Book Shop Inventory Management System

Introduction

The *EasyTech Bookstore Inventory Management System* is a Java-based desktop application designed to streamline and automate bookstore operations. Developed using JavaFX and socket programming, this system offers a modern and interactive graphical user interface (GUI) to manage inventory, handle sales, process payments, and register customer complaints. It connects multiple clients to a server using sockets and integrates with a MySQL database for persistent data storage and retrieval. The system is tailored to empower bookstore owners by simplifying day-to-day operations such as adding/editing inventory items, recording sales, and accepting payments via mobile banking or cards.

Motivation

Traditional bookstore operations often rely on manual processes which are time-consuming, error-prone, and lack real-time insights into sales or inventory levels. These limitations inspired the development of a software-based solution that can:

- Automate inventory and sales tracking.
- Handle real-time multi-client communication.
- Integrate payment systems like **BKASH**, **NAGAD**, and **Card payments**.
- Maintain proper transaction logs and generate digital receipts.
- Provide a professional dashboard for easy access and operation.

This project was also motivated by the opportunity to apply advanced object-oriented programming concepts such as **multithreading**, **socket communication**, **JavaFX UI development**, and **MySQL integration**.

Objectives

The main objectives of the *Bookstore Inventory Management System* are:

1. **Inventory Management**
    - Add, edit, delete, and search items.
    - Maintain stock level integrity by validating sales quantities.
2. **Sales Recording**
    - Log each sale with time, item name, quantity, and revenue.
    - Display total items sold and total revenue in real time.
3. **Payment Integration**
    - Accept payments via BKASH, NAGAD, and CARD.
    - Validate payment credentials.
    - Store transaction history and generate PDF receipts.
4. **User Authentication**
    - Secure login/logout mechanism.
    - Store user credentials in the database with validation.
5. **Complain Box**
    - Allow users to submit complaints.
    - Enable client-server chat support through socket programming.
6. **Socket Communication & Threading**
    - Maintain live server-client communication for complaints and system operations using multi-threaded socket handling.
    - Ensure responsive UI by managing blocking operations on background threads.
7. **Data Export & Reporting**

- Export sales data to CSV for analysis or backup.

Features

1 Socket Programming Details

Socket programming forms the backbone of the client-server communication model. It involves establishing virtual communication endpoints (sockets) through which data streams can flow.

1.1. Server-Side Socket Implementation (*ChatServer.java*)

The *ChatServer.java* component acts as the central hub, responsible for listening for incoming connections and managing data flow between clients.

i) ServerSocket: The `ServerSocket` class is exclusively used on the server to create a listening point for incoming client connections. It binds to a specific port, making the server discoverable and accessible on the network.

Implementation:
private static final int PORT = 12345; // Defines the port for server listening
ServerSocket serverSocket = new ServerSocket(PORT); // Binds the server to the port

Connection Acceptance (`accept()` method):
Socket clientSocket = serverSocket.accept(); // Blocking call

The `accept()` method is a crucial blocking operation. The server's main thread pauses execution at this point, waiting indefinitely until a client attempts to establish a connection. Upon a successful connection, `accept()` returns a *new* `Socket` object. This `clientSocket` represents the dedicated, unique communication channel established between the server and that specific client. The original `serverSocket` remains active, continuing to listen for subsequent new client connections.

ii) `Socket` (within `ClientHandler`):

Role: Each `ClientHandler` instance on the server manages the communication with one distinct client. The `Socket` object passed to the `ClientHandler` constructor is the direct link to that client.

Implementation:

```
class ClientHandler implements Runnable {
    private Socket clientSocket;
    // ...
    public ClientHandler(Socket socket) {
        this.clientSocket = socket; // Stores the client's dedicated socket
        // ...
    }
}
```

Input/Output Streams (`BufferedReader`, `PrintWriter`):

- **Role**: These classes facilitate the actual exchange of text data over the `Socket` connections.

Implementation (within `ClientHandler` constructor):

```
this.in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
this.out = new PrintWriter(clientSocket.getOutputStream(), true); // 'true' for auto-flush
```

clientSocket.getInputStream(): Provides the raw byte stream of data coming *from* the client.

InputStreamReader: Converts the byte stream from `getInputStream()` into a character stream, essential for reading text data.

BufferedReader: Wraps the ***InputStreamReader*** to provide efficient buffered reading, particularly for line-by-line input using `readLine()`. The `readLine()` method is a **blocking call**, waiting until a full line of text is received.

clientSocket.getOutputStream(): Provides the raw byte stream for data going *to* the client.

PrintWriter: Wraps the OutputStream to offer convenient methods for writing text. The `true` argument in its constructor enables **auto-flushing**, meaning that every call to `println()` automatically flushes the buffer, ensuring immediate transmission of the message over the network.

1.2. Client-Side Socket Implementation (*ChatClient.java*)

The `ChatClient.java` component is responsible for initiating and maintaining a connection with the server from the client application.

`Socket`: On the client, the `Socket` object is used to initiate a connection to the server.

Implementation (within `ChatClient.connect()` method):
socket = new Socket(serverAddress, serverPort); // Blocking call

This line attempts to establish a connection to the server at the specified IP address (`serverAddress`) and port (***serverPort***). This is a **blocking call** that will pause execution until the connection is successfully established or a connection error occurs.

Input/Output Streams (*BufferedReader, PrintWriter*): Similar to the server, these streams are used on the client to send messages *to* the server and receive messages *from* the server.

Implementation (within `ChatClient.connect()` method):

in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
out = new PrintWriter(socket.getOutputStream(), true); // 'true' for auto-flush

out.println(username);: Sends the client's initial username to the server upon connection.

out.println(message);: Sends user-typed chat messages to the server.

message = in.readLine();: Reads incoming messages from the server. This is a **blocking call**, waiting for a message to arrive.

2 Threading Details

Multi-threading is paramount for the responsiveness and scalability of the chat application. It ensures that blocking network operations do not freeze the application's user interface and that the server can handle multiple clients simultaneously.

2.1 Server-Side Threading (*ChatServer.java*)

Thread Submission:

pool.execute(clientHandler); // Submits ClientHandler to the pool

When a new client connects, a ClientHandler instance (which implements Runnable) is submitted to this ExecutorService. The pool then assigns an available thread to execute the run() method of that ClientHandler.

ClientHandler implements Runnable: Each ClientHandler instance is a dedicated task designed to run on its own thread, managing the entire communication lifecycle with a single client.

run() Method Execution*:* The run() method is the entry point for the thread assigned to this ClientHandler.

```
while ((message = in.readLine()) != null) {
    // ... process message ...
    ChatServer.broadcastMessage(fullMessage);
}
```

This while loop continuously reads messages from its associated client using in.readLine(). Since readLine() is a blocking operation, executing it within a separate thread ensures that this blocking does not impede the server's ability to accept new connections or process messages from other clients.

Shared Resource Management (clients list):

private static List<ClientHandler> clients = Collections.synchronizedList(new ArrayList<>());

The clients list, which stores all active ClientHandler instances, is wrapped using *Collections.synchronizedList()*. This is crucial for thread safety, as multiple ClientHandler threads will concurrently access (add, remove, iterate) this shared list. The synchronization prevents race conditions and ConcurrentModificationException.

Thread-Safe Logging (logMessage() method):

```
private static void logMessage(String message) {
    synchronized (ChatServer.class) { // Synchronizes access to the log file
        // ... file writing logic ...
    }
}
```

The logMessage() method, responsible for writing chat messages to chat_log.txt, is synchronized on the ChatServer.class object. This ensures that only one thread can write to the

log file at any given moment, preventing data corruption that could arise from concurrent write attempts by multiple ClientHandler threads.

2.2 Client-Side Threading (*ChatClient.java and ChatController.java*)

To prevent the JavaFX User Interface (UI) from freezing during blocking network operations, thus maintaining responsiveness.

Mechanism: Client-side threading involves offloading network tasks to background threads and using a specific JavaFX utility for safe UI updates.

Explicit Thread Creation for Connection (ChatController.java):

Implementation:

```
new Thread(() -> {
    chatClient = new ChatClient(messages);
    boolean success = chatClient.connect(SERVER_ADDRESS, SERVER_PORT, username);
    Platform.runLater(() -> { /* UI updates */ });
}).start();
```

The *chatClient.connect()* method involves establishing a Socket connection, which is a blocking operation. By executing this within a new, separate thread, the JavaFX Application Thread (which manages the UI) remains free to process user input and render the interface, preventing the application from freezing during the connection attempt.

Explicit Thread Creation for Message Listening (ChatClient.java):

Implementation:

```
new Thread(this::listenForMessages).start(); // Launched after successful connection
```

The listenForMessages() method contains a while loop that continuously calls in.readLine(). As readLine() is a blocking call (it waits indefinitely for data), this operation *must* be performed on a

separate background thread. This ensures that the client's UI remains interactive and responsive, even when no messages are being received from the server.

