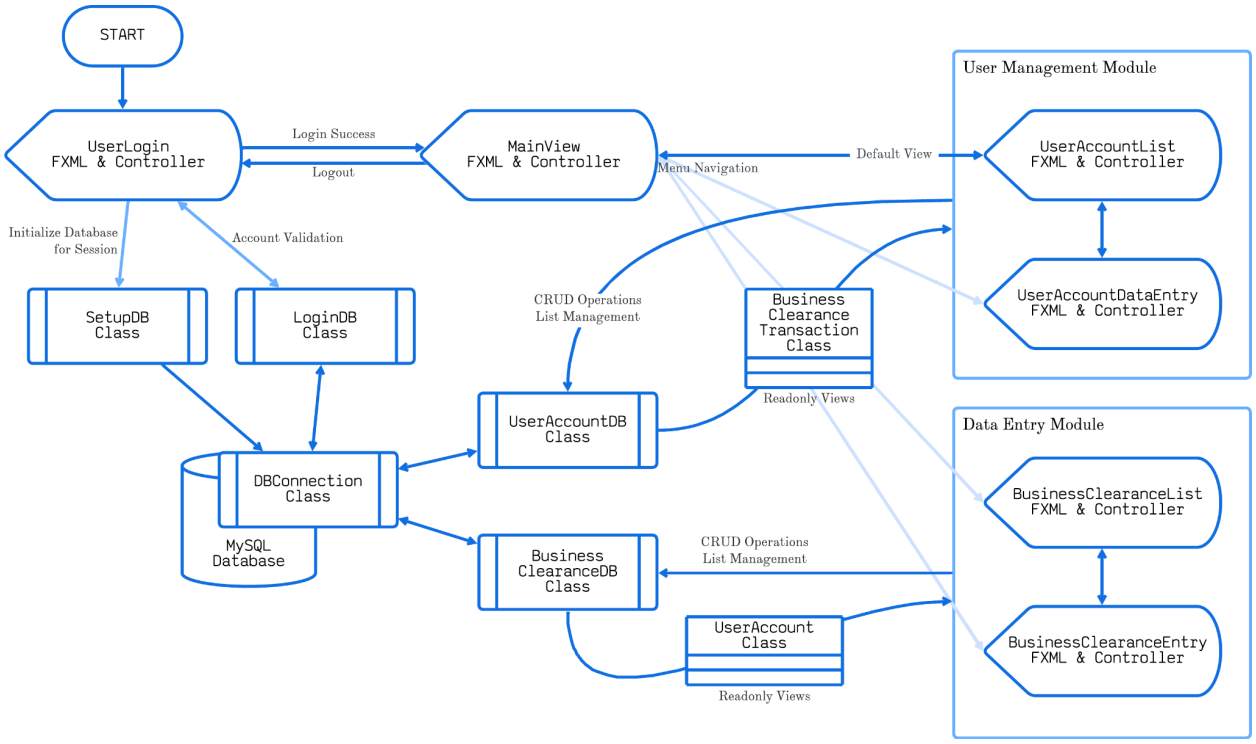


Figure 35: **Data Flow** between entities making up the Database Management

6. Utility Classes

With a project this big, it was important to have some shared utility classes to reduce redundancy. The utility classes used in this project are the **PopulateTable**, **ThrowAlert** and **NodeValidation** classes, as shown in **Figure 36**.

The functionality of the **PopulateTable** class was already covered in the database section of the documentation.

The **ThrowAlert** class allows for automated creation of Alert scenes in JavaFX, with the contents of the alerts parametrized. This was very useful for handling caught Exceptions, especially by passing the exception error message into the Alert content argument.

