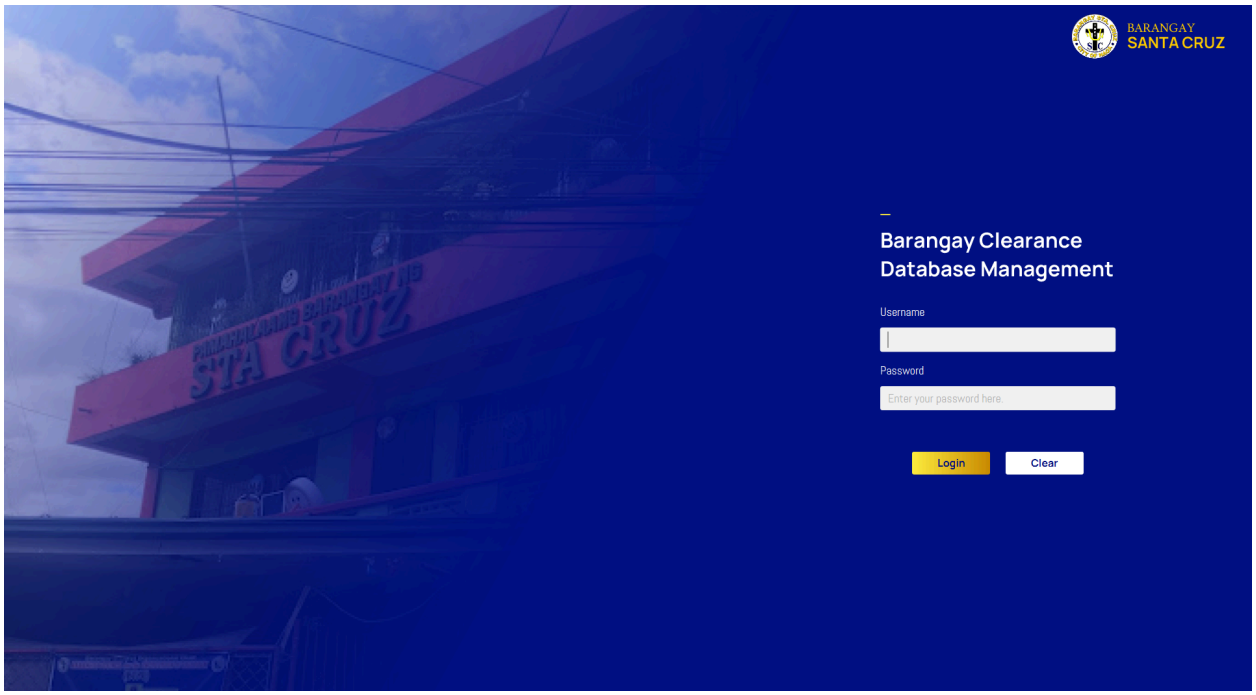


Business Clearance Database Management System for Barangay Sta. Cruz

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.
- **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
);
```

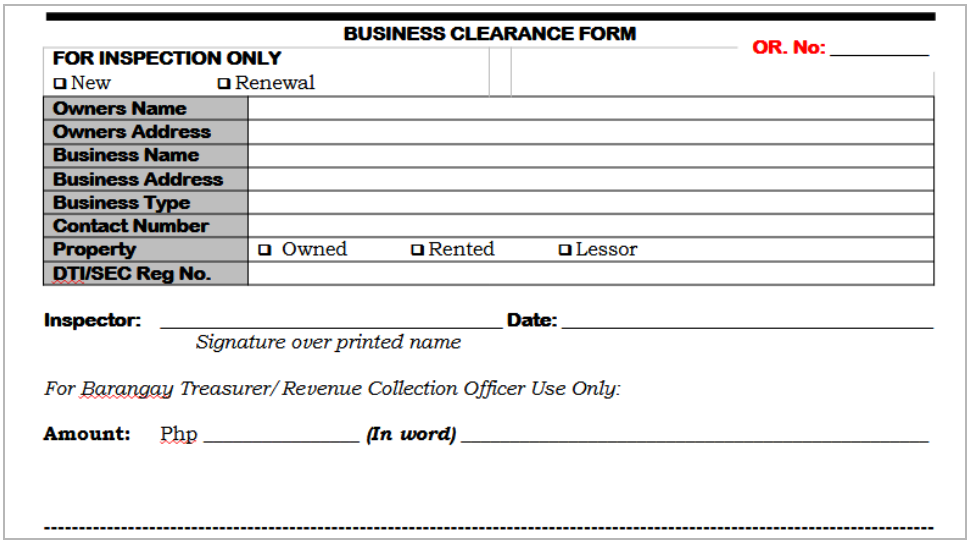
Figure 2: MySQL Query to create the **user_account** table

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.