Platform.runLater() (ChatClient.java and ChatController.java):

This is a critical JavaFX utility for thread-safe UI updates. JavaFX mandates that all modifications to UI components must occur exclusively on the JavaFX Application Thread. Direct manipulation from background threads will lead to IllegalStateException or other unpredictable behavior.

Implementation:

In ChatClient.listenForMessages():
Platform.runLater(() -> messageList.add(receivedMessage));

When a message is received by the background listenForMessages thread, this line queues a task to be executed on the JavaFX Application Thread. This task safely adds the receivedMessage to the ObservableList (messageList), which automatically updates the ListView displayed in the UI.

In ChatController.onConnectDisconnect():
Platform.runLater(() -> { / UI updates */ });*

After the background connection attempt completes, this block is used to safely update UI elements such as the connectButton's text, and enable/disable the messageInput and sendButton based on the connection status.

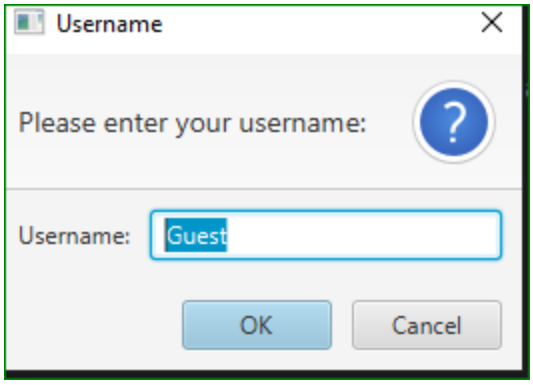


Fig : User Name box for Enter in the Server

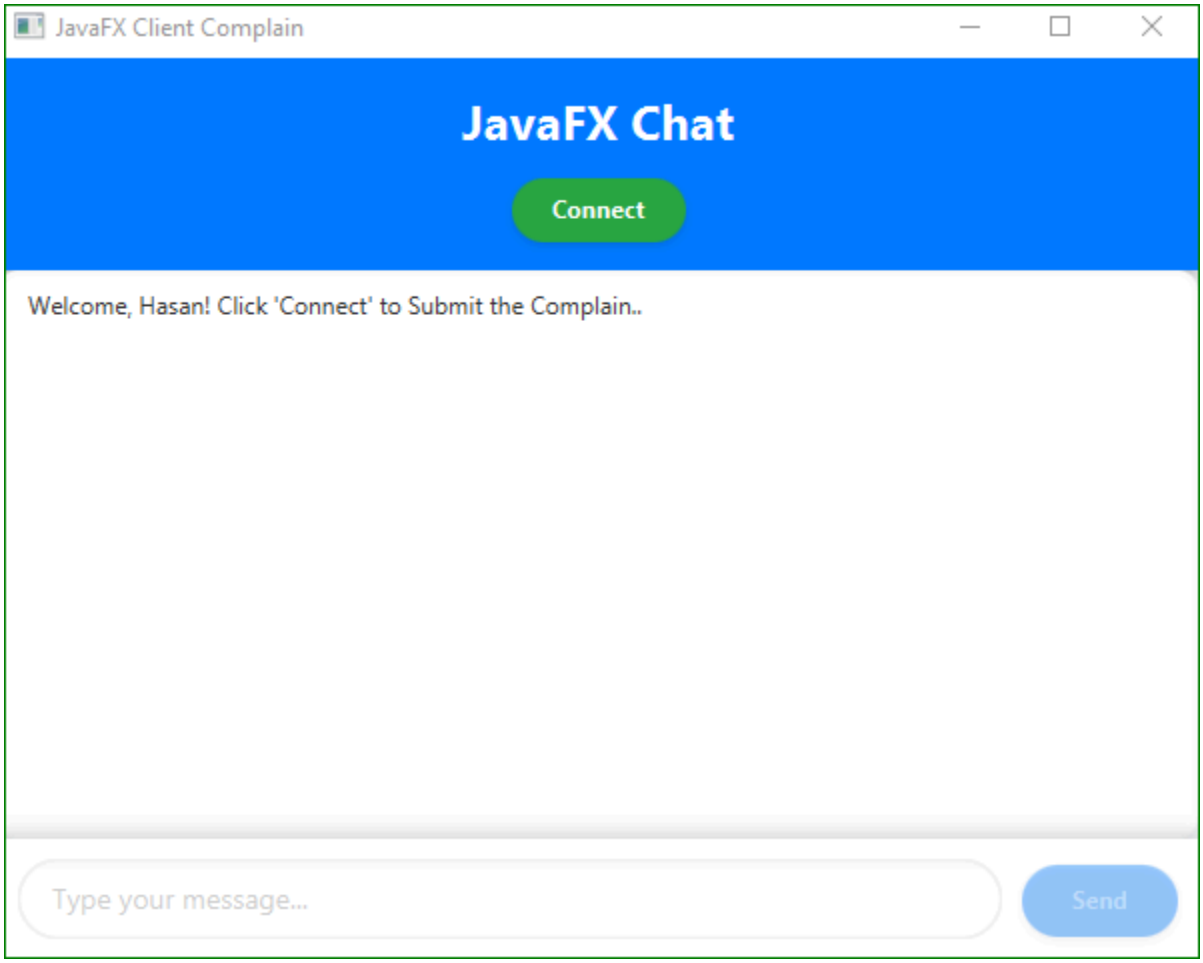


Fig : Server Connection page for User

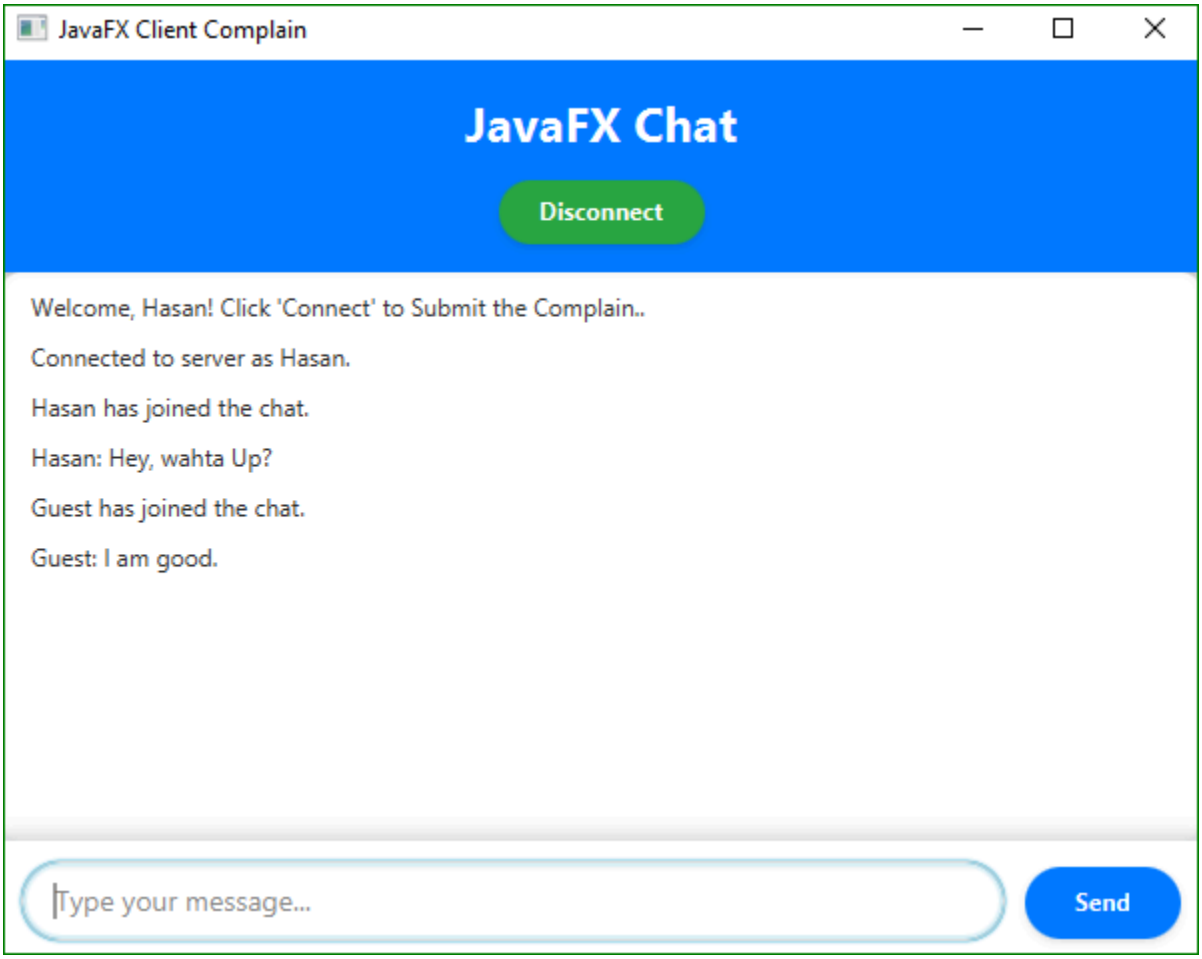


Fig : Complain box for multiple clients

Other features

User Authentication & Management:

- **User Registration:** Allows new users to sign up with a unique username and password. Includes basic input validation (e.g., username length, password matching).
- **User Login:** Secure login functionality to authenticate existing users.

