UE21CS352B - Object Oriented Analysis & Design using Java

Mini Project Report

"STORE – POINT OF SALES"

Submitted by:

Sushmitha H S	**PES1UG21CS650**
Swathi Karanth	**PES1UG21CS654**
Sriraksha M S	**PES1UG21CS625**
Thejas N	**PES1UG21CS674**

6th Semester K Section

Prof. Bhargavi Mokashi
Assistant Professor

January - May 2024

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

Table of Contents

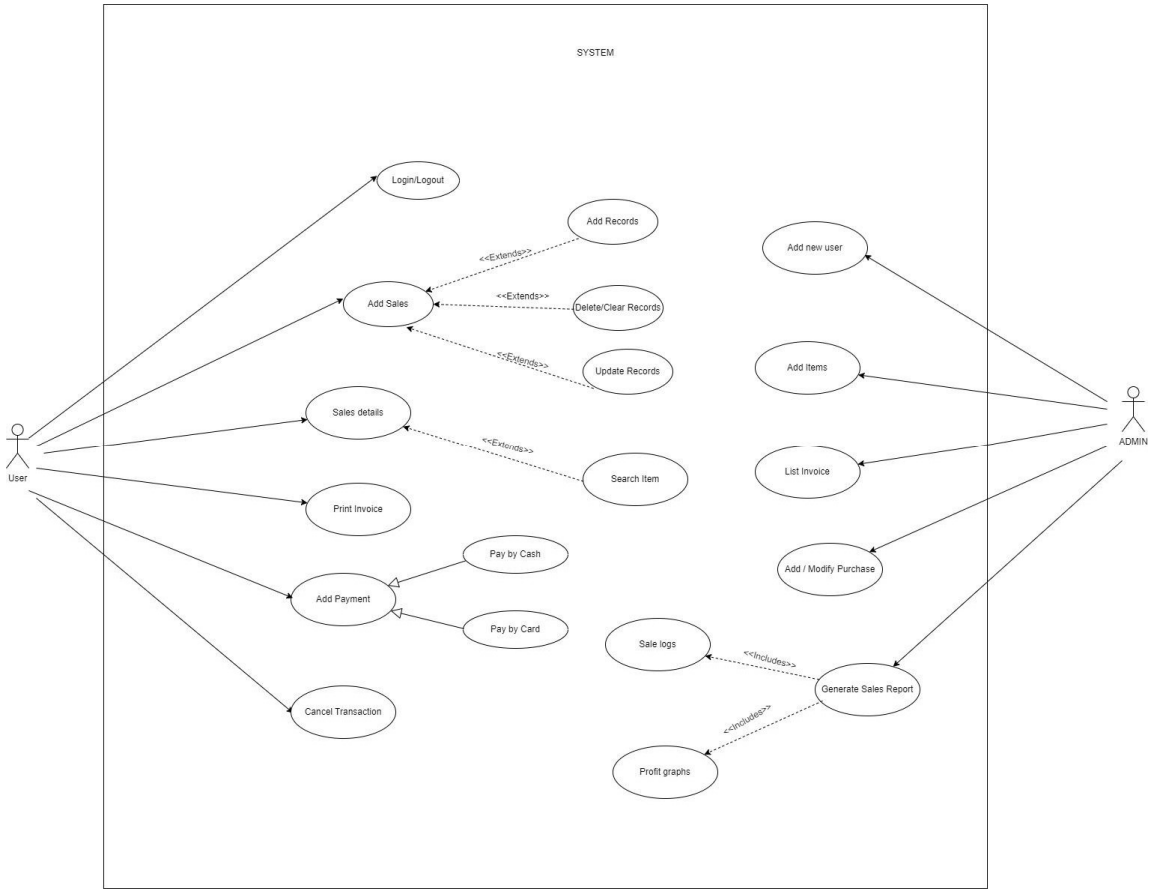
PROBLEM STATEMENT 3
MODELS 3
Use Case 3
Class Diagram 4
Activity Diagram – Major Use Case 5
Activity Diagram – Minor Use Case 6
State Diagram 7
DESIGN PATTERNS AND PRINCIPLES 8
MVC ARCHITECTURE 8
FACADE PATTERN 9
OBSERVER PATTERN 10
FACTORY PATTERN 12
LISKOV SUBSTITUTION PRINCIPLE 13
GITHUB LINK TO OUR CODE 13
INDIVIDUAL CONTRIBUTIONS 14
OUTPUTS 19

PROBLEM STATEMENT

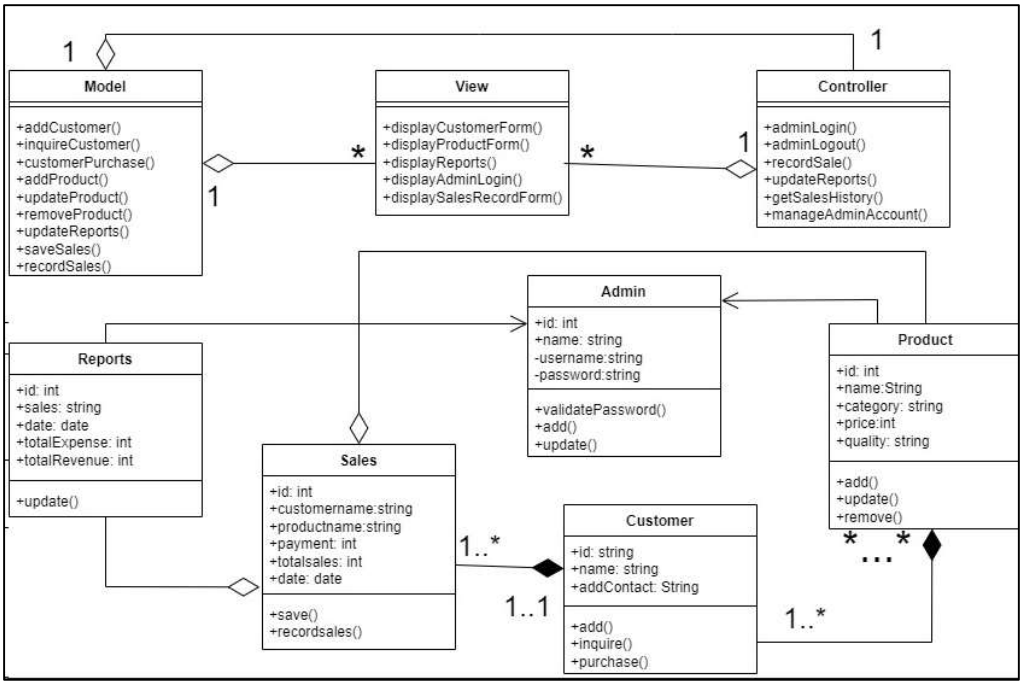
Point of sale (POS) software allows transactions between a company and its clients at the point of sale. POS software's main goals are to control inventory, expedite the sales process, and offer several other functions that improve customer support and business operations.