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

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* 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



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

maximum integer value. Lastly, the amount is stored in **decimal** with **2** decimal digits to represent the Peso currency.

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.

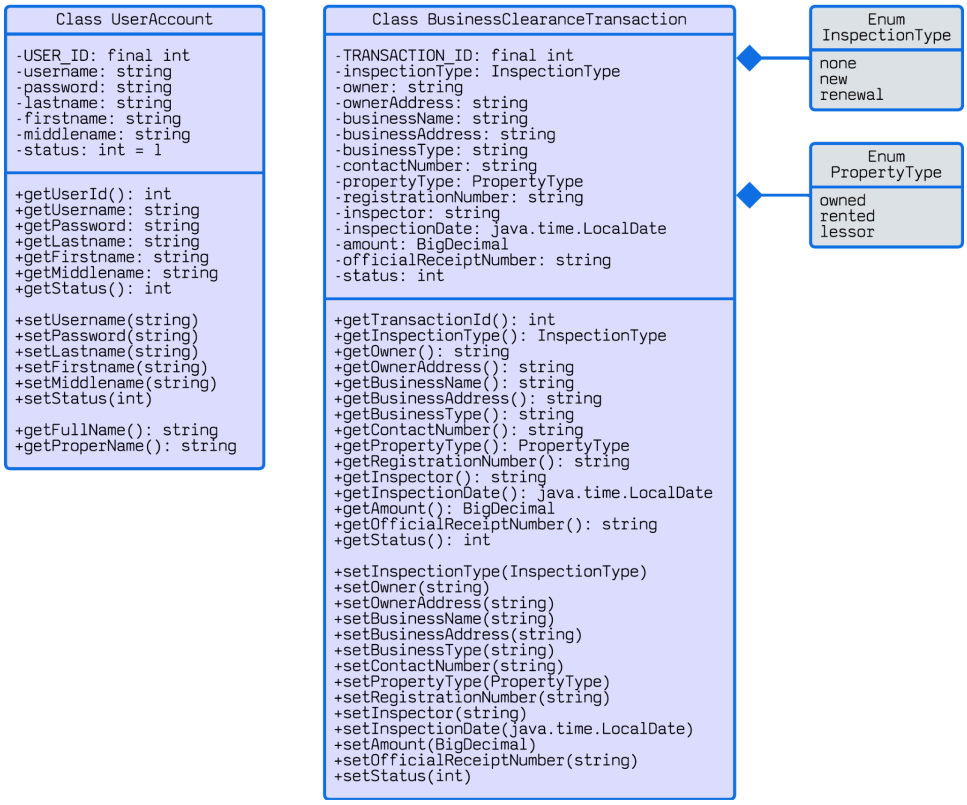