- **Logout:** Provides a clear option to log out of the system, returning to the login screen.
- **Database Integration:** User credentials are stored and validated against a MySQL database (`users` table).

Dashboard & Inventory Management:

- **Interactive Dashboard GUI:** A modern, styled Swing interface with a side navigation bar and a top welcome bar displaying the logged-in username.
- **Inventory Display:** Displays all book items in a sortable and searchable table, showing ID, Item Name, Quantity, and Price.
- **Item Search:** Allows users to search for inventory items by name.
- **Add New Item:** A dedicated module to add new books to the inventory with fields for name, quantity, and price.
- **Edit Item Details:** Users can directly edit existing item details (name, quantity, price) from the inventory table.
- **Delete Item:** Functionality to remove items from the inventory directly from the table.
- **Sell Item:**
    - Allows users to "sell" items by specifying a quantity.
    - Includes stock validation to prevent selling more than available.
    - Automatically updates the inventory quantity after a sale.
    - Logs each sale transaction in a separate sales record.
- **Live Summary Statistics:** Dynamically displays "Total Quantity Sold" and "Total Revenue" on the dashboard.
- **CSV Export:** Enables exporting sales data to a CSV file for reporting or external analysis.
- **Database Integration:** Inventory data is managed in the `items` table in MySQL.

Sales Tracking:

- **Sales Records Table:** Displays a comprehensive list of all sales, including Sale ID, Item Name, Quantity Sold, and Sale Time.
- **Database Integration:** Sales transactions are recorded in the `sales` table in MySQL.

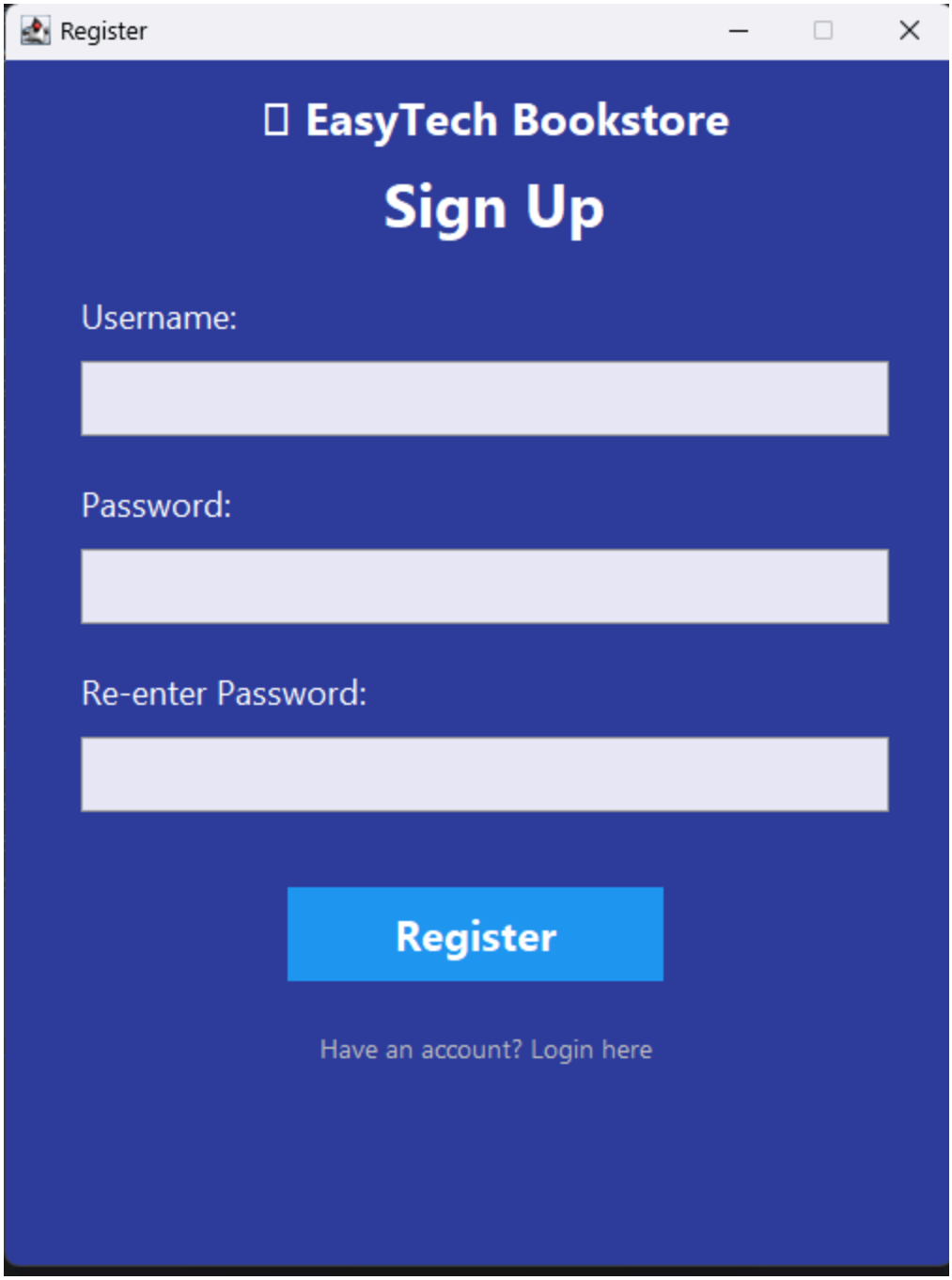
Payment System:

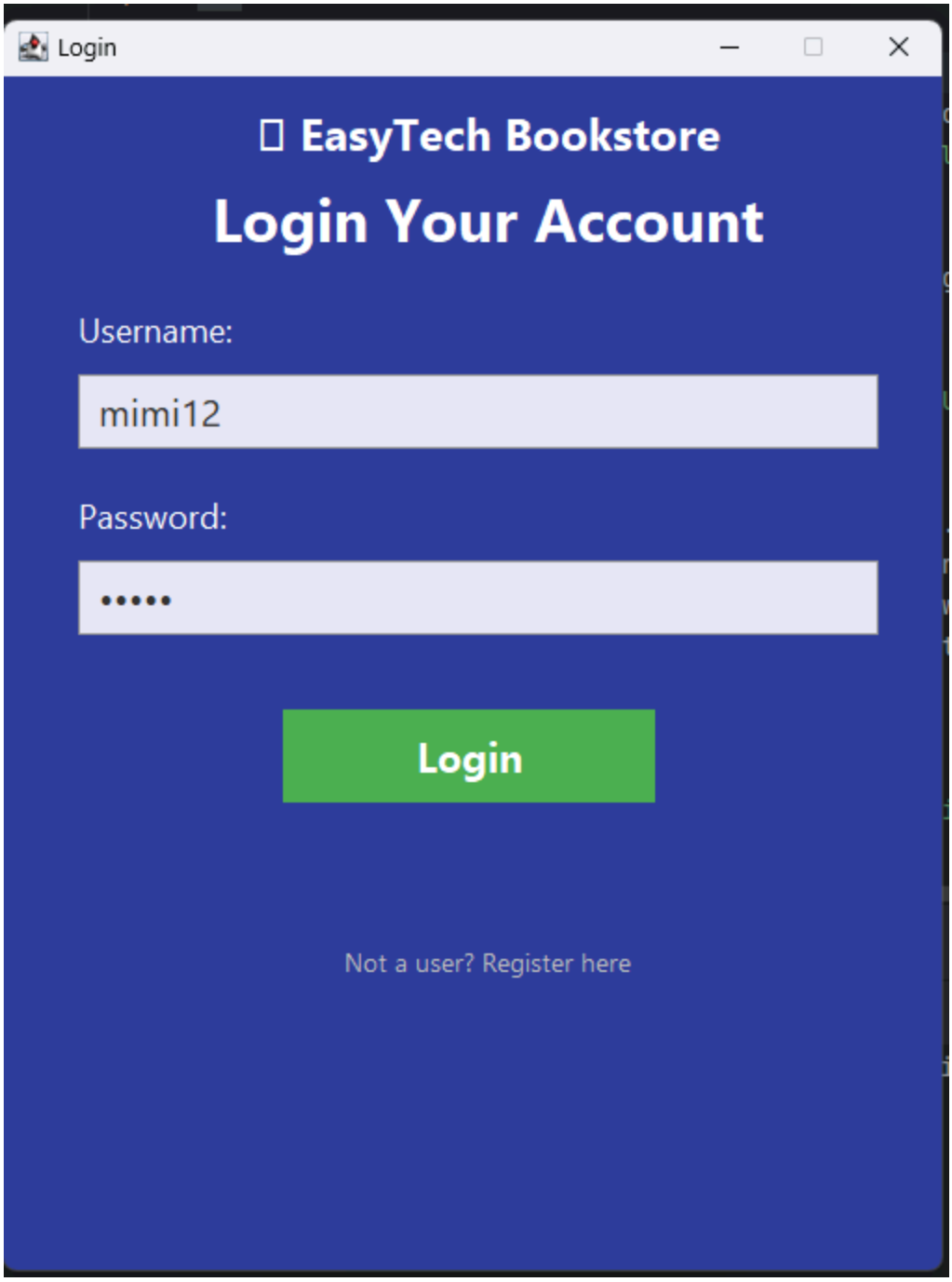
- **Multiple Payment Types:** Supports BKASH, NAGAD, and CARD payments.