MODELS

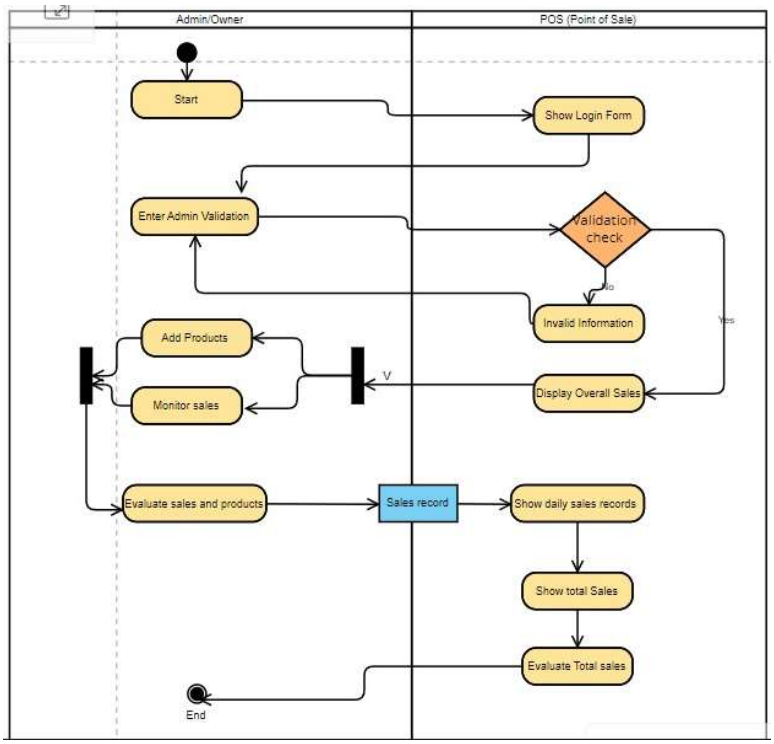
Use Case



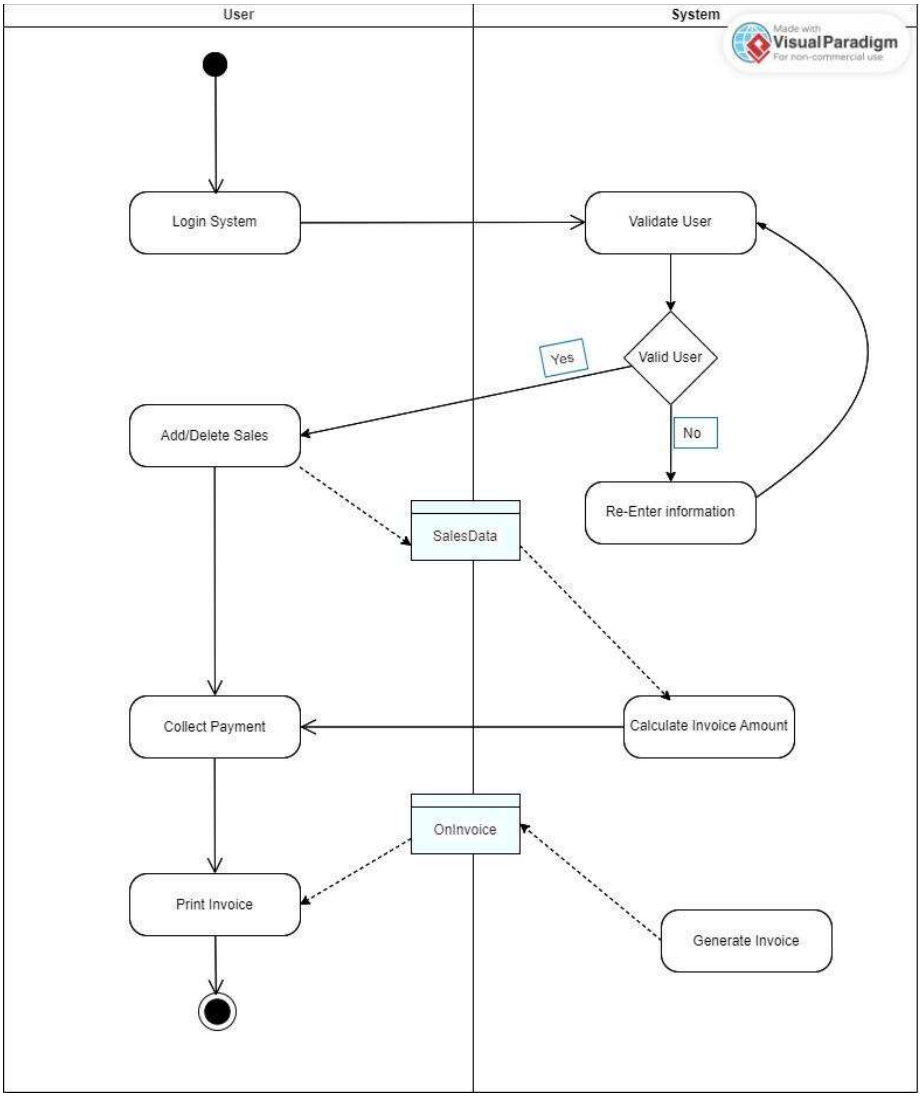
Class Diagram



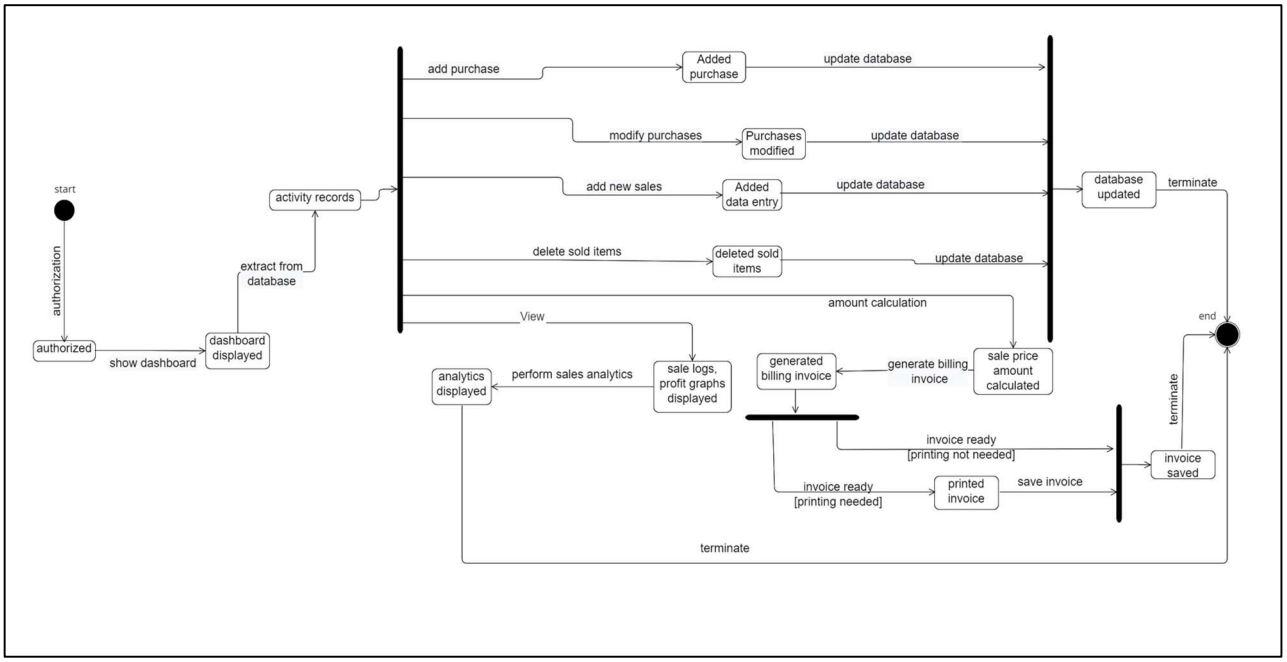
Activity Diagram – Major Use Case



Activity Diagram – Minor Use Case



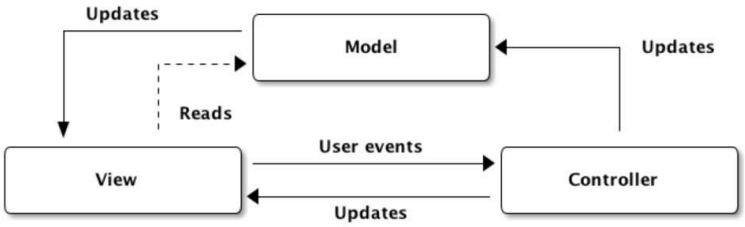
State Diagram



DESIGN PATTERNS AND PRINCIPLES

MVC ARCHITECTURE

Components of the MVC Design Pattern



- Model

The Model component in the MVC (Model-View-Controller) design pattern represents the data and business logic of an application. It is responsible for managing the application's data, processing business rules, and responding to requests for information from other components, such as the View and the Controller.

- View

Displays the data from the Model to the user and sends user inputs to the Controller. It is passive and does not directly interact with the Model. Instead, it receives data from the Model and sends user inputs to the Controller for processing.

- Controller

Controller acts as an intermediary between the Model and the View. It handles user input and updates the Model accordingly and updates the View to reflect changes in the Model. It contains application logic, such as input validation and data transformation.

MVC in the add sales functionality in the POS project

The AddSalesController class has references to the UI components defined in the AddSales.fxml file, allowing it to control and manipulate the View.

The AddSalesController class creates an instance of the SalesModel class and uses its methods to set and retrieve data related to the sales order.

When the user interacts with the UI, the controller handles these events and updates the Model accordingly.
The controller also updates the View based on changes in the Model. For example, when the user selects an item, the controller retrieves the corresponding item details from the database and updates the relevant UI components with the data.

The controller manages the flow of data between the Model and the View, ensuring that the View displays the correct data from the Model and that user input is properly handled and reflected in the Model.

The controller also handles additional functionality, such as printing invoices, performing calculations, and saving data to the database.

FACADE PATTERN

Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to an existing system to hide its complexities.

In our POS project, the AlertFacade class acts as the Facade, which encapsulates the complex logic of creating and showing alerts using the AlertHelper class.

The actual technical details for generating and presenting JavaFX alerts are included in the AlertHelper class. It provides a showAlert method that creates and displays the alert dialogue after receiving the required parameters. Additionally, it offers a way to record the user's reaction (Cancel or OK) by adjusting the static result variable.

The AlertFacade class provides a simplified and unified interface for creating and displaying alerts. The showAlert method in this class accepts the same parameters as the AlertHelper class but hides the implementation details. It simply delegates the call to the AlertHelper class.

The code for the AlertFacade class:

```java
package helper;

import javafx.scene.control.Alert;
import javafx.stage.Window;

public class AlertFacade {
    private static AlertHelper alertHelper = new AlertHelper();

    public static void showAlert(Alert.AlertType alertType, Window owner, String title, String message) {
        alertHelper.showAlert(alertType, owner, title, message);
    }
}
```

Code for the AlertHelper class

```java
package helper;

import java.util.Optional;
import javafx.scene.control.Alert;
import javafx.scene.control.ButtonType;
import javafx.stage.Window;

public class AlertHelper {
    public static boolean result = false;

    public static void showAlert(Alert.AlertType alertType, Window owner, String title, String message) {
        Alert alert = new Alert(alertType);
        alert.setTitle(title);
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.initOwner(owner);

        Optional<ButtonType> result = alert.showAndWait();
        if (result.get() == ButtonType.OK) {
            AlertHelper.result = true;
        } else {
            AlertHelper.result = false;
        }
    }
}
```

OBSERVER PATTERN

The Observer pattern is used to validate text fields dynamically in the register controller class.

Observer Interface (Observer):
This interface defines the contract for an observer. It has a single method update TextFieldObserver observer, which is called when the state of a text field observer changes.

TextFieldObserver Class:

This class represents the concrete observers in the pattern.
Each TextFieldObserver instance is associated with a TextField from JavaFX.

It has properties like minLength, maxLength, and errorMessage to define the validation rules and error message for the associated text field. The isValid() method checks whether the text in the associated text field meets the defined validation rules.
The getErrorMessage() method returns the error message associated with the observer.
getTextField() method returns the associated text field.

RegisterController Class:
This class serves as the subject in the Observer pattern. It implements the Observer interface. It maintains a list of TextFieldObserver instances that observe the text fields. In the initialize() method, it adds TextFieldObserver instances for each text field to the observers list. In the update() method, which is from the Observer interface, it handles updates from the observers. In this case, it shows an error alert if the observed text field is not valid.

The isValidated() method iterates through all observers and checks if any text field is invalid. If the username is already registered, it also displays an error alert.
In the register() method, it checks if the form is validated before attempting to insert the user data into the database.
The showLoginStage() method switches the scene to the login view when the user clicks on the corresponding button.

Code for the observer pattern:

```
interface Observer {
    void update(TextFieldObserver observer);
}
```

```
@Override
public void initialize(URL url, ResourceBundle rb) {
    observers.add(new TextFieldObserver(firstName));
    //instances of TextFieldObserver are created for each text field that requires observation
    observers.add(new TextFieldObserver(lastName));
    observers.add(new TextFieldObserver(email));
    observers.add(new TextFieldObserver(username));
    observers.add(new TextFieldObserver(password));
    observers.add(new TextFieldObserver(confirmPassword));
    for (TextFieldObserver observer : observers) {
    observer.getTextField().textProperty().addListener((observable, oldValue, newValue) -> {
        update(observer);
    });
}
}
```

FACTORY PATTERN

The Factory Pattern is used to create instances of SalesModel objects in the ListSalesController class.
Factory Class (SalesModelFactory):

The SalesModelFactory class contains a static method createSalesModel() which is responsible for creating instances of SalesModel.
The method createSalesModel() accepts various parameters required to initialize a SalesModel object.
Inside the method, a new instance of SalesModel is created using the provided parameters and returned to the caller.

Usage in ListSalesController:

In the ListSalesController class, the createData() method is responsible for populating the data in the TableView.
Instead of directly creating instances of SalesModel within the createData() method, it utilizes the SalesModelFactory.createSalesModel() method to create instances.

The createData() method retrieves data from the database and then calls SalesModelFactory.createSalesModel() to create SalesModel instances with the retrieved data.

This approach decouples the creation of SalesModel instances from the ListSalesController class, making it more flexible and easier to maintain.

If there are any changes in the way SalesModel objects are created, they can be handled within the SalesModelFactory class without affecting the ListSalesController class.

```java
package controller;

import model.SalesModel;

public class SalesModelFactory {

    public static SalesModel createSalesModel(long orderId, String invoiceDate, String partyName, String curr
        // Create and return a new SalesModel instance with the provided parameters
        return new SalesModel(orderId, invoiceDate, partyName, currency, taux, totalQuantity, totalAmount, ot
    }
}
```

LISKOV SUBSTITUTION PRINCIPLE

The AmountCalculator class extends the EditSalesController class, which means it inherits all the fields and methods from the base class.

The AmountCalculator class overrides two methods from the base class: calculateDueAmount and calculateTotalAmount.

In both overridden methods, AmountCalculator does not introduce any new functionality or behavior that would violate the contract of the base class methods.

The calculateDueAmount method calculates the due amount based on the total payable amount and the paid amount, which is the expected behavior of this method in the base class.

The calculateTotalAmount function, the normal behaviour of this method in the base class, determines the total amount by adding the amounts of each individual item in the tableViewItem and taking textFieldTotalOther into account.
The two methods in the AmountCalculator class access and utilize identical fields and properties as the base class.