Figure 6: UML Diagrams of the **user_account** and **business_clearance_transaction** tables

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.

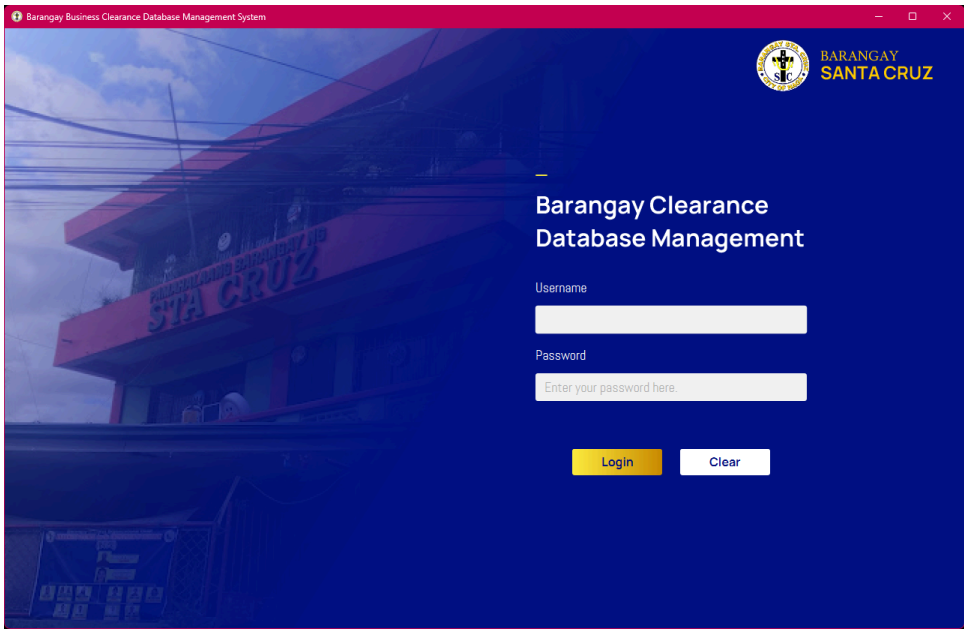


Figure 7: **User Login View**

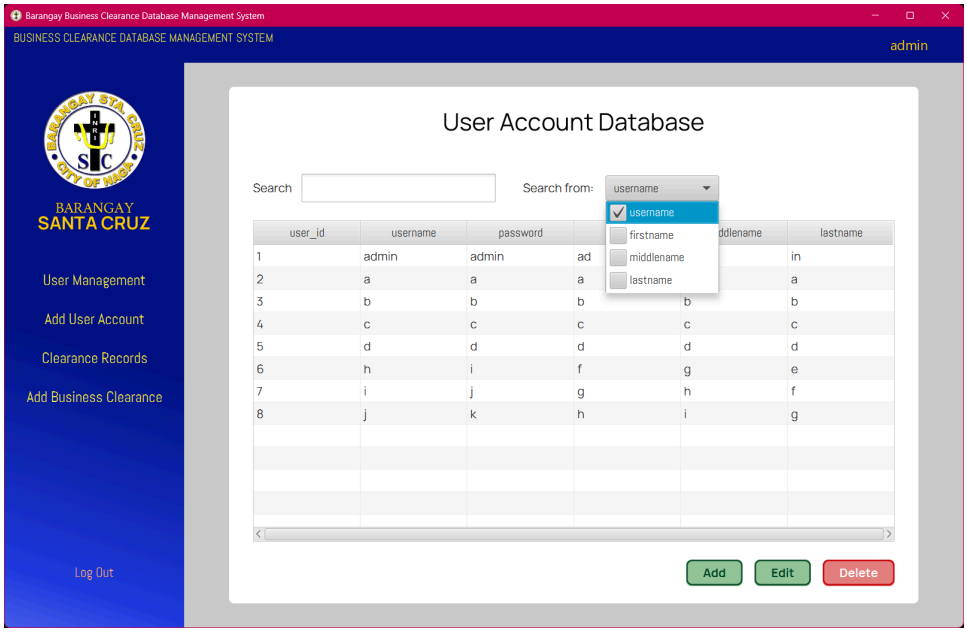


Figure 8: **Main View** and **User Management Views** together

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.

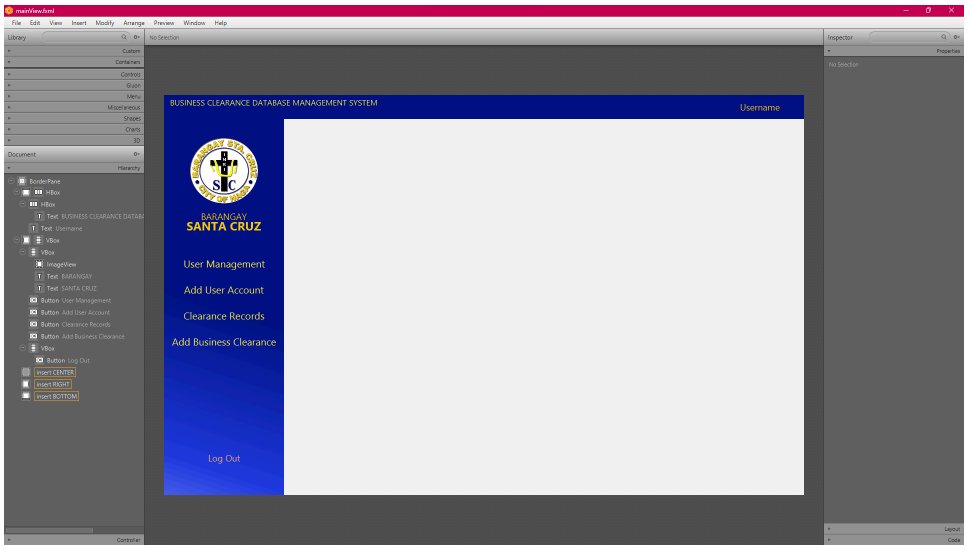


Figure 9: **Main View** in SceneBuilder