- **Dynamic Input Fields:** Payment form adapts to show relevant input fields based on the selected payment type (e.g., mobile number for mobile payments, card number, CVV, expiry for card payments).
- **Input Validation:** Robust validation for payment details (e.g., 11-digit mobile number, 10-digit card number, CVV, expiry date format).
- **Payment History:** Displays a table of all past payments made by the logged-in user within the Payment GUI.
- **PDF Receipt Generation:** Automatically generates a PDF receipt for each successful payment, including transaction details.
- **Database Integration:** Payment records are stored in the `payments` table in MySQL.

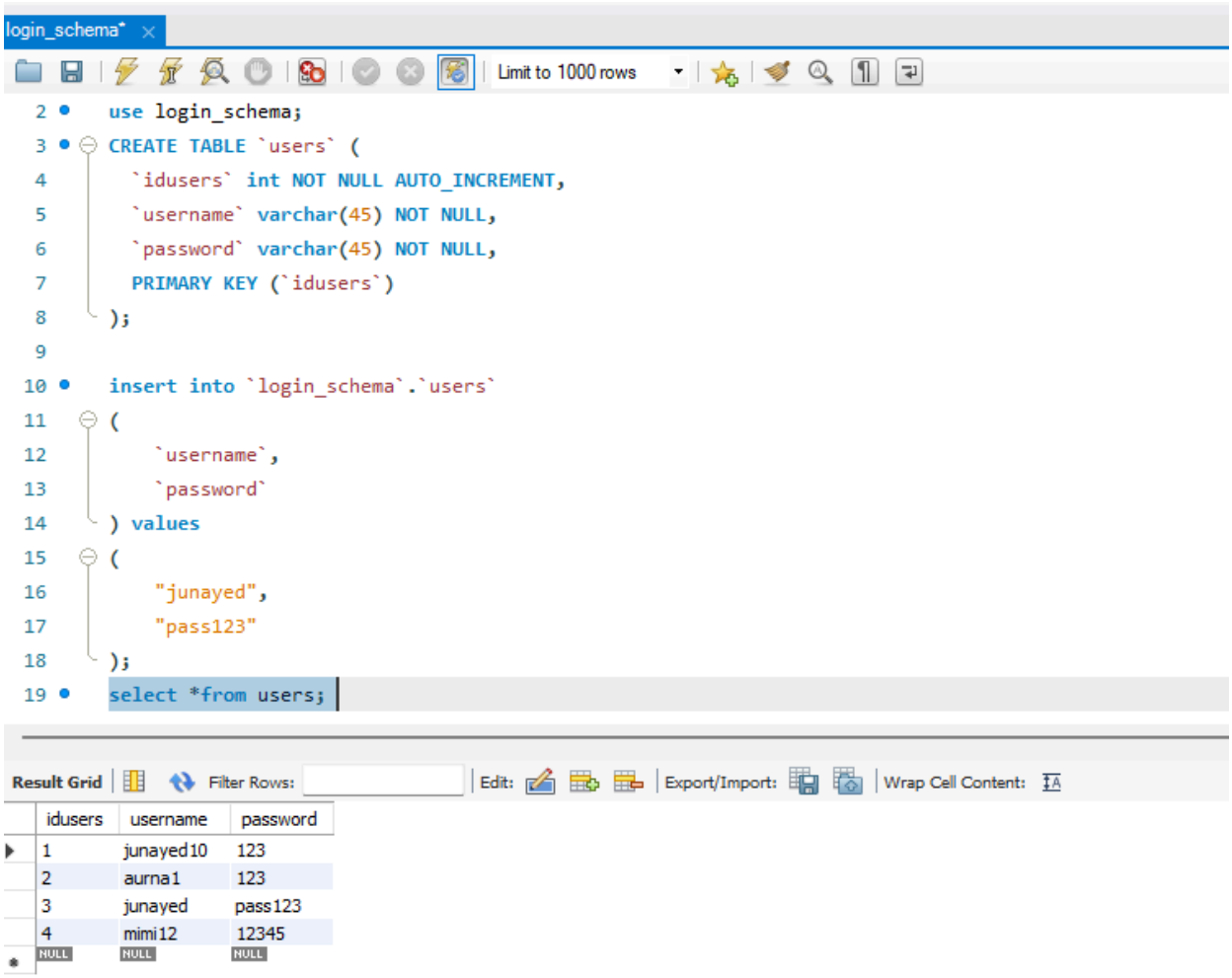
Screenshots of UI

Login and registration:

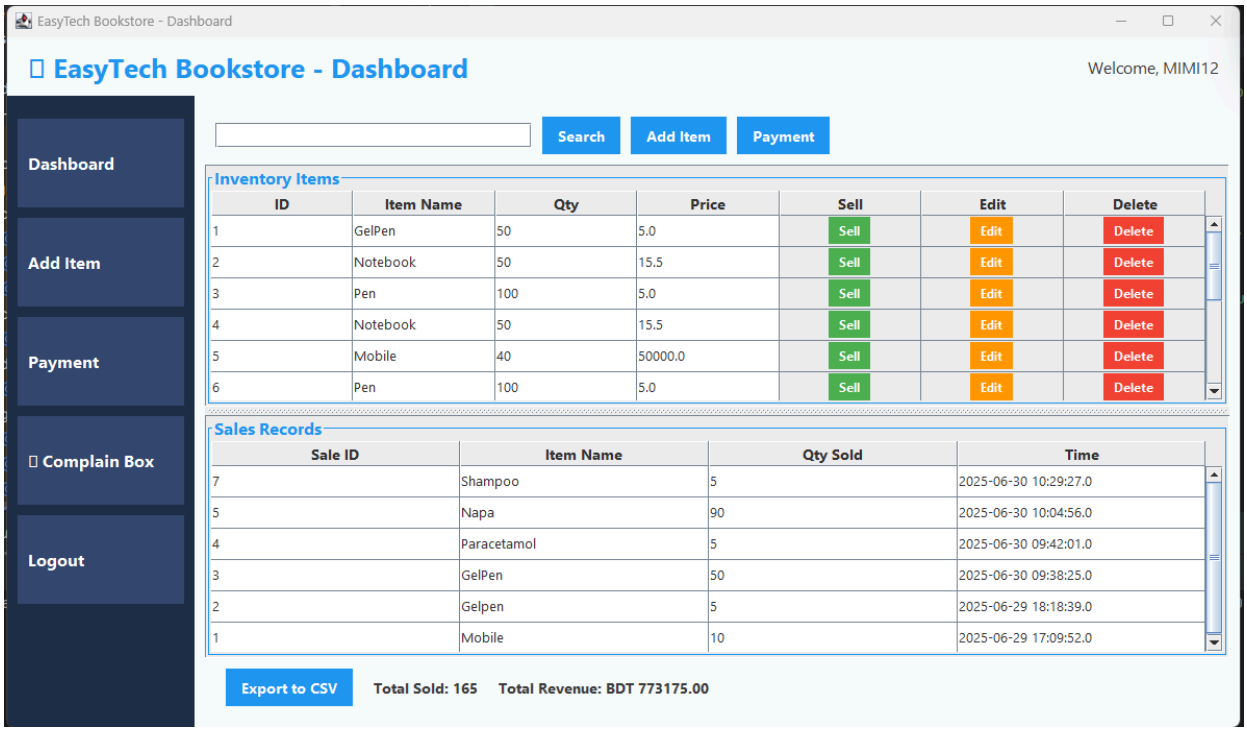

Register
EasyTech Bookstore
Sign Up
Username:
Password:
Re-enter Password:
Register
Have an account? Login here