GITHUB LINK TO OUR CODE

https://github.com/sushihebbar/POS

INDIVIDUAL CONTRIBUTIONS

Sriraksha: In our Point of Sales project, my contributions were focused on implementing key controllers following the GRASP and SOLID principles, along with integrating design patterns for efficient code organization. Specifically, I developed:

My contributions include:

- **ListPurchaseController:**
  - This controller manages the list of purchases, with a clear separation of concerns according to the Single Responsibility Principle (SRP).
  - The purchases in the database are listed and the same is displayed when loaded.
  - The controller allows users to interact with the displayed data by selecting records for viewing or deletion.
- **ListPurchaseReturnController:**
  - It is responsible for handling purchase returns, maintains high cohesion and low coupling within the system.
  - It retrieves data from a database and displays order ID, invoice date, party name, and financial amounts.
  - The return orders are initiated here and an invoice bill for the same is generated once the transaction process is completed.
- **MainPanelController:**
  - Central to our project's interface, it controls the main panel interface of the application, providing buttons for various functionalities like adding purchases, viewing details, handling returns, and managing sales. It handles the navigation between different views within the application.
  - It retrieves data from the database to create graphical representations of financial transactions using JavaFX charts.
- **ListSalesReturnController:**
  - Manages sales returns with the Open-Closed Principle (OCP).
  - The controller responds to user actions such as selecting and deleting sales return records.
- **ListSalesController with Factory Design Pattern:**
  - This controller has the implementation of the Factory design pattern, which dynamically creates instances of sales lists.
  - The controller responds to user actions like selecting and deleting sales records.
  - It provides functionality to view detailed information about a selected record and asks for confirmation before deleting any record from the database.
  - These controllers collectively demonstrate our commitment to clean, maintainable code that aligns with best practices and design principles.

Sushmitha: My contributions to the Point of Sales project included integrating the observer pattern into the register controller and developing the following classes:

★ **AddSalesController** ->
- This class is responsible for adding items that can be selected from the dropdown which autocompletes the text for the user and also sets the price of the chosen item.
- The class utilizes JavaFX components like TextField, ComboBox, TableView, and DatePicker to create an interactive interface for users.
- This class also includes methods for saving the sales transaction data to the Oracle database and calculating total amounts.
- Jasper Viewer is used to display the invoice which can be printed as well. Each sale is stored in our database and displayed in the Sales Details functionality.

★ **RegisterController** ->
- The RegisterController manages user registration by validating input fields, checking for existing usernames, and inserting user data into the database upon successful registration.
- It utilizes TextFieldObservers to monitor text fields for input length constraints and displays error messages accordingly.
- If the user already has an account or created a new one, they can switch to the login page.
- The observer pattern promotes maintainability and modularity.

★ **AddItemsViewController** ->
- The AddItemViewController in the JavaFX application handles the task of item addition to purchase orders.
- Through its user-friendly interface, it allows users to input various item details like item code, name, quantity, and price.
- Users can seamlessly add, edit, or delete items as needed, providing flexibility and ease of use.
- After saving, a dialog notifies the user that the data has been successfully stored in the database.
- It also incorporates error handling and validation mechanisms to validate user inputs.

★ **EditPurchaseControlle**r ->
- This controller manages user input for editing purchase orders.
- It makes it easier to alter purchasing information such as item names, pricing, and quantities.
- It enables easy data entry with features like auto-completion and real-time computations of payable, paid, and due amounts. It adheres to the MVC architecture.

Thejas: I ensured code quality and maintainability by implementing key controllers adhering to Liskov and SOLID principles. Additionally, I leveraged design patterns for efficient organization.

★ **Login Controller**->
- The code handles user authentication by validating username and password against a database. If valid, it grants access to the main panel view.
- It utilizes JavaFX for managing the user interface, including loading different views, displaying alerts for errors, and facilitating navigation between login and registration stages.
- Demonstrated by the subclass GuestLoginController, which extends LoginController. This subclass inherits login functionality and adds specific features for guest user login, adhering to the Liskov Substitution Principle.
- Ensures user input is validated and appropriate error messages are displayed for blank or invalid fields. It maintains length constraints for username and password, providing a seamless user experience.

★ **Add purchase Controller** ->
- The code facilitates user input for purchase details such as item name, quantity, price, etc., with features like auto-completion for item and customer names to enhance user experience.
- It connects to a database to fetch item details based on user input and saves purchase records into the database using JDBC connections, enabling seamless data management.
- The code includes functionality for generating printable invoices using JasperReports, allowing users to create and view invoices for the purchases made within the application.

★ **Edit Sales Controller** ->
- Manages the user interface components defined in the corresponding FXML file, initializes the controller by setting up event handlers, table columns, and initial data population, facilitating user interaction and interface functionality.
- Utilizes JDBC for connectivity, executing SQL queries to retrieve and update sales return data from related tables, including items, customers, and sales return details.
- The controller manages all aspects of sales return functionality, including item and customer search, list manipulation, amount calculation, and user interaction handling.

★ **Add Sales Return Controlle**r ->
- The AddSalesReturnController class implements the Initializable interface, responsible for initializing the controller and setting up various UI components and event handlers for a sales return management system.

- It interacts with a database using JDBC to retrieve and update data related to items, customers, and sales return details, executing SQL queries for database connectivity and manipulation.

Swathi: In the Point of Sales project, I have incorporated the facade design pattern into the alert helper functionality. Additionally, I have developed the following classes:

★ **AddPurchaseReturnController** ->
- The AddPurchaseReturnController is responsible for managing purchase return orders within the Point of Sales system.
- It handles item and customer search functionalities, facilitates the addition of items with their details to a table view, and calculates total amounts.
- The controller enables users to save order information to the database, generate invoices using JasperReports, and clear the form for new entries.
- It makes purchase return orders easier with features like auto-completion and dynamic population when the user selects an item.

★ **EditPurchaseReturnController ->**
- The EditPurchaseReturnController manages the user interface and database interactions related to editing purchase return details in a business application.
- The controller initializes UI components, sets up auto-completion for text fields, and facilitates the overall process of editing purchase return records.
- This controller uses pagination to display a list of purchase returns, fetching data from a database.
- It handles tasks such as retrieving and displaying data, calculating totals, adding and deleting items, and saving changes to the database.

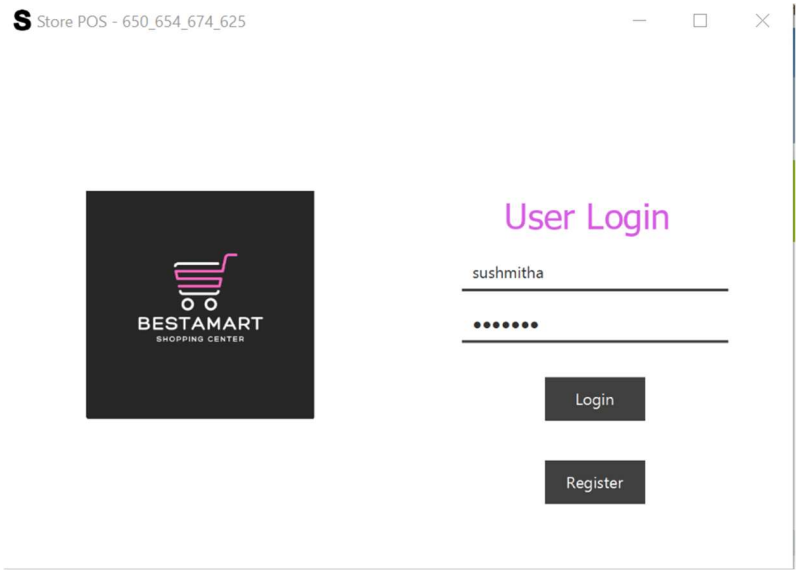
★ **EditSalesReturnController ->**
- The EditSalesReturnController helps users edit the sales return details in the point of sales system.
- Its functionality includes initializing UI components, retrieving sales return data from the database, allowing users to edit and update this data, calculating totals for quantities and amounts, and handling user interactions such as adding, deleting, and updating sales return items.
- The controller also incorporates features like auto-completion for item and customer selection as well as error handling for database queries and user inputs.

★ **AlertHelper and AlertFacade ->**

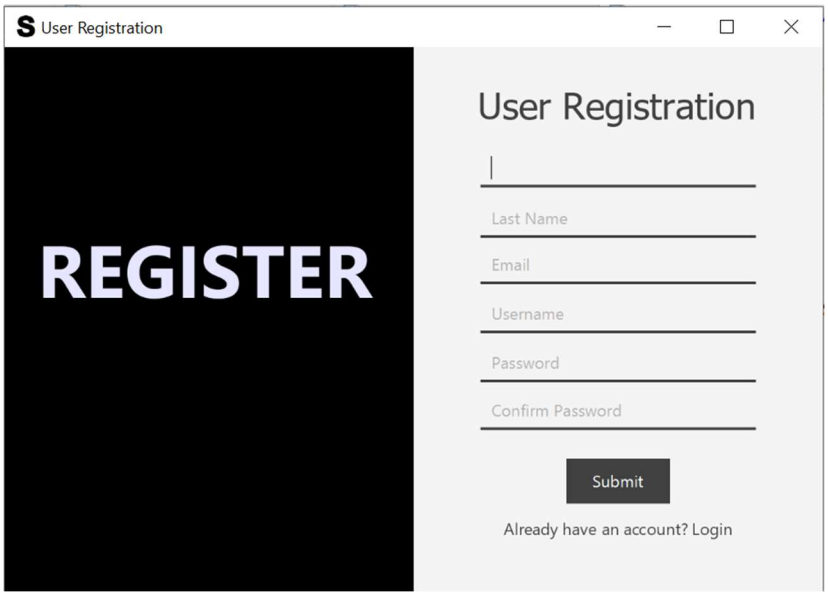
- AlertFacade serves as a facade for displaying alerts by delegating the task to AlertHelper.
- It offers a static method showAlert that accepts parameters such as alert type, owner window, title, and message and displays alerts from any part of the application.
- The AlertHelper encapsulates the logic for creating and showing JavaFX alert dialogs. It handles the user's response to the alert, setting a static boolean result variable based on whether the user clicked "OK" or not.
- This design promotes reusability and maintainability.