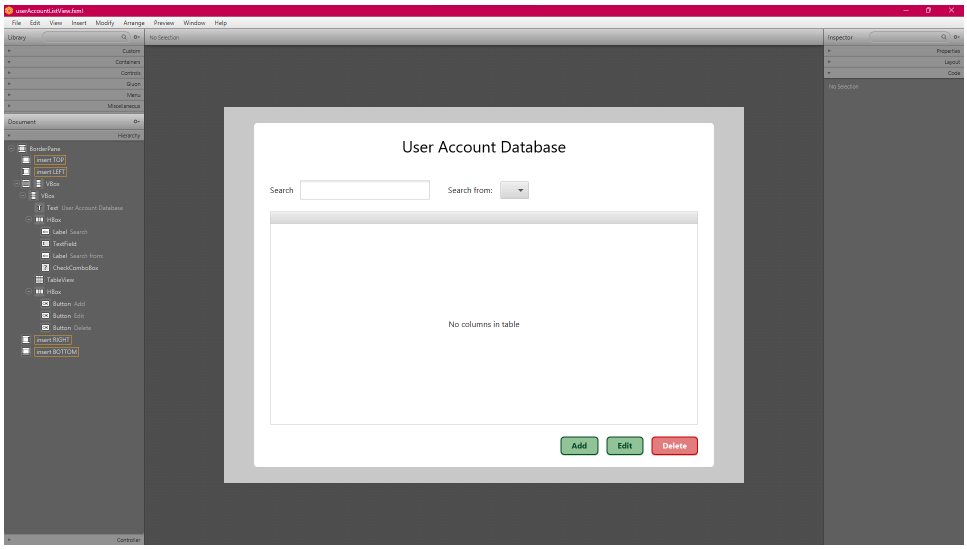


Figure 10: **User Management View** in SceneBuilder

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.**

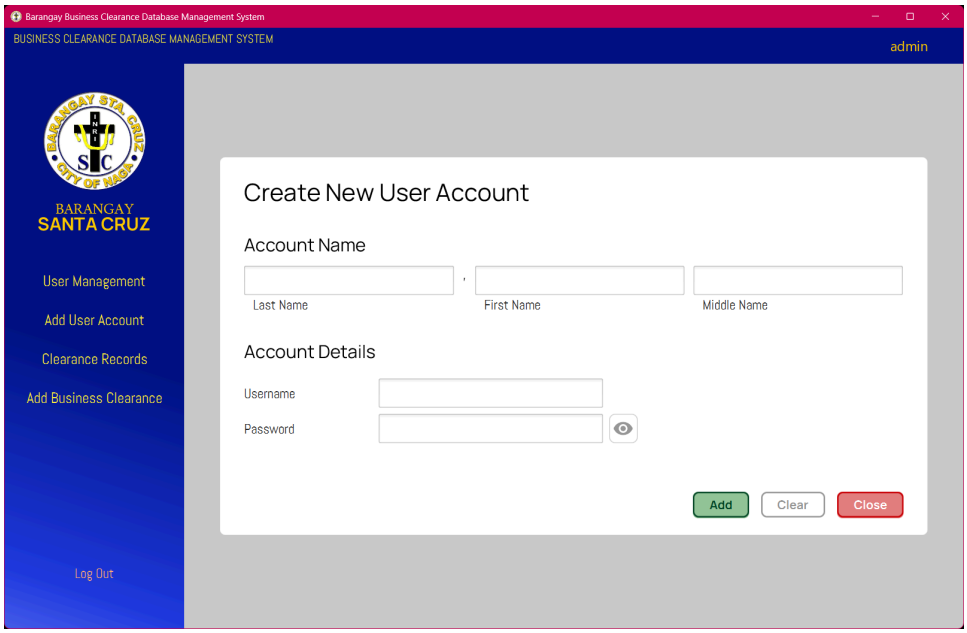


Figure 11: Adding a new account using **User Account Entry**

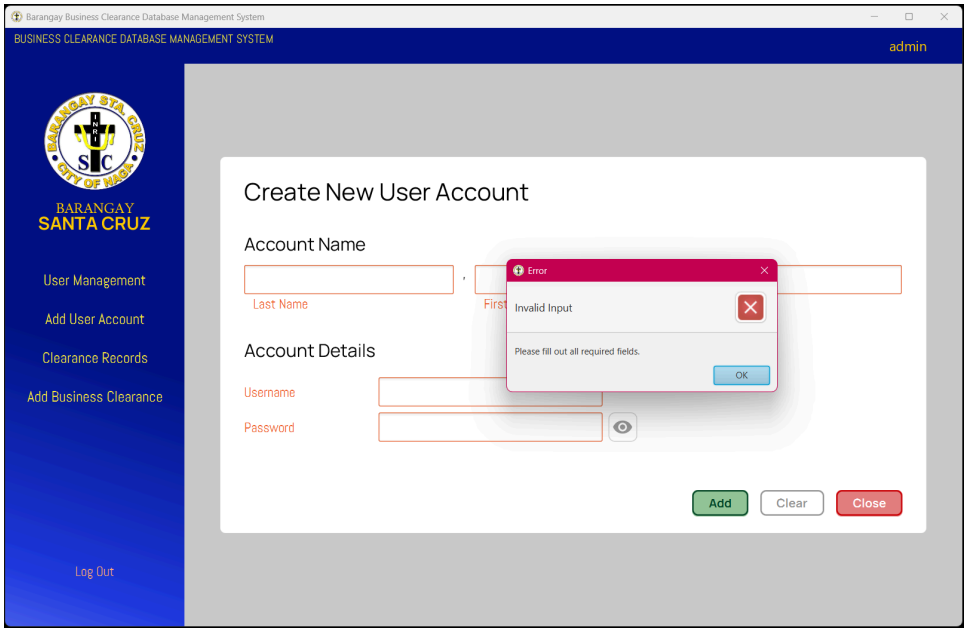


Figure 12: Required fields have to be filled

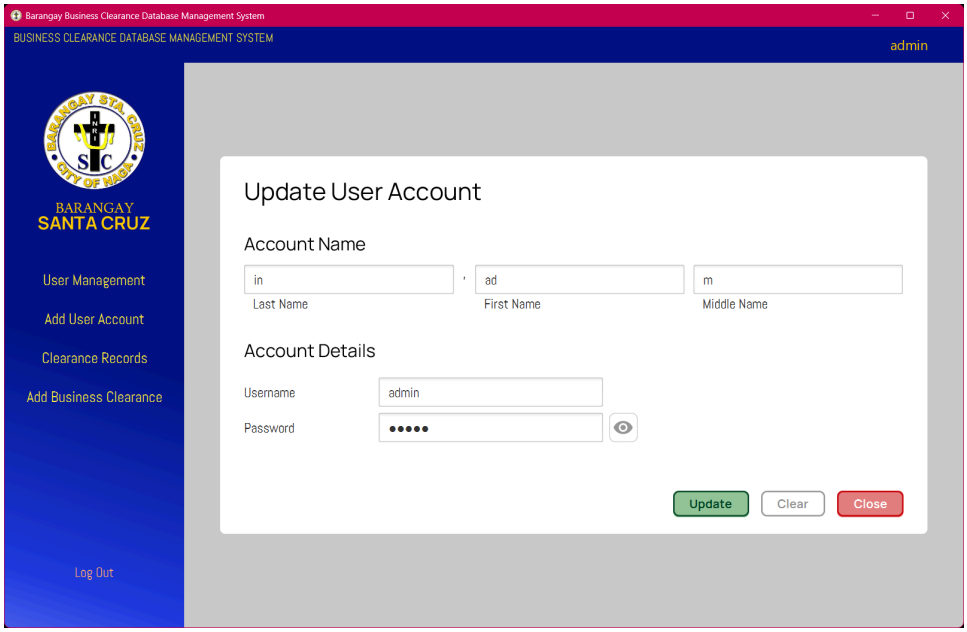


Figure 13: Loading an existing account in **User Account Entry**