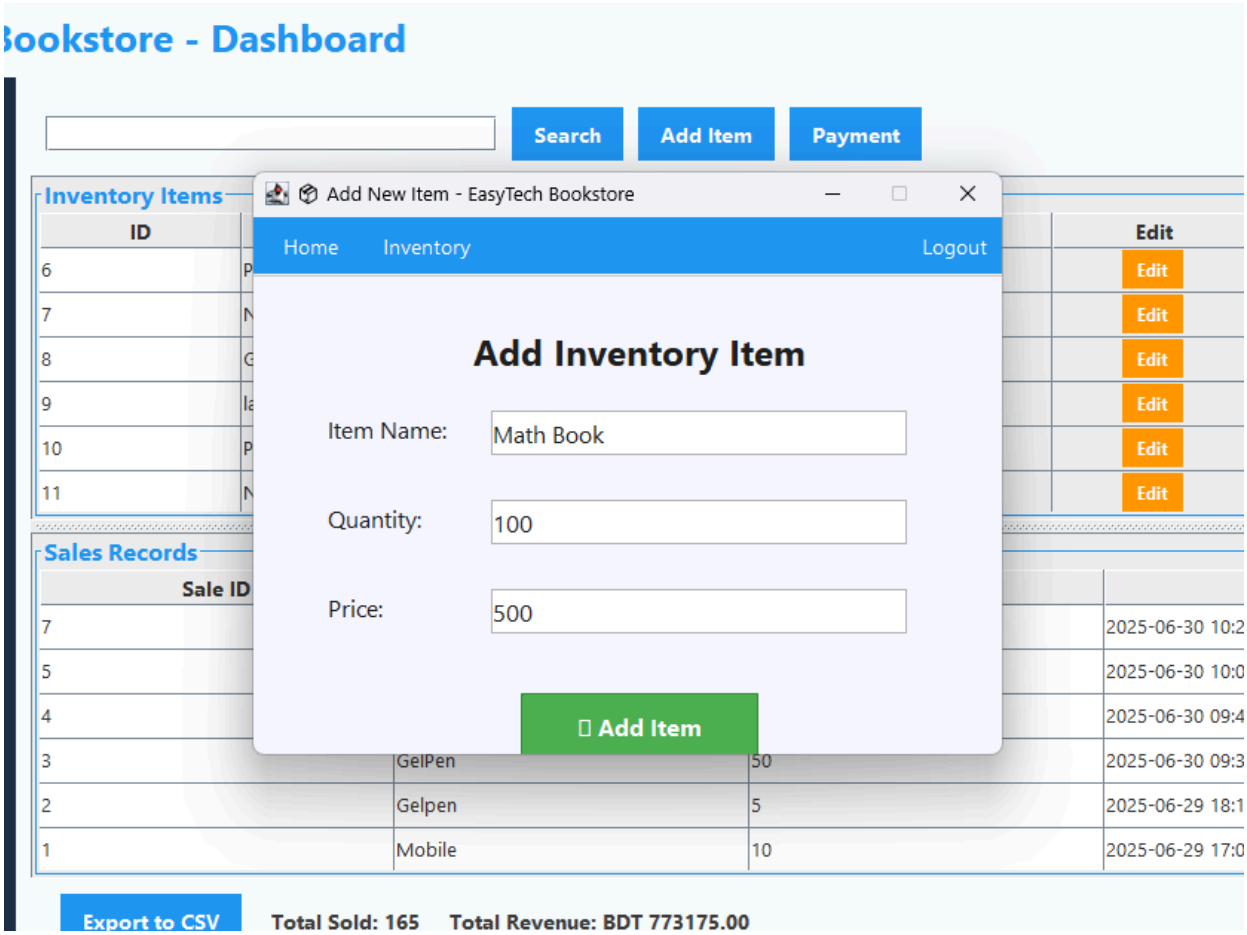
Then this gets in the Database



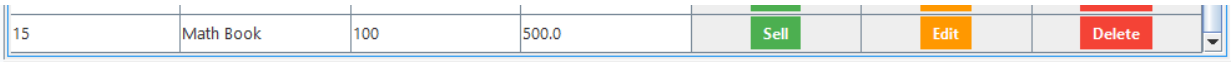
Then after login Dashboard appears


EasyTech Bookstore - Dashboard
EasyTech Bookstore - Dashboard
Welcome, MIMI12
Dashboard
Add Item
Payment
Complain Box
Logout
Search
Add Item
Payment
Inventory Items
ID
Item Name
Qty
Price
Sell
Edit
Delete
1 GelPen 50 5.0 Sell Edit Delete
2 Notebook 50 15.5 Sell Edit Delete
3 Pen 100 5.0 Sell Edit Delete
4 Notebook 50 15.5 Sell Edit Delete
5 Mobile 40 50000.0 Sell Edit Delete
6 Pen 100 5.0 Sell Edit Delete
Sales Records
Sale ID
Item Name
Qty Sold
Time
7 Shampoo 5 2025-06-30 10:29:27.0
5 Napa 90 2025-06-30 10:04:56.0
4 Paracetamol 5 2025-06-30 09:42:01.0
3 GelPen 50 2025-06-30 09:38:25.0
2 Gelpen 5 2025-06-29 18:18:39.0
1 Mobile 10 2025-06-29 17:09:52.0
Export to CSV
Total Sold: 165
Total Revenue: BDT 773175.00