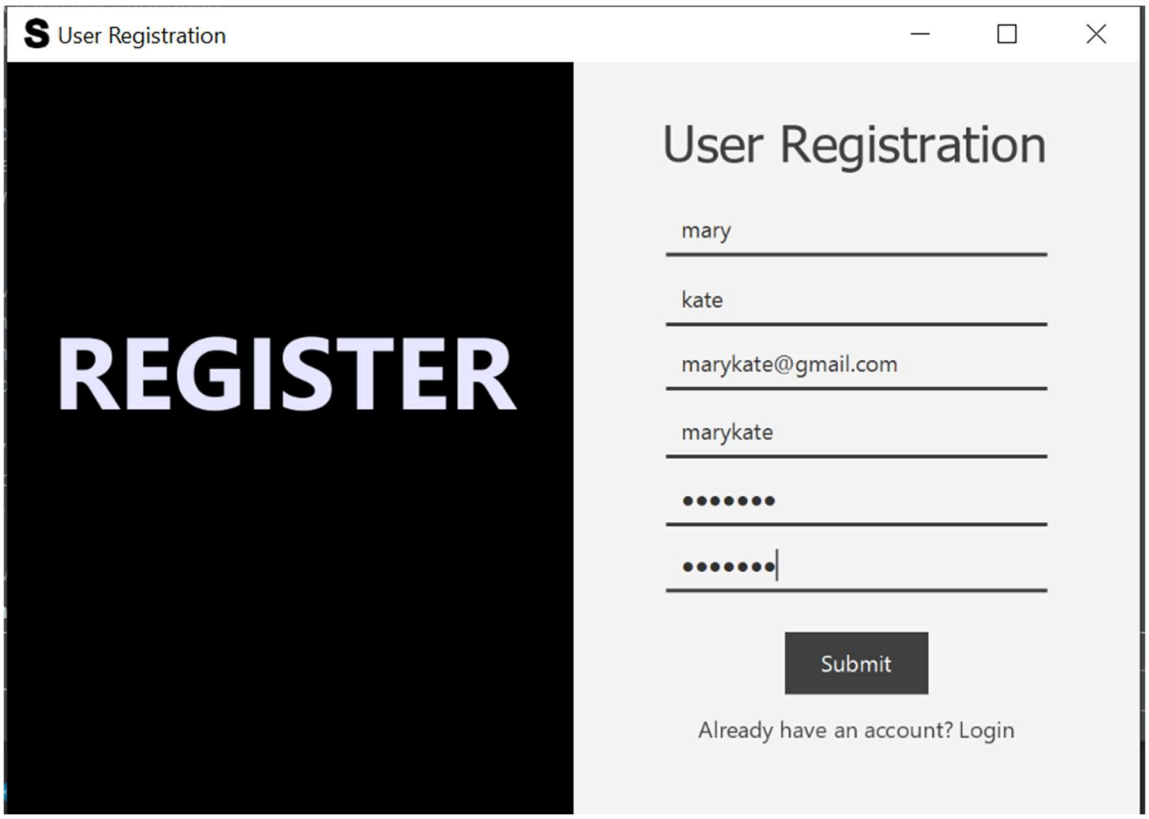
OUTPUTS

Login Page View



User Registration



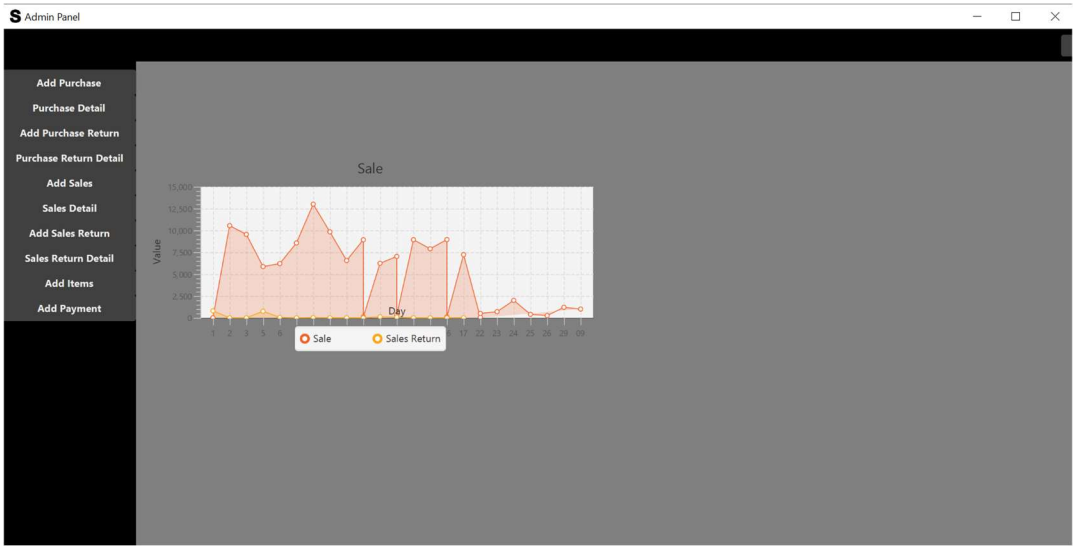


User Registered in Database

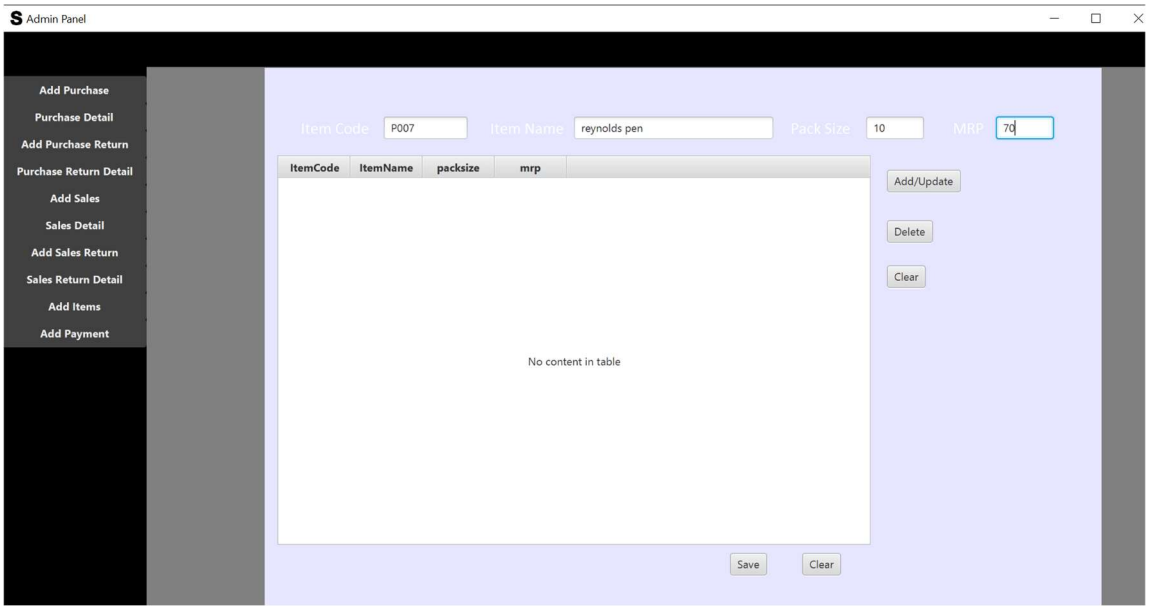
select * from pos.users Enter a SQL expression to filter results (use Ctrl+Space)

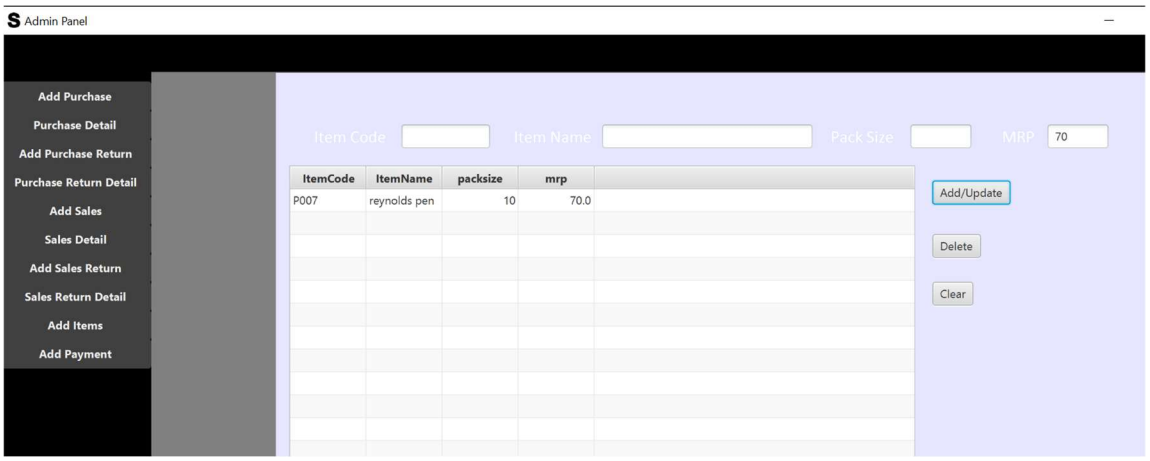
	FIRST_NAME	LAST_NAME	EMAIL	USER_NAME	PASSWORD
1	John	Doe	john.doe@example.com	johndoe	john@123
2	sush	hs	abc@gmail.com	sushmitha	sush123
3	alex	meyers	alexm@gmail.com	alexmeyers	alex@123
4	regina	george	reginag@gmail.com	reggie	reg123
5	camden	ryder	camden@gmail.com	camdenr	camden123
6	mary	kate	marykate@gmail.com	marykate	1234567

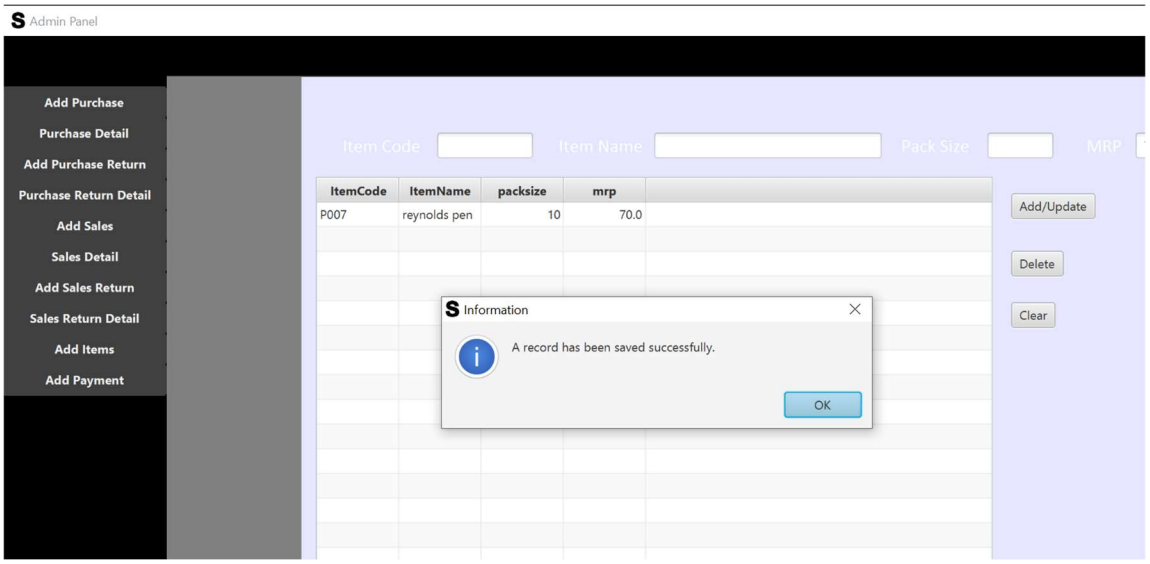
Main Panel View



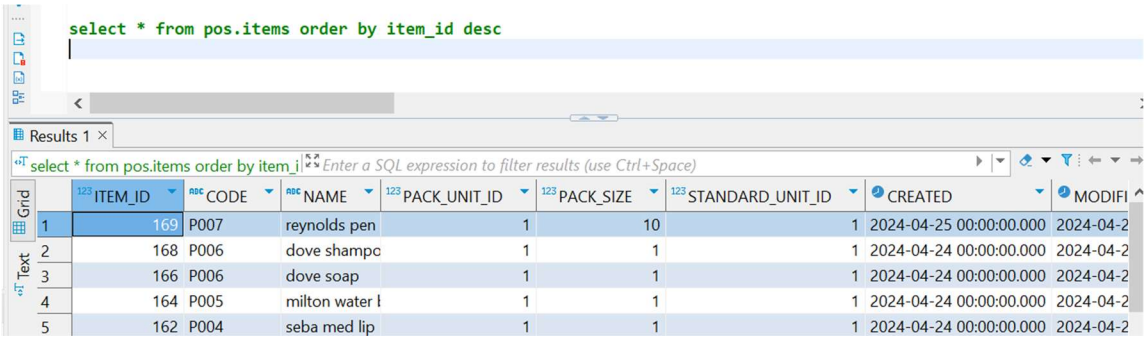
Add Items







Add Items reflected in the Oracle database



```
select * from pos.items order by item_id desc
```