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.

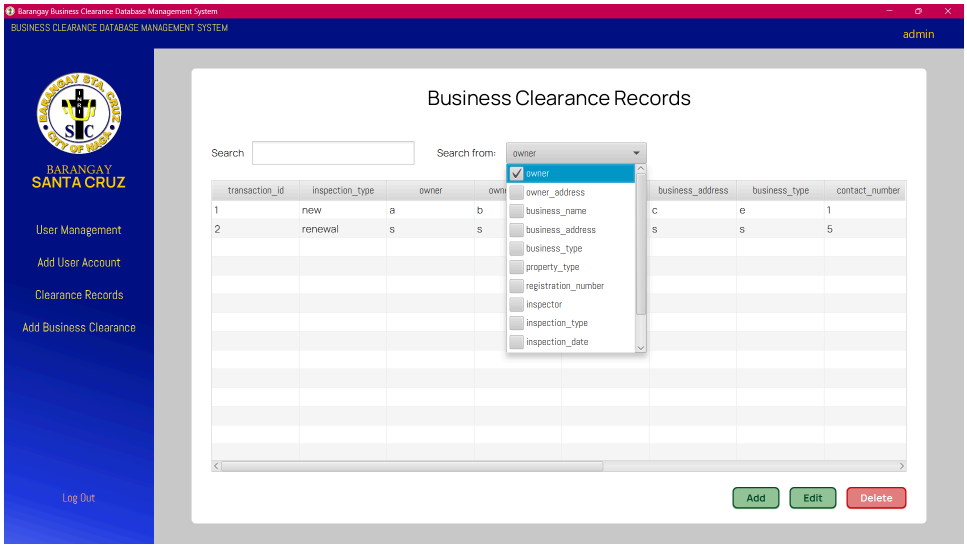


Figure 14: Loading an existing account in **User Account Entry**

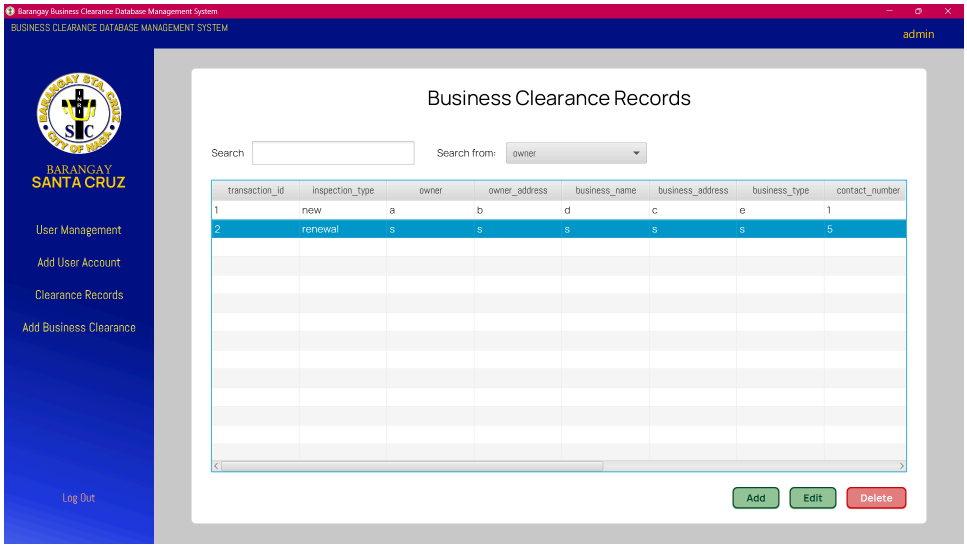


Figure 15: Selecting individual transactions from the table

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**.

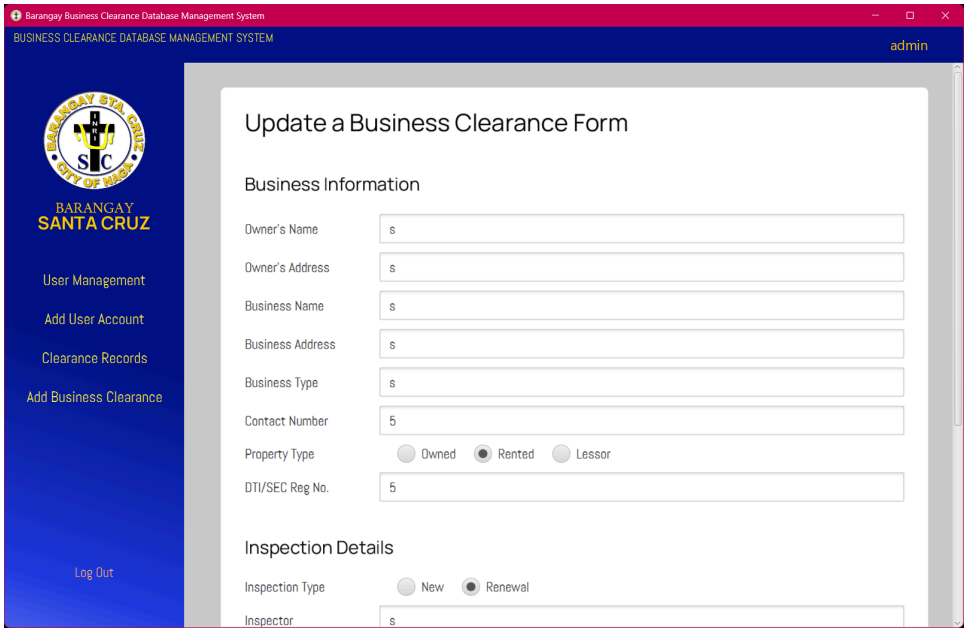


Figure 16: Top of the **User Account Entry View**, editing the selected transaction from Figure 15

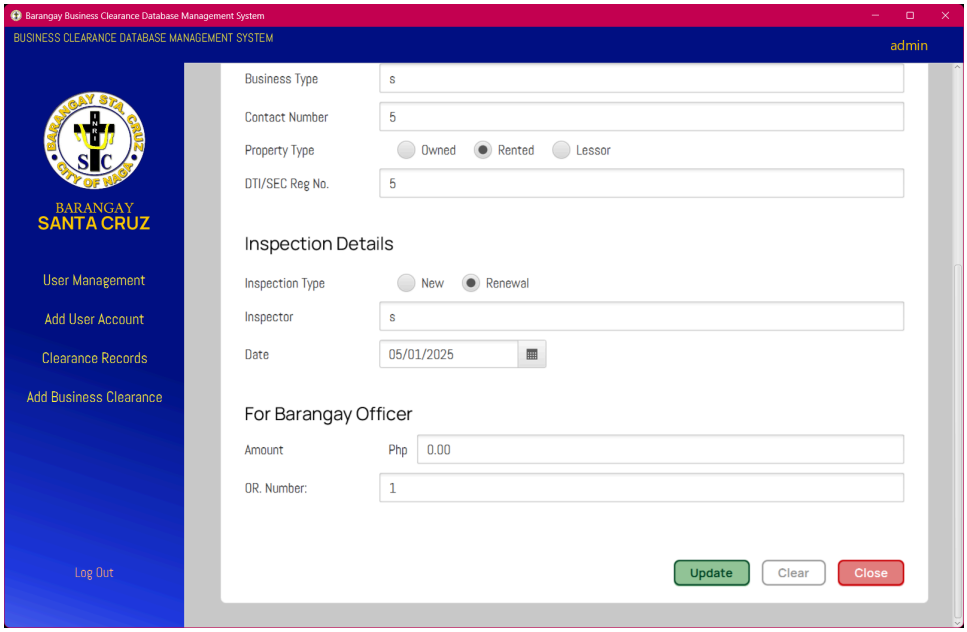