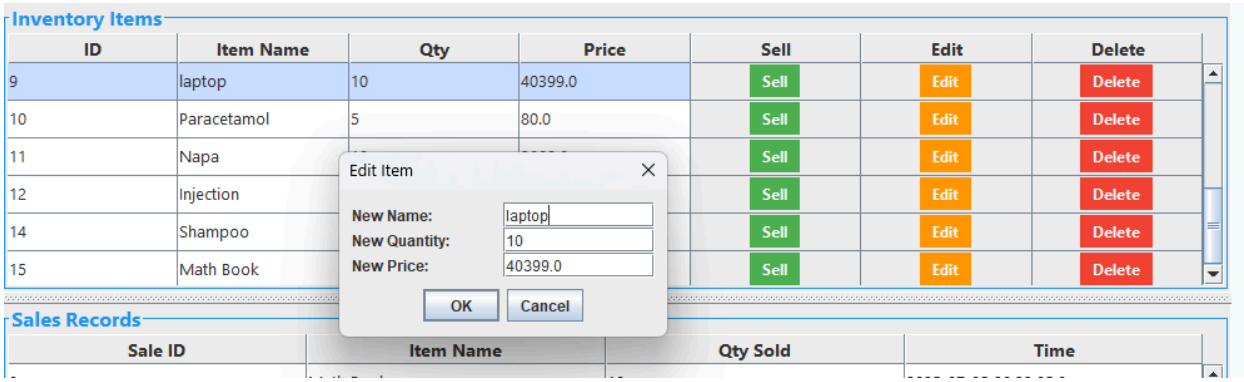
Can add items



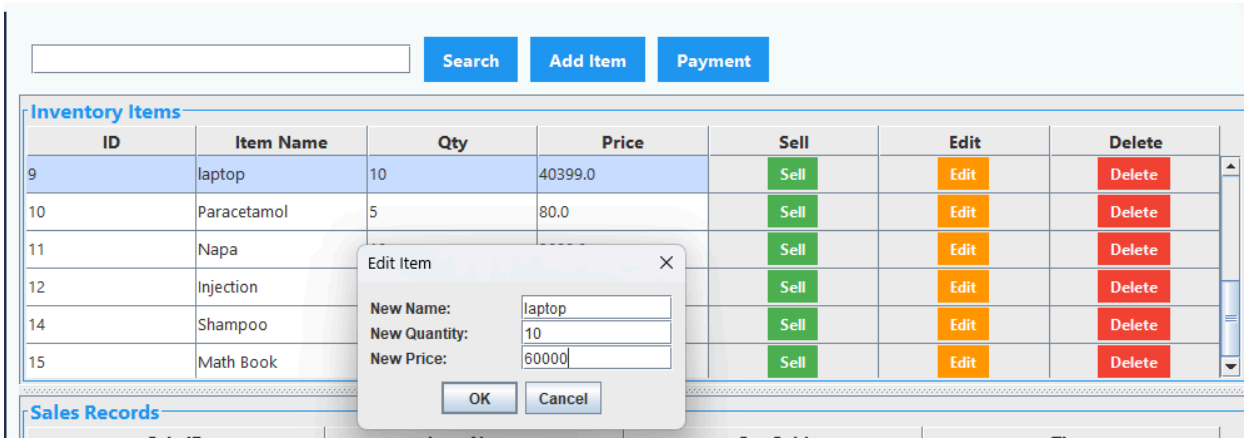
Thus added

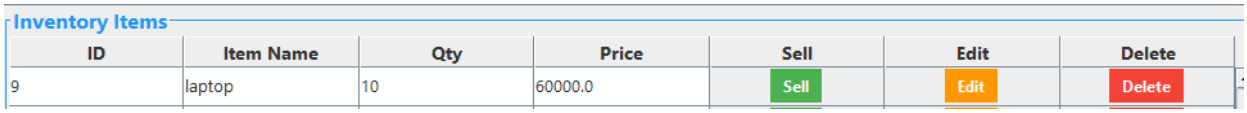


Can edit items

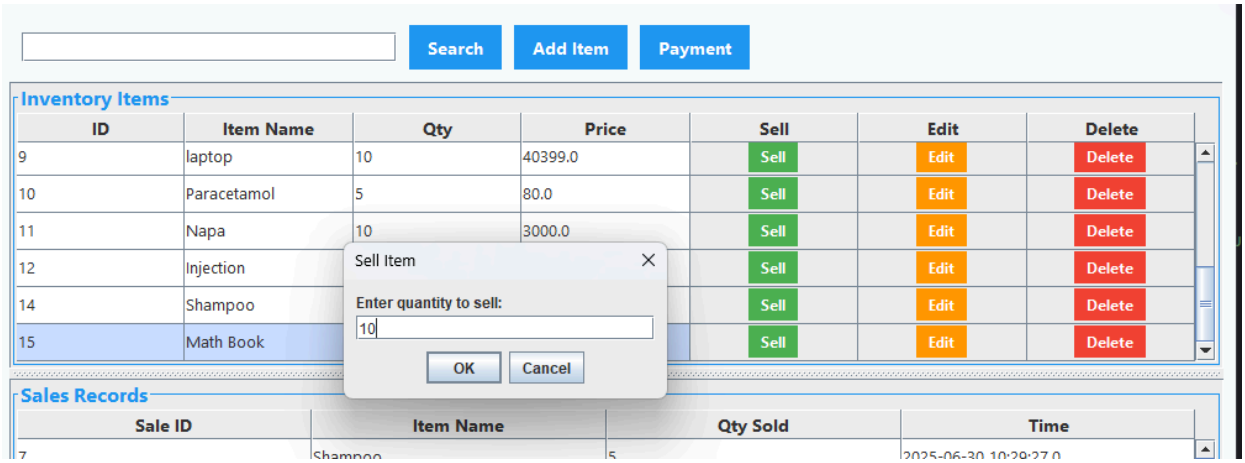


edited:

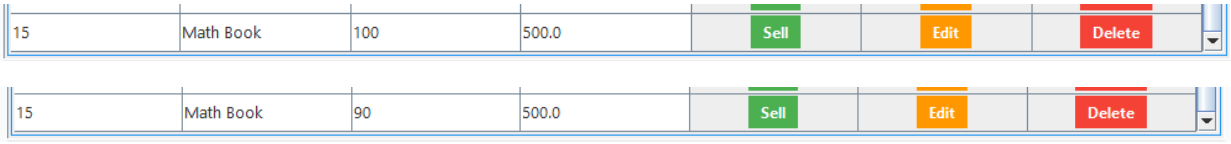




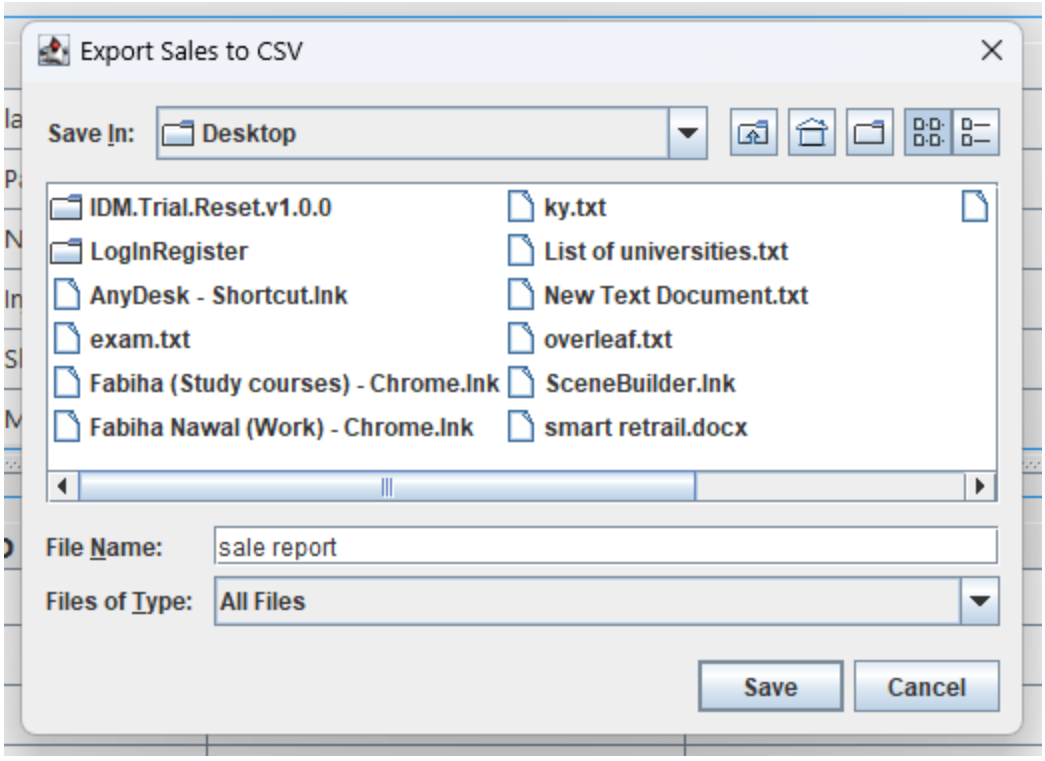
Can sell items:

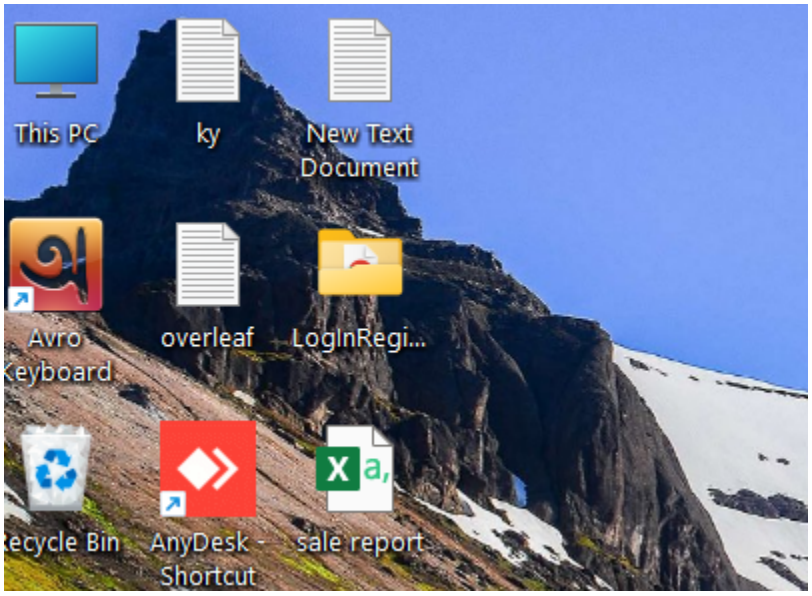


After selling Mathbook quantity changed.