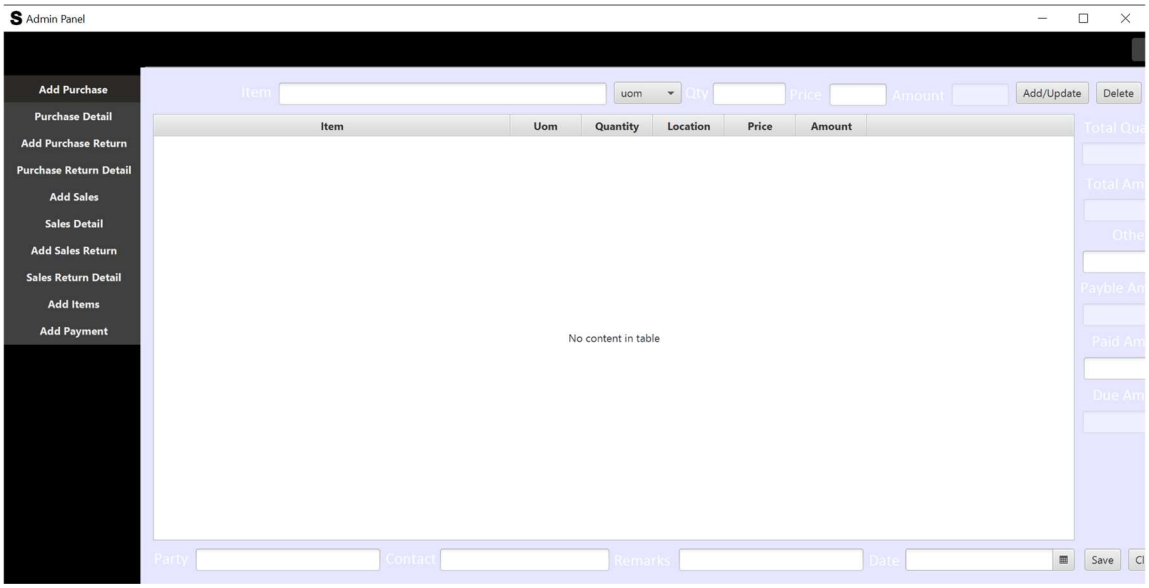
Results 1

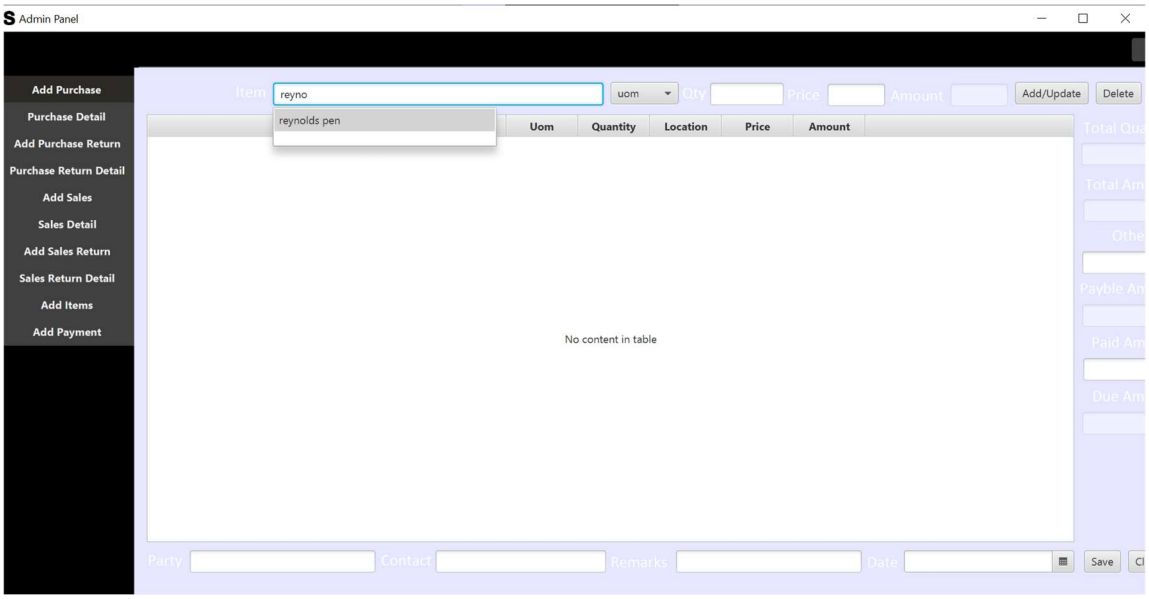
select * from pos.items order by item_i

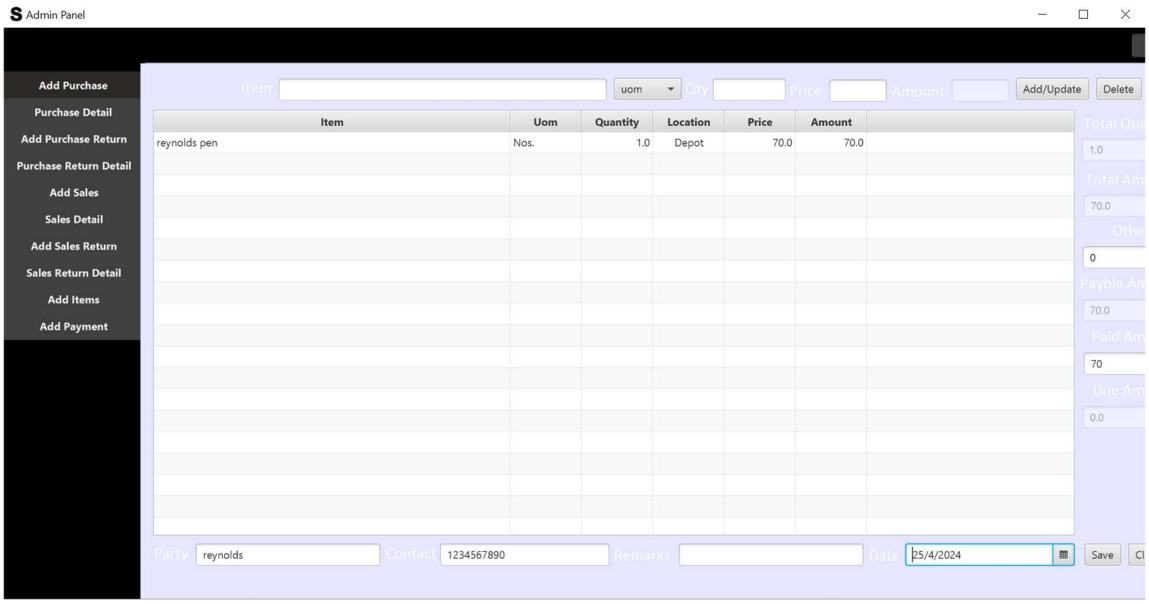
Enter a SQL expression to filter results (use Ctrl+Space)

	ITEM_ID	CODE	NAME	PACK_UNIT_ID	PACK_SIZE	STANDARD_UNIT_ID	CREATED	MODIFI
1	169	P007	reynolds pen	1	10	1	2024-04-25 00:00:00.000	2024-04-2
2	168	P006	dove shampo	1	1	1	2024-04-24 00:00:00.000	2024-04-2
3	166	P006	dove soap	1	1	1	2024-04-24 00:00:00.000	2024-04-2
4	164	P005	milton water l	1	1	1	2024-04-24 00:00:00.000	2024-04-2
5	162	P004	seba med lip	1	1	1	2024-04-24 00:00:00.000	2024-04-2

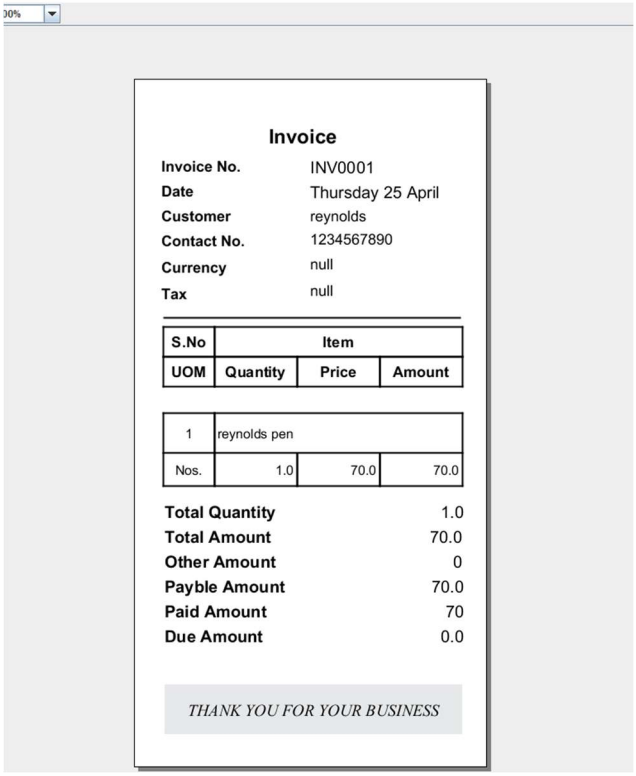
Add Purchase







Generate Invoice Bill for Purchase



Invoice

Invoice No.	INV0001
Date	Thursday 25 April
Customer	reynolds
Contact No.	1234567890
Currency	null
Tax	null

S.No	Item		
UOM	Quantity	Price	Amount
1	reynolds pen		
Nos.	1.0	70.0	70.0

Total Quantity	1.0
Total Amount	70.0
Other Amount	0
Payble Amount	70.0
Paid Amount	70
Due Amount	0.0

THANK YOU FOR YOUR BUSINESS

Purchase Reflected in the database

```
select b.* from pos.purchases a, pos.purchase_details b, pos.items c
where a.order_id = b.order_id and b.item_id = c.item_id
order by a.order_id desc
```

Results 1

select b.* from pos.purchases a, pos.pu | Enter a SQL expression to filter results (use Ctrl+Space)

	ORDER_ID	ITEM_ID	ITEM_NAME	LOCATION	UOM	QUANTITY	PRICE	AMOUNT
1	170	169	reynolds pen	[NULL]	Nos.	1	70	70
2	167	166	dove soap	[NULL]	Meter	1	40	40
3	165	164	milton water bottle	[NULL]	Meter	100	200	20,000
4	163	162	seba med lip balm	[NULL]	Meter	100	300	30,000
5	161	99	abc	[NULL]	Meter	100	1	100