Figure 17: Bottom of the **User Account Entry View**

4. Database

Database Configuration

oop_dbms > src > main > java > com > dbms > database > DBConnection.java > ...

```java
package com.dbms.database;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

// Utility class to connect to the database
public class DBConnection {
    static private final String DB_URL = "jdbc:mysql://localhost:3309/";
    static private final String USER = "root";
    static private final String PASS = "NotCat24";
    static private final String DB_NAME = "oop_dbms";

    // Returns a connection to the database
    protected static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(DB_URL, USER, PASS);
    }

    // Returns the database name
    protected static String getDBName() {
        return DB_NAME;
    }
}
```

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.

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

The **SetupDB** is run to initialize the database after a successful login, creating the database and tables if they don't exist.

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

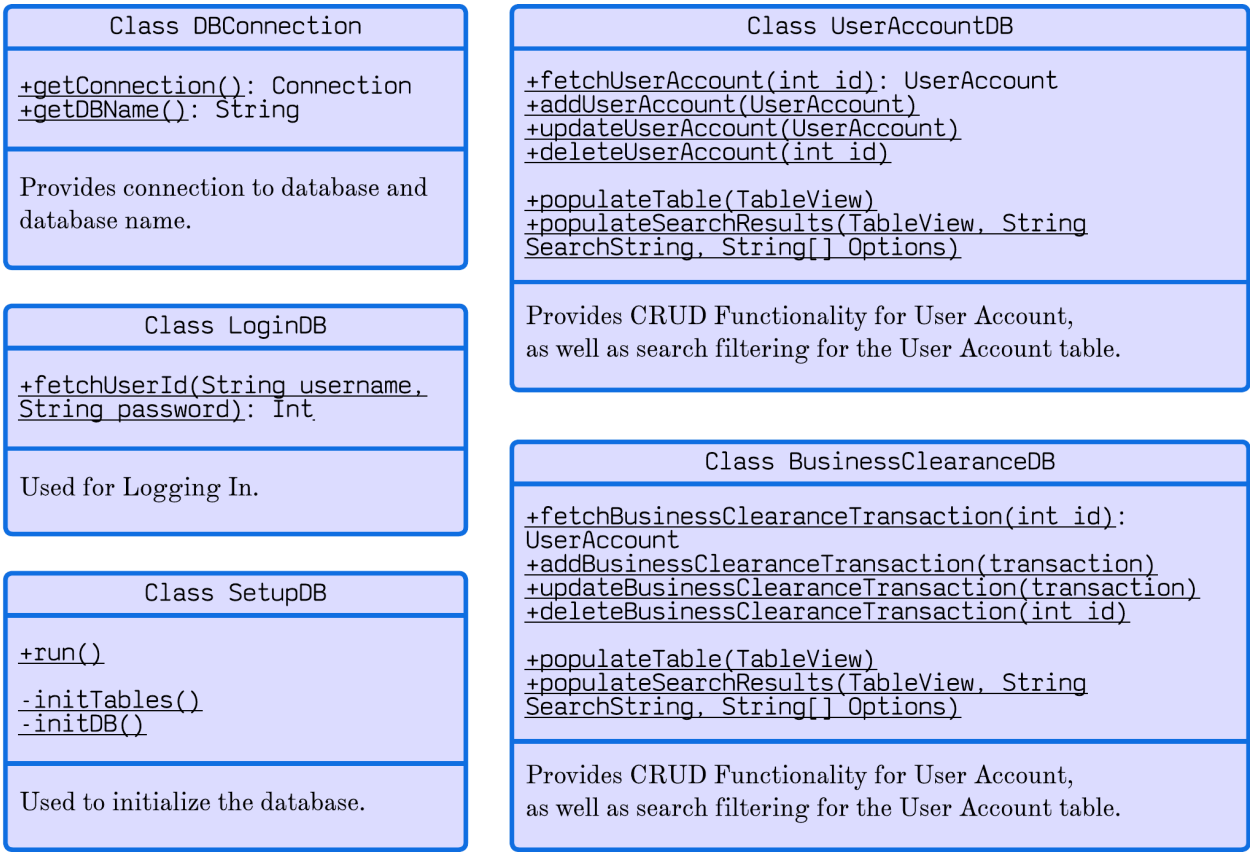


Figure 19: **Database Interface Classes**

5. Data Flow

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

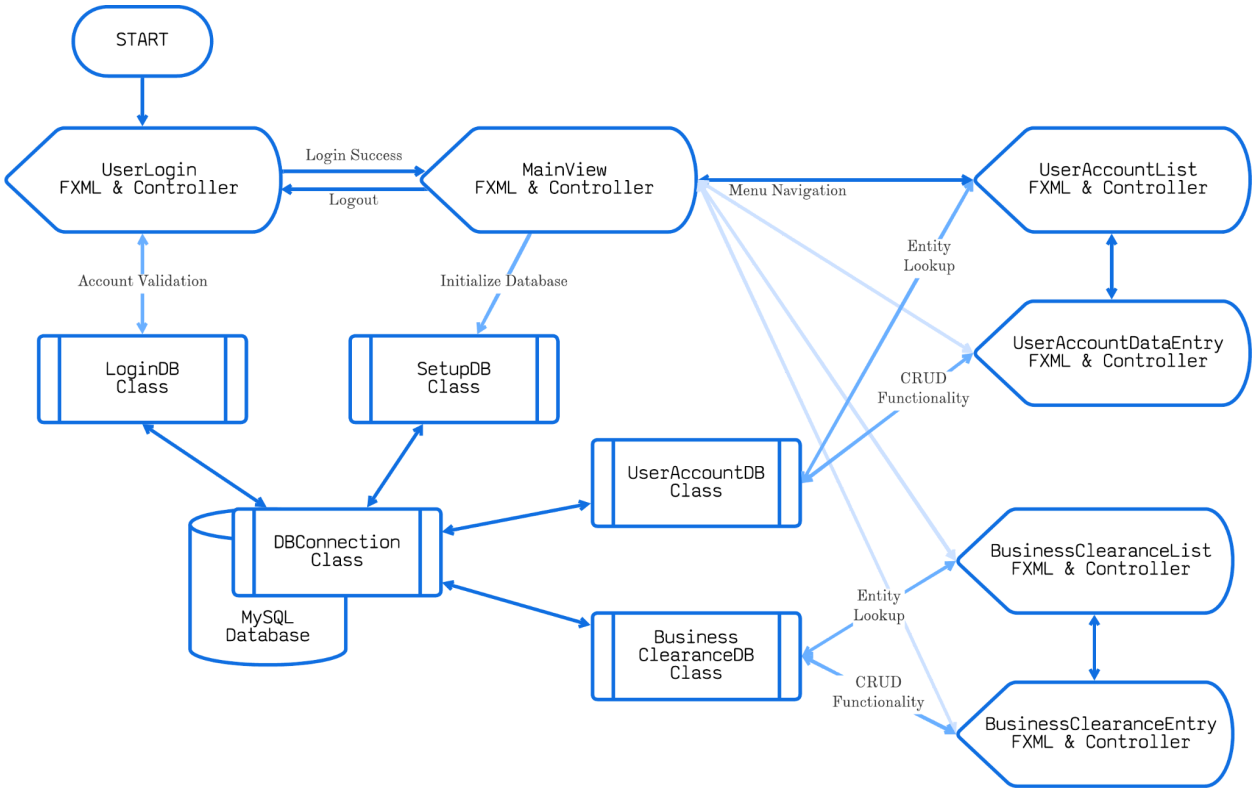


Figure 20: **Data Flow** between entities making up the Database Management