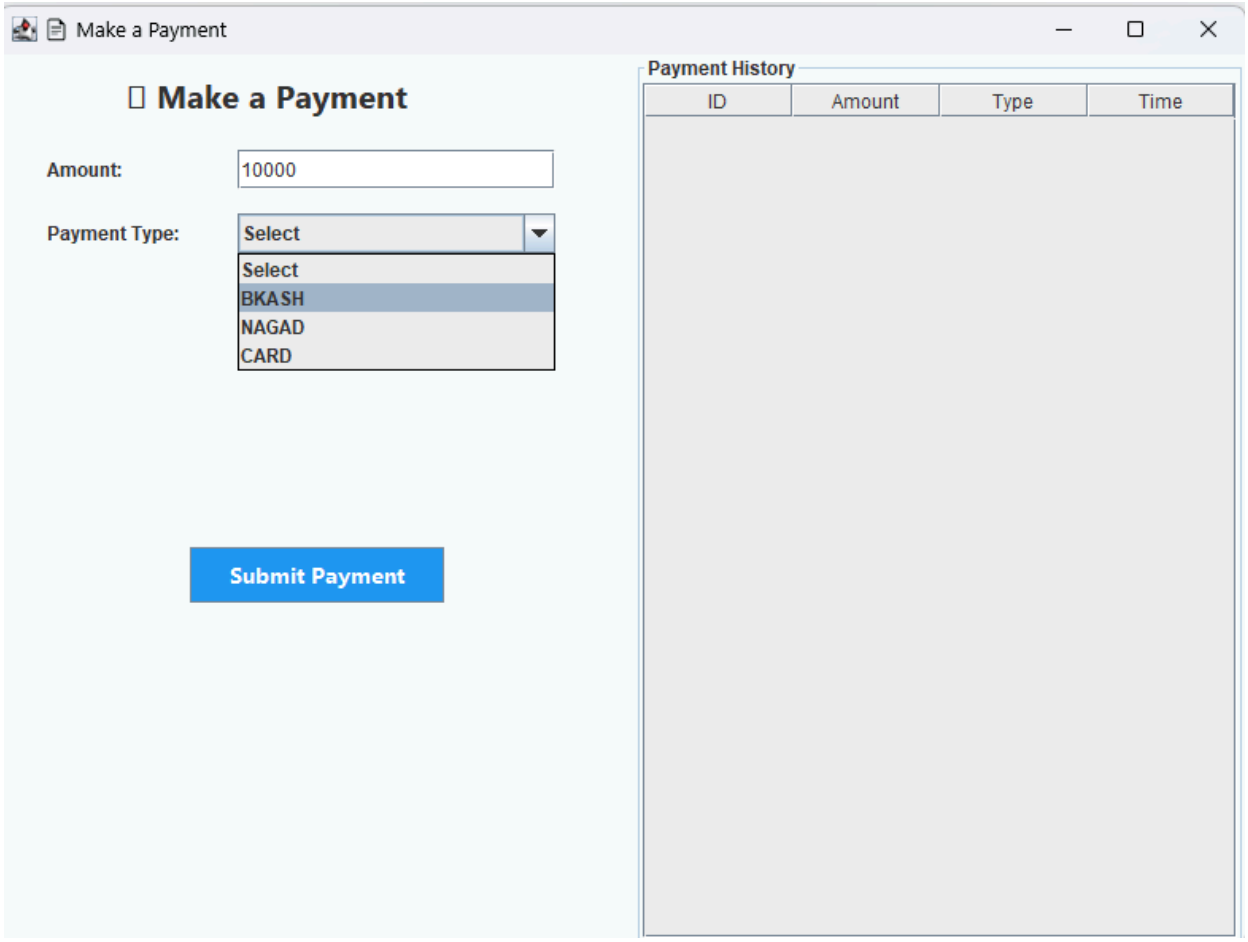
Can export to CSV file as sale report:

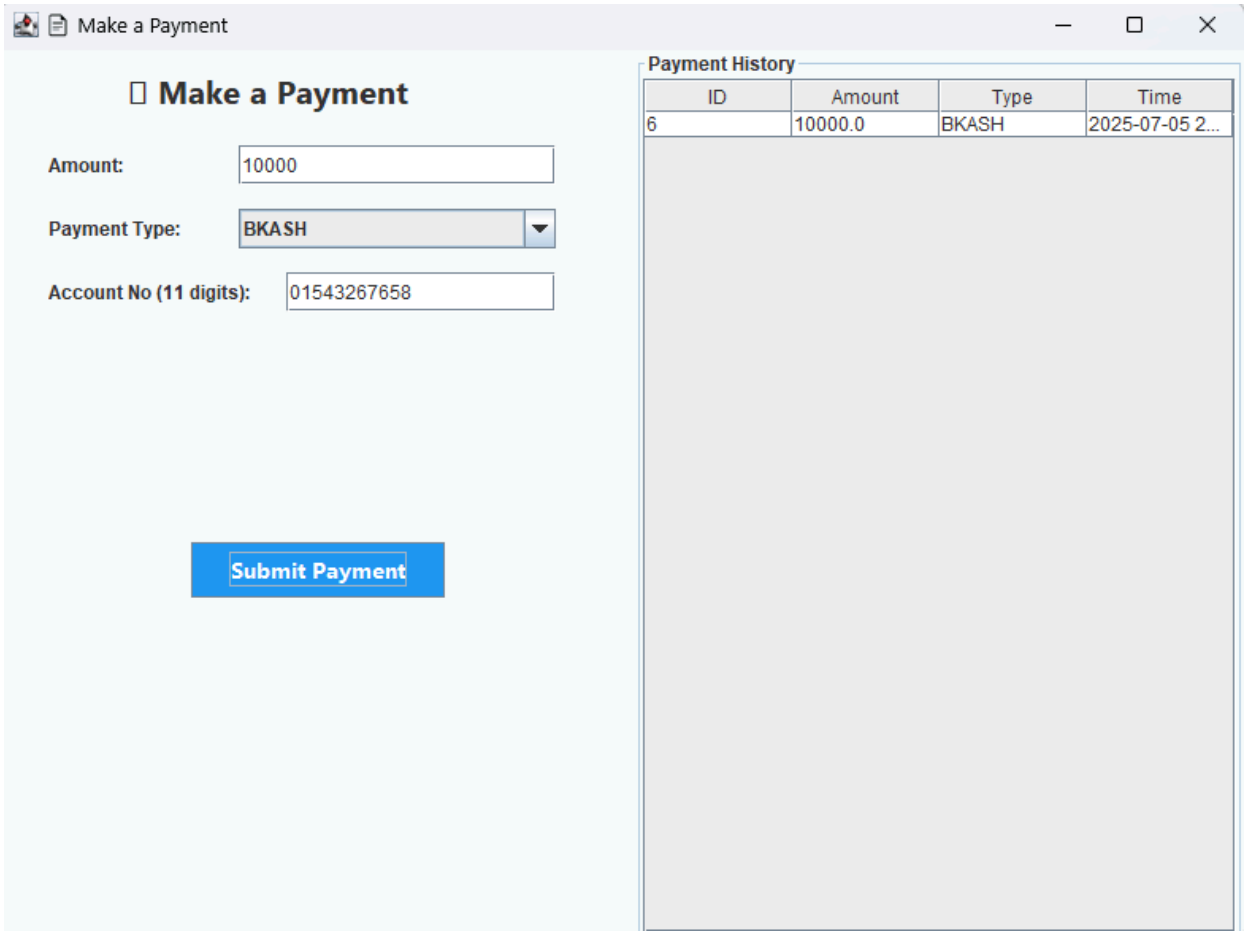


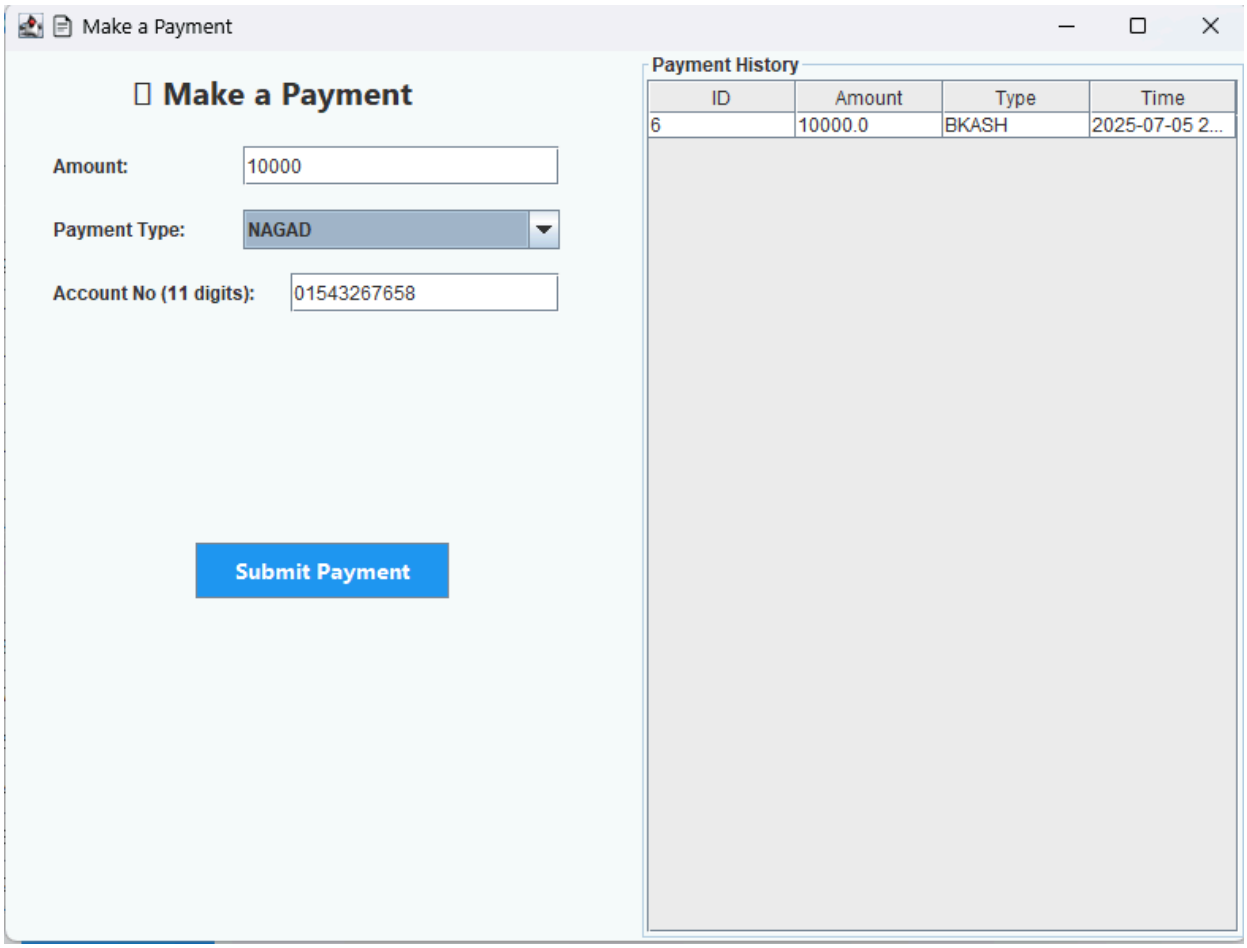


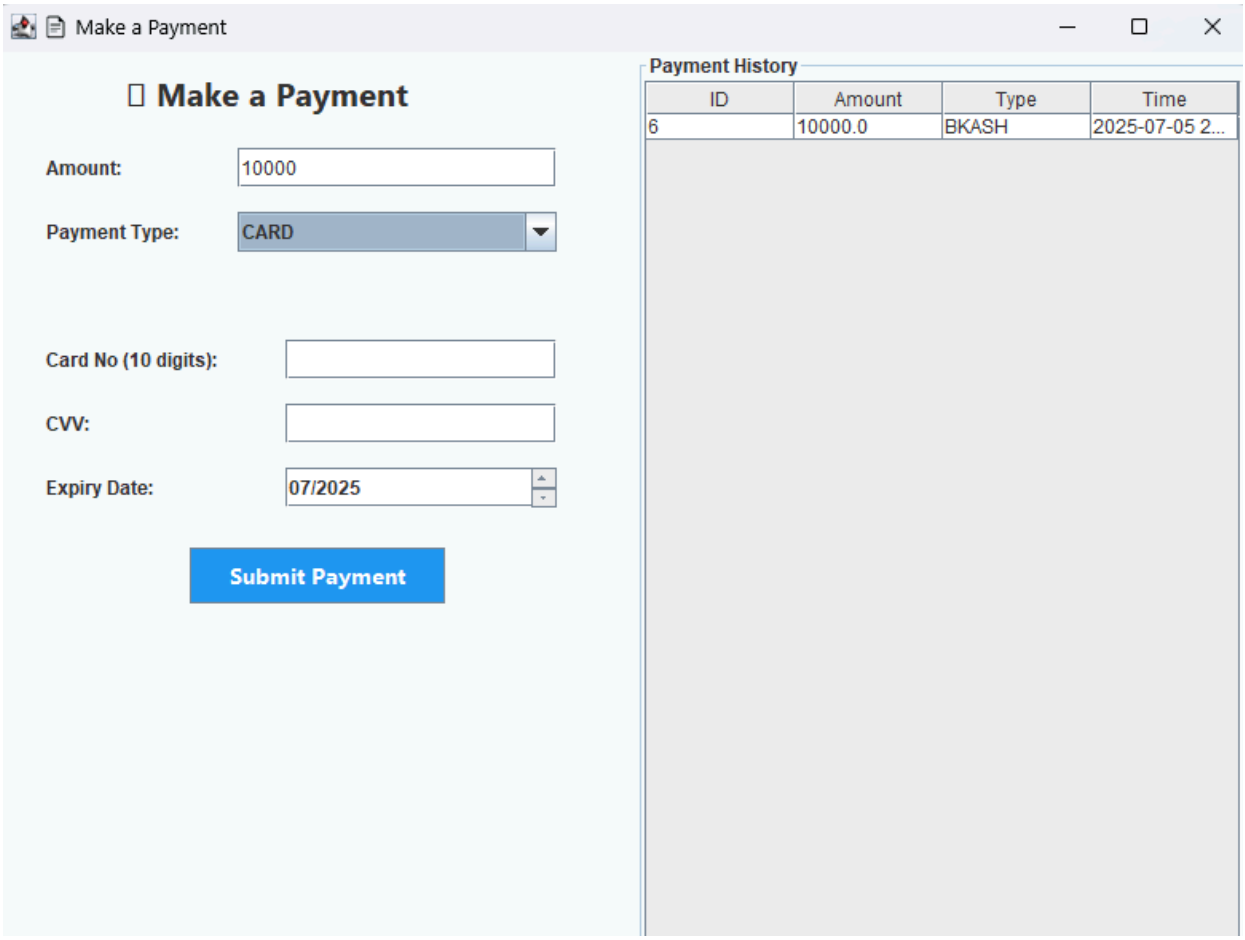
exported in the desktop as sale report

Can make payment in 3 types: Bkash, Nagad, card

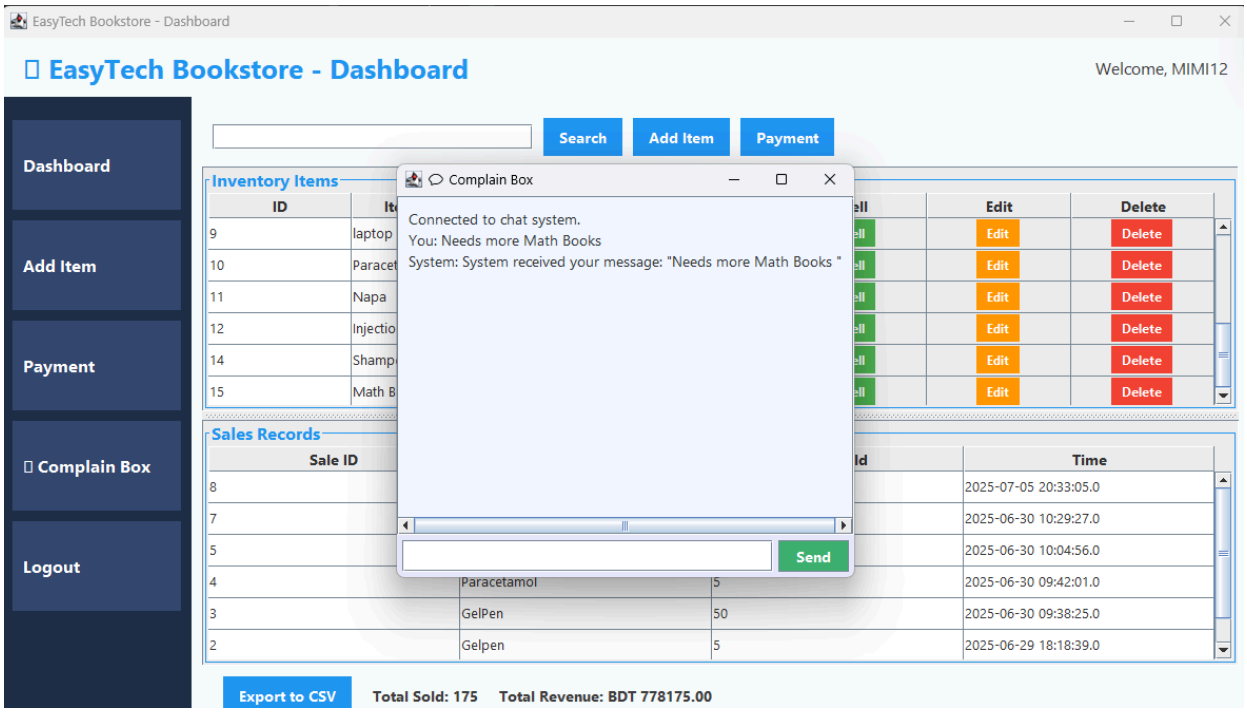

Make a Payment
Make a Payment
Amount:
10000
Payment Type:
Select
Select
BKASH
NAGAD
CARD
Submit Payment
Payment History
ID
Amount
Type
Time


Make a Payment
Make a Payment
Amount:
10000
Payment Type:
BKASH
Account No (11 digits):
01543267658
Submit Payment
Payment History
ID
Amount
Type
Time
6
10000.0
BKASH
2025-07-05 2...


Make a Payment
Make a Payment
Amount:
10000
Payment Type:
NAGAD
Account No (11 digits):
01543267658
Submit Payment
Payment History
ID
Amount
Type
Time
6
10000.0
BKASH
2025-07-05 2...



Can write any kind of complains taken from the customers for further improvement of the shop:



Conclusion

The *EasyTech Bookstore Inventory Management System* successfully achieves its goal of simplifying and digitizing bookstore operations. It combines real-time client-server communication with a robust database-driven backend and a user-friendly graphical interface. Key business functions such as inventory management, sales tracking, payment processing, and customer support are seamlessly integrated into one unified platform. The system not only reduces manual workload but also increases transparency and efficiency, making it a valuable tool for small and medium retail businesses. Furthermore, the implementation of socket programming and multithreading demonstrates a strong application of advanced programming concepts in real-world systems.