Purchase Details

Admin Panel

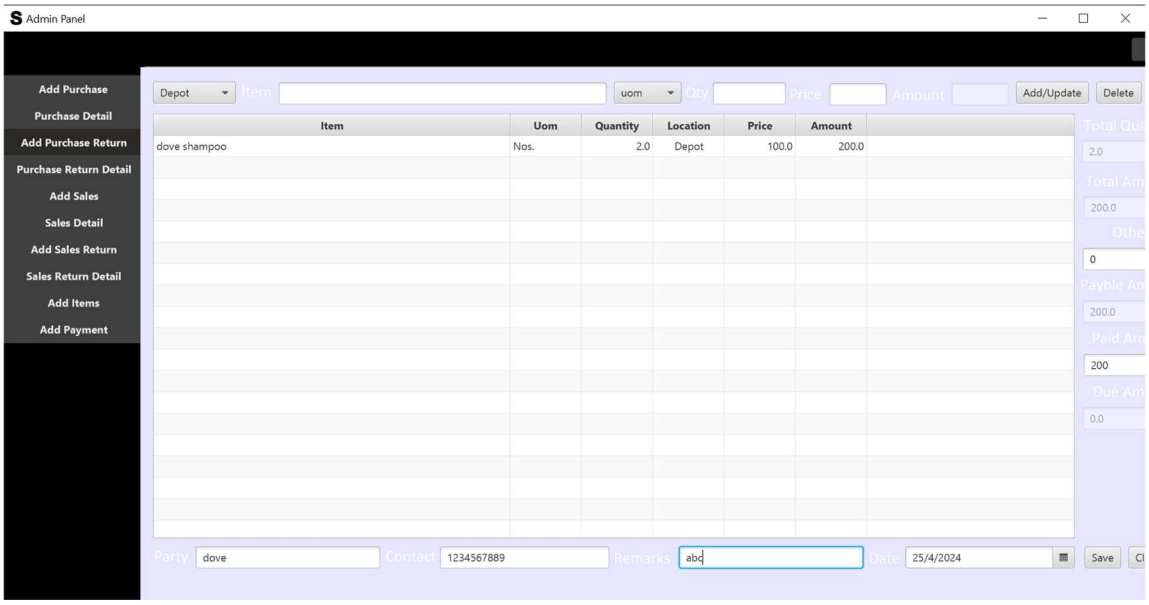
- Add Purchase
- Purchase Detail
- Add Purchase Return
- Purchase Return Detail
- Add Sales
- Sales Detail
- Add Sales Return
- Sales Return Detail
- Add Items
- Add Payment

Order Id	Invoice Date	Party	Quantity	Total Amount	Other Amount	Payble Amount	Paid Amount	Due Amount
170	2024-04-25	reynolds	1.0	70.0	0.0	70.0	70.0	0.0
167	2024-04-24		1.0	40.0	0.0	40.0	40.0	0.0
165	2024-04-24		100.0	20000.0	0.0	20000.0	20000.0	0.0
163	2024-04-24		100.0	30000.0	0.0	30000.0	30000.0	0.0
161	2024-04-23		100.0	100.0	0.0	100.0	100.0	0.0
142	2024-04-13		1.0	100.0	0.0	100.0	100.0	0.0
141	2024-04-13		1.0	100.0	0.0	100.0	100.0	0.0
122	2024-04-18		1.0	100.0	0.0	100.0	100.0	0.0
121	2024-04-11		1.0	100.0	0.0	100.0	100.0	0.0
102	2024-03-27		100.0	10000.0	0.0	10000.0	0.0	10000.0
100	2024-03-27		10.0	10.0	0.0	10.0	0.0	10.0
69	2024-03-06		10.0	500.0	0.0	500.0	0.0	500.0
68	2024-02-27		0.0	0.0	0.0	0.0	0.0	0.0
67	2024-03-05		0.0	0.0	0.0	0.0	0.0	0.0
66	2024-03-05		0.0	0.0	0.0	0.0	0.0	0.0
65	2024-03-15		0.0	0.0	0.0	0.0	0.0	0.0
64	2024-03-05	ABC Company	1.0	50.0	0.0	50.0	0.0	50.0
1	2024-03-20	XYZ Supplier	20.0	2000.0	100.0	2100.0	2000.0	100.0

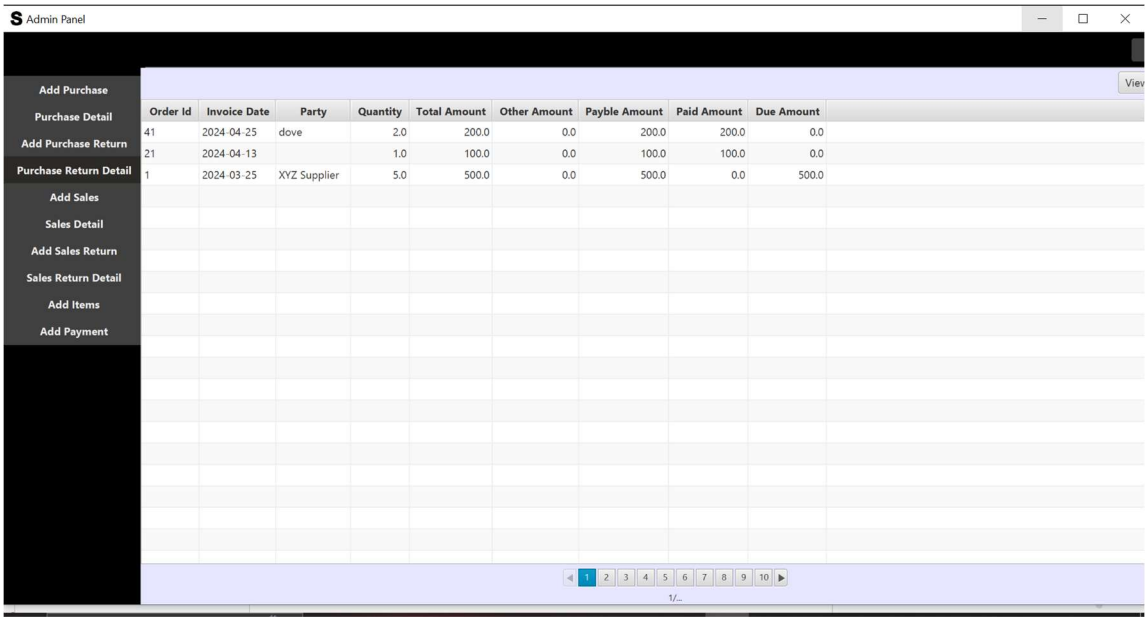
1 2 3 4 5 6 7 8 9 10

1/...