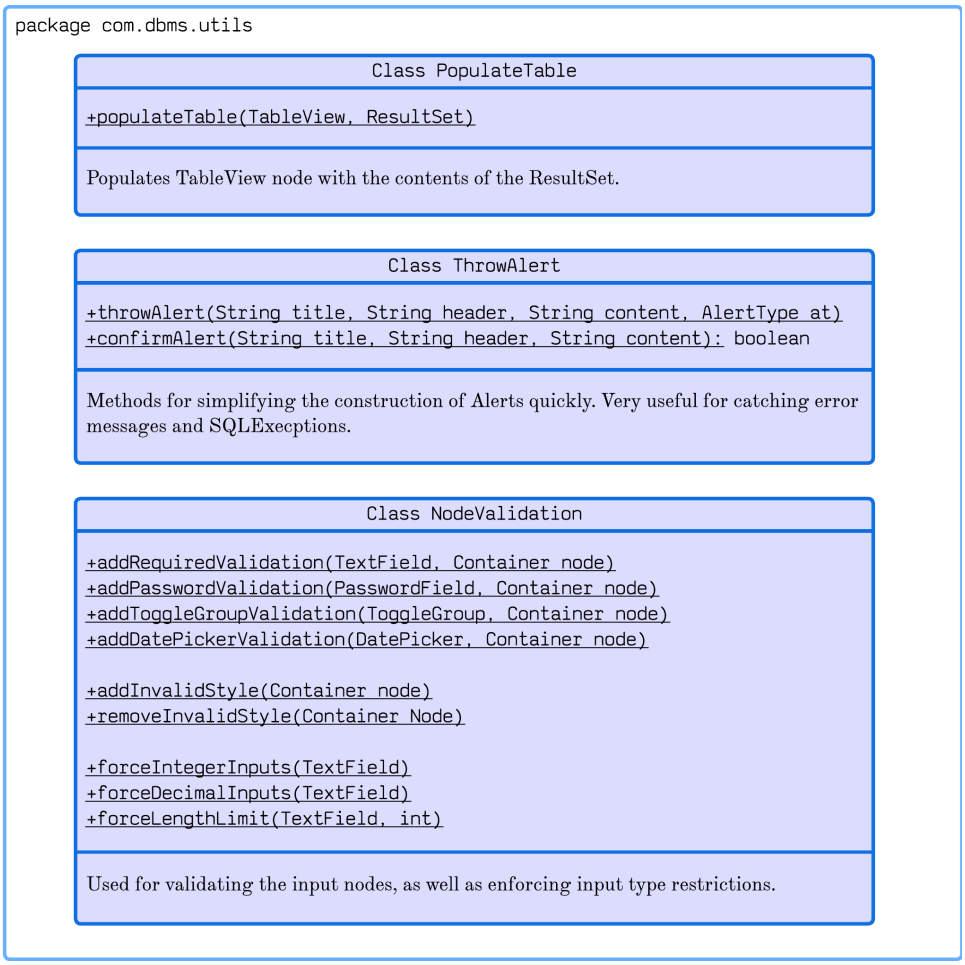


Figure 36: **Utility Classes**

The NodeValidation initially was used for adding the input validation for the data entry views.

The first four methods **addRequiredValidation()**, **addPasswordValidation()**, **addToggleGroupValidation()** and **addDatePickerValidation()** are methods that add a *required* property to the Nodes passed in the first argument. The second argument takes in the "container" of the required Node.

The *required* property is implemented by adding an event listener to the *required* Node which checks if it was left empty, in which it would then add the **.invalid** class to the container, which through **CSS** would then change the state of all the nodes within the container.

An example of this would be shown in **Figures 37** and **38**, in which the **TextField** is passed into the **addRequiredValidation()** method, with the **HBox** around it as the container node. Through the **CSS** shown in **Figure 39**, both the **TextField** and **Label** turns red when the **TextField** is left empty.

Figure 37: Normal **HBox** containing a TextField and a Label

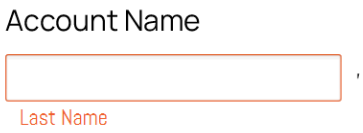


Figure 38: The same **HBox** with **.invalid** class

```
Label {
    -fx-text-fill: #2c2c2c;
}

TextField,
PasswordField {
    -fx-border-color: #ccc;
    -fx-border-width: 1px;
    -fx-border-radius: 2px;
}

.invalid Label {
    -fx-text-fill: #f06431;
    -fx-focus-color: #cc0105;
    -fx-faint-focus-color: #cc010522;
}

.invalid TextField,
.invalid PasswordField {
    -fx-focus-color: #cc0105;
    -fx-faint-focus-color: #cc010522;
    -fx-border-color: #f06431;
    -fx-prompt-text-fill: #cc010580;
}
```

Figure 39: CSS Snippet showing how the **.invalid** class works

The **addInvalidStyle()** and **removeInvalidStyle()** methods should be self explanatory, which easily adds or removes the **.invalid** class from the passed container node. It also helps against repeated addition of the **.invalid** class when the container is already invalid.

The **forceIntegerInputs(), forceDecimalInputs()** and **forceLengthLimit()** are more about restricting the inputs of the **TextField**. These are added to enforce the same limits defined in the MySQL tables, like limited length of the **VARCHAR**, **INTEGER** and **DECIMAL** fields. You can also pass a **PasswordField** to these by casting them first to **TextField**.