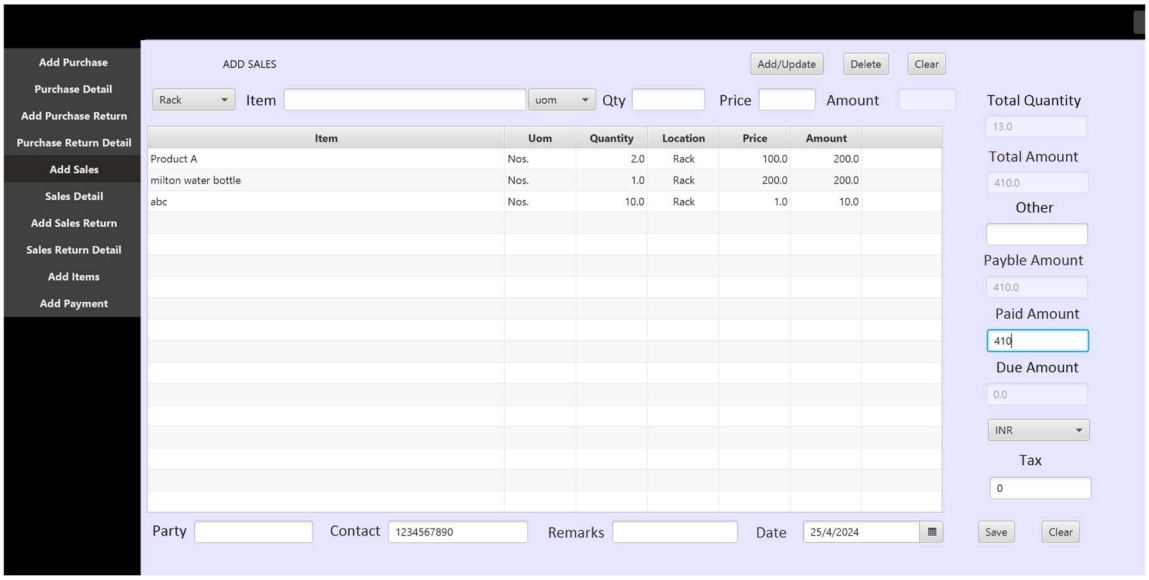
Add Purchase Return



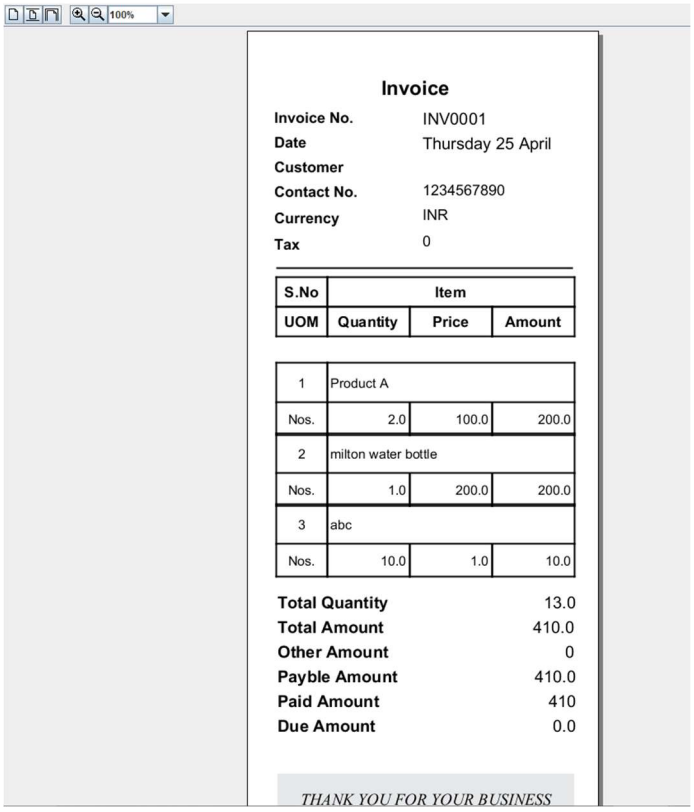
Purchase Return Details



Add Sales



Generate Invoice Bill for Added Sales



Invoice

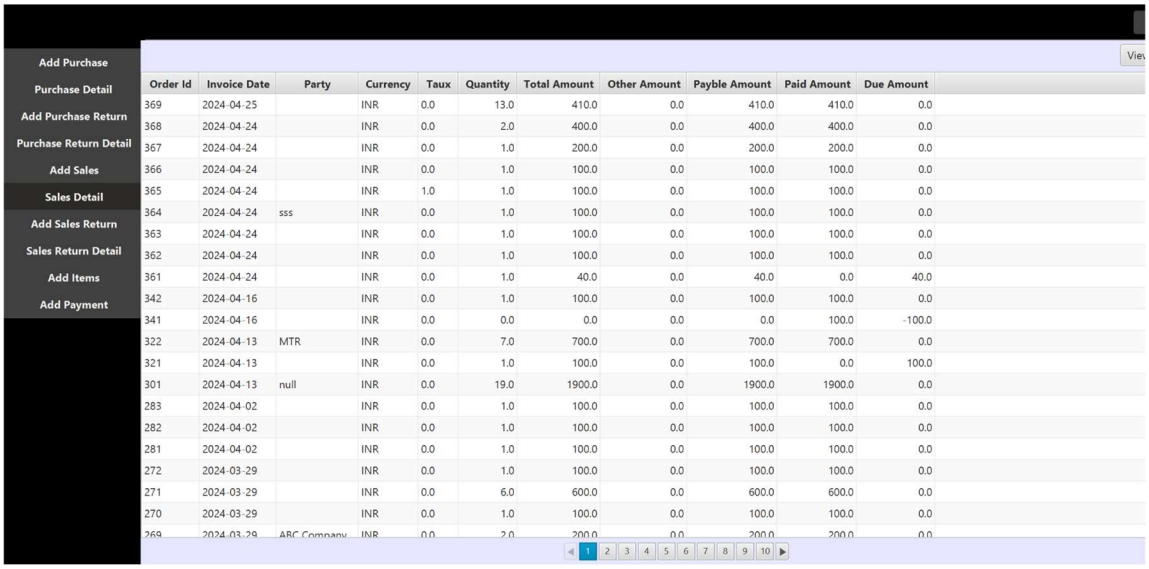
Invoice No.	INV0001
Date	Thursday 25 April
Customer	
Contact No.	1234567890
Currency	INR
Tax	0

S.No	Item		
UOM	Quantity	Price	Amount
1	Product A		
Nos.	2.0	100.0	200.0
2	milton water bottle		
Nos.	1.0	200.0	200.0
3	abc		
Nos.	10.0	1.0	10.0

Total Quantity	13.0
Total Amount	410.0
Other Amount	0
Payble Amount	410.0
Paid Amount	410
Due Amount	0.0

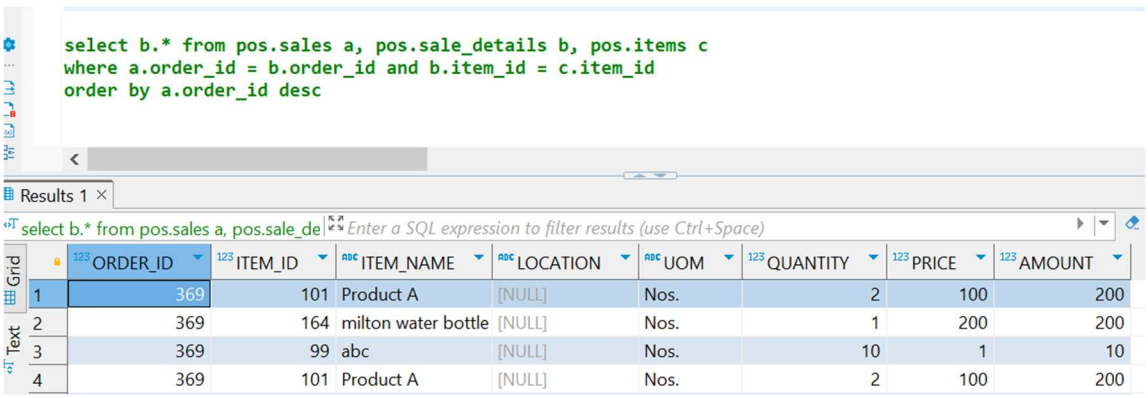
THANK YOU FOR YOUR BUSINESS

Sales Details



Order Id	Invoice Date	Party	Currency	Taux	Quantity	Total Amount	Other Amount	Payble Amount	Paid Amount	Due Amount
369	2024-04-25		INR	0.0	13.0	410.0	0.0	410.0	410.0	0.0
368	2024-04-24		INR	0.0	2.0	400.0	0.0	400.0	400.0	0.0
367	2024-04-24		INR	0.0	1.0	200.0	0.0	200.0	200.0	0.0
366	2024-04-24		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
365	2024-04-24		INR	1.0	1.0	100.0	0.0	100.0	100.0	0.0
364	2024-04-24	sss	INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
363	2024-04-24		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
362	2024-04-24		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
361	2024-04-24		INR	0.0	1.0	40.0	0.0	40.0	0.0	40.0
342	2024-04-16		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
341	2024-04-16		INR	0.0	0.0	0.0	0.0	0.0	100.0	-100.0
322	2024-04-13	MTR	INR	0.0	7.0	700.0	0.0	700.0	700.0	0.0
321	2024-04-13		INR	0.0	1.0	100.0	0.0	100.0	0.0	100.0
301	2024-04-13	null	INR	0.0	19.0	1900.0	0.0	1900.0	1900.0	0.0
283	2024-04-02		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
282	2024-04-02		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
281	2024-04-02		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
272	2024-03-29		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
271	2024-03-29		INR	0.0	6.0	600.0	0.0	600.0	600.0	0.0
270	2024-03-29		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
269	2024-03-29	ABC Company	INR	0.0	2.0	200.0	0.0	200.0	200.0	0.0

Add Sales Reflected in Database



```
select b.* from pos.sales a, pos.sale_details b, pos.items c
where a.order_id = b.order_id and b.item_id = c.item_id
order by a.order_id desc
```

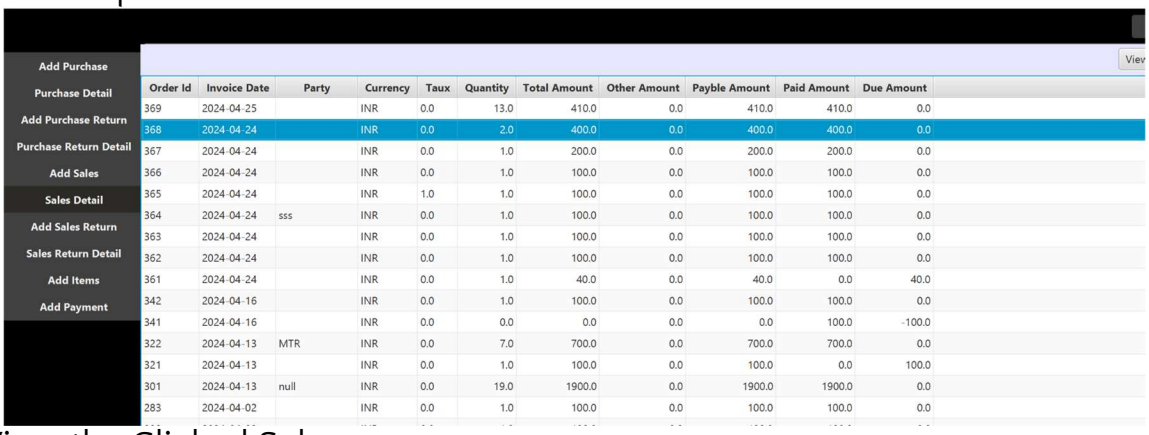
Results 1

select b.* from pos.sales a, pos.sale_de

Enter a SQL expression to filter results (use Ctrl+Space)

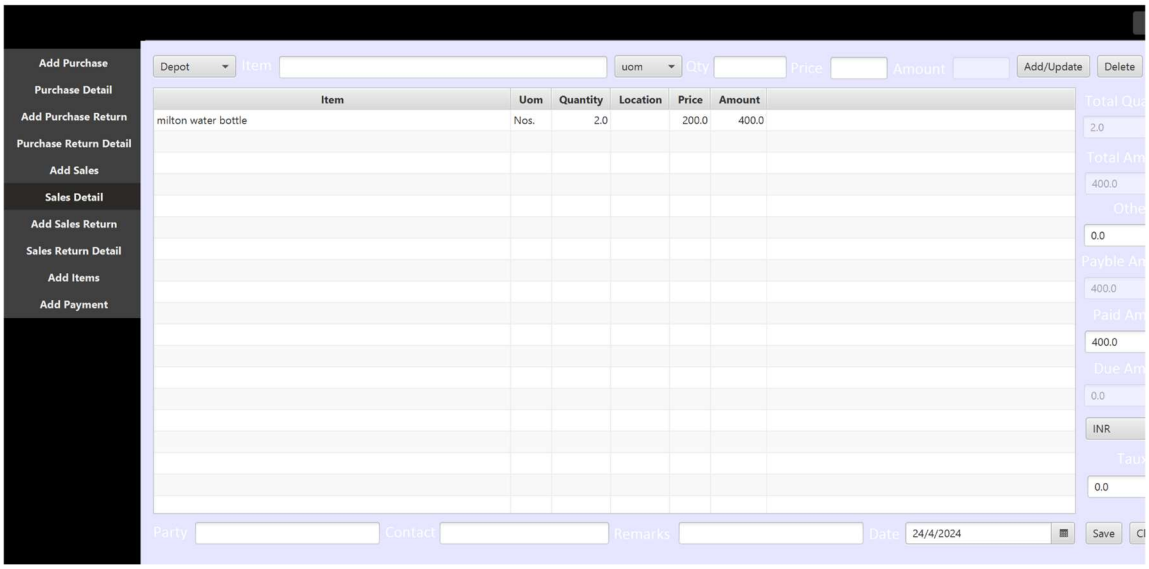
	ORDER_ID	ITEM_ID	ITEM_NAME	LOCATION	UOM	QUANTITY	PRICE	AMOUNT
1	369	101	Product A	[NULL]	Nos.	2	100	200
2	369	164	milton water bottle	[NULL]	Nos.	1	200	200
3	369	99	abc	[NULL]	Nos.	10	1	10
4	369	101	Product A	[NULL]	Nos.	2	100	200

Click on particular Sales

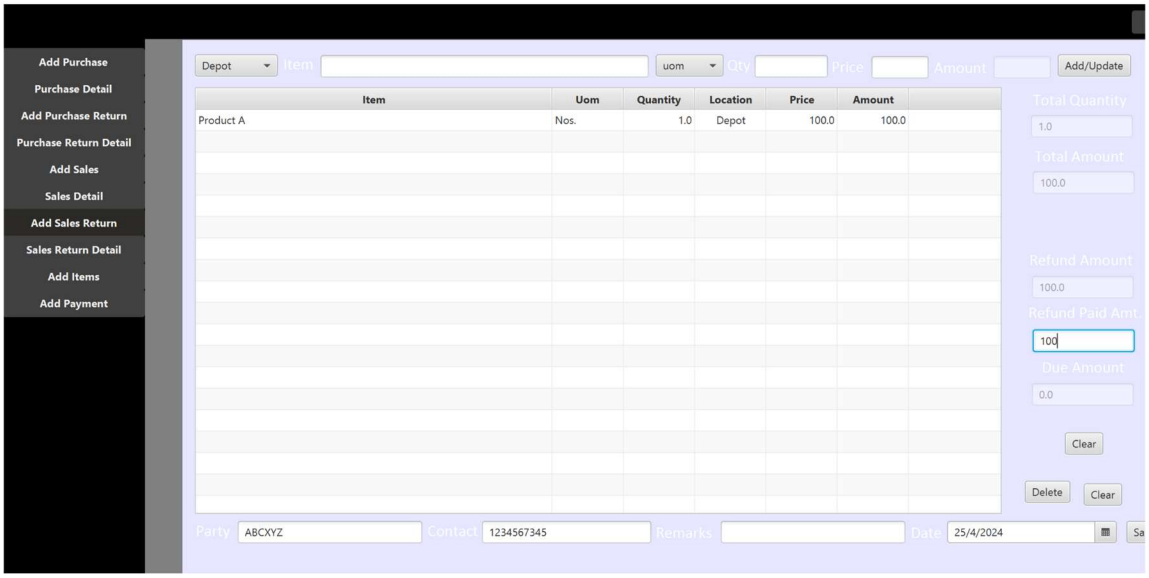


Order Id	Invoice Date	Party	Currency	Taux	Quantity	Total Amount	Other Amount	Payble Amount	Paid Amount	Due Amount
369	2024-04-25		INR	0.0	13.0	410.0	0.0	410.0	410.0	0.0
368	2024-04-24		INR	0.0	2.0	400.0	0.0	400.0	400.0	0.0
367	2024-04-24		INR	0.0	1.0	200.0	0.0	200.0	200.0	0.0
366	2024-04-24		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
365	2024-04-24		INR	1.0	1.0	100.0	0.0	100.0	100.0	0.0
364	2024-04-24	sss	INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
363	2024-04-24		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
362	2024-04-24		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
361	2024-04-24		INR	0.0	1.0	40.0	0.0	40.0	0.0	40.0
342	2024-04-16		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
341	2024-04-16		INR	0.0	0.0	0.0	0.0	0.0	100.0	-100.0
322	2024-04-13	MTR	INR	0.0	7.0	700.0	0.0	700.0	700.0	0.0
321	2024-04-13		INR	0.0	1.0	100.0	0.0	100.0	0.0	100.0
301	2024-04-13	null	INR	0.0	19.0	1900.0	0.0	1900.0	1900.0	0.0
283	2024-04-02		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0

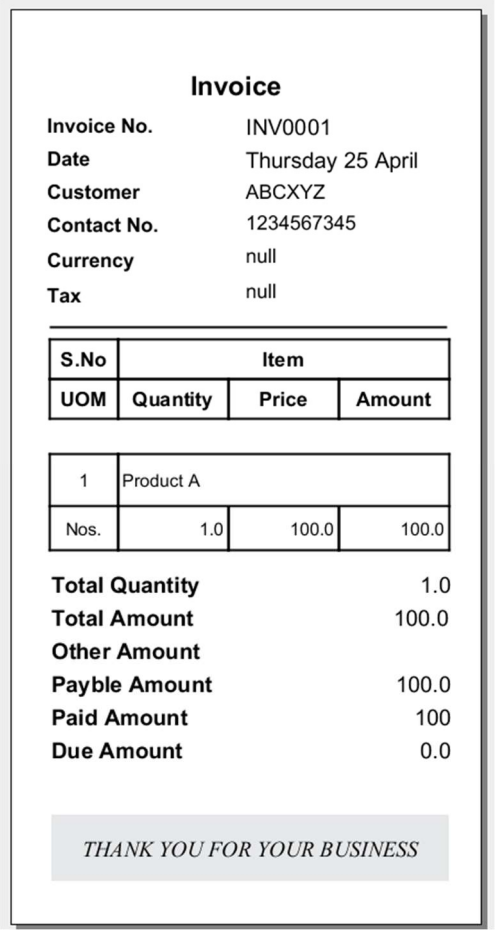
View the Clicked Sales



Refund Purchase



Generate Invoice Bill for the Returned Purchase



Invoice

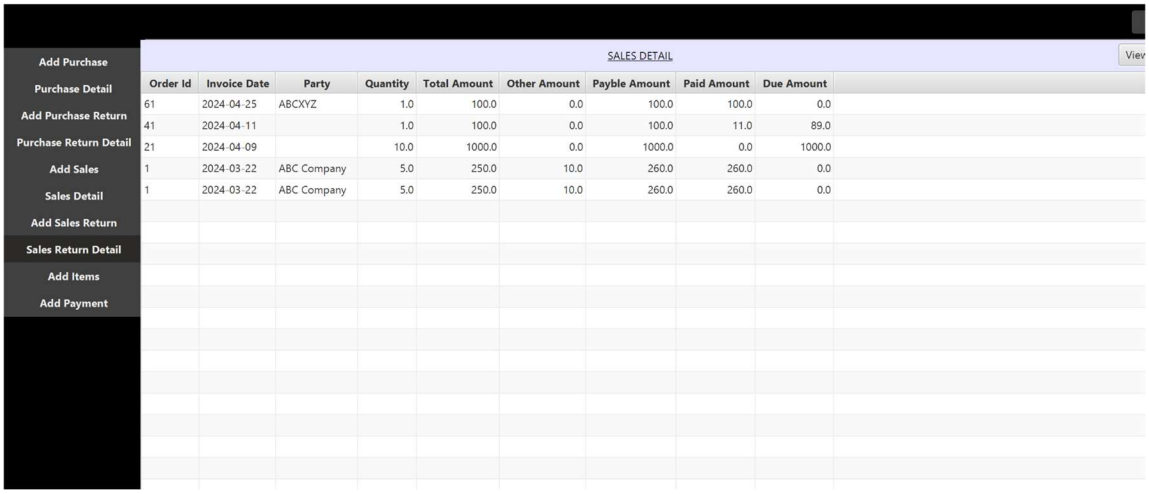
Invoice No.	INV0001
Date	Thursday 25 April
Customer	ABCXYZ
Contact No.	1234567345
Currency	null
Tax	null

S.No	Item		
UOM	**Quantity**	**Price**	**Amount**
1	Product A		
Nos.	1.0	100.0	100.0

Total Quantity	1.0
Total Amount	100.0
Other Amount	
Payble Amount	100.0
Paid Amount	100
Due Amount	0.0

THANK YOU FOR YOUR BUSINESS

Sales Return Details

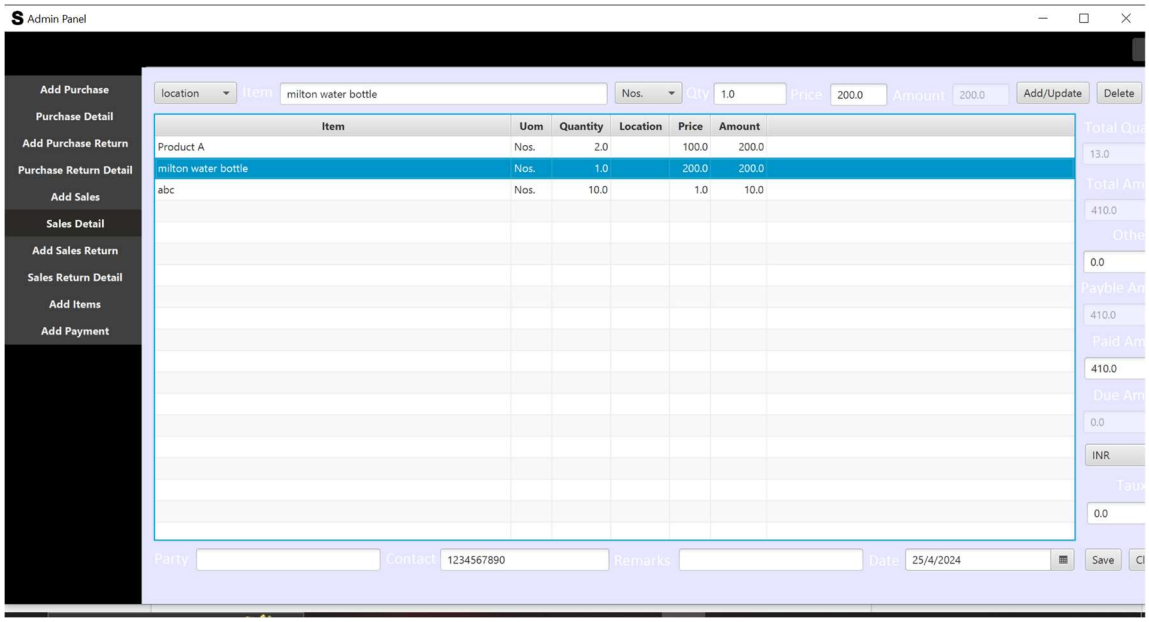


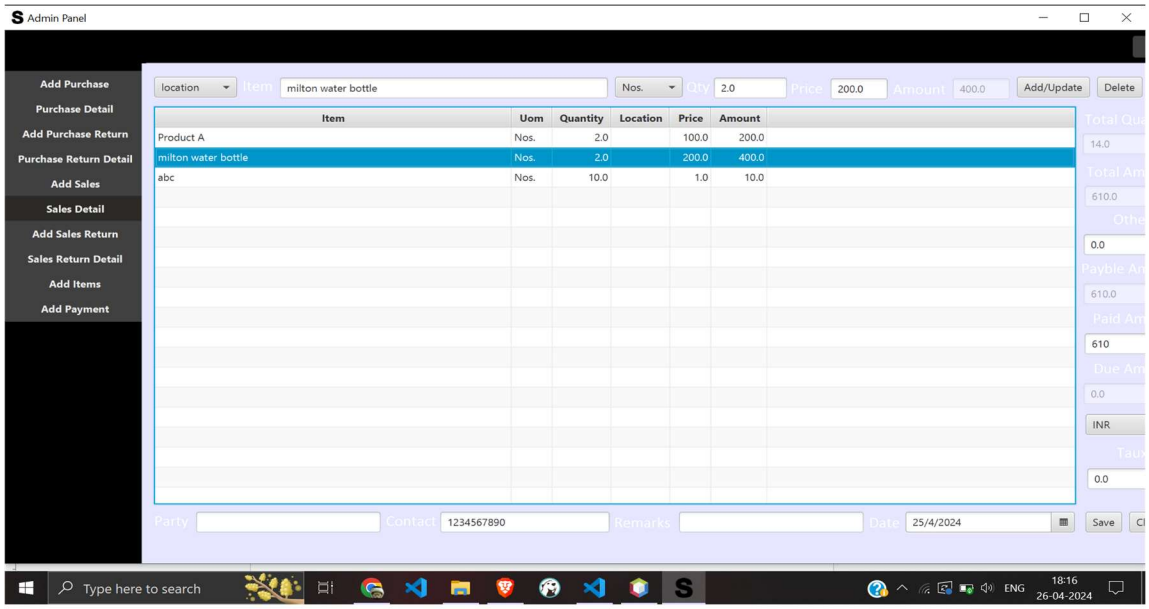
Add Purchase
Purchase Detail
Add Purchase Return
Purchase Return Detail
Add Sales
Sales Detail
Add Sales Return
Sales Return Detail
Add Items
Add Payment

SALES DETAIL

Order Id	Invoice Date	Party	Quantity	Total Amount	Other Amount	Payble Amount	Paid Amount	Due Amount
61	2024-04-25	ABCXYZ	1.0	100.0	0.0	100.0	100.0	0.0
41	2024-04-11		1.0	100.0	0.0	100.0	11.0	89.0
21	2024-04-09		10.0	1000.0	0.0	1000.0	0.0	1000.0
1	2024-03-22	ABC Company	5.0	250.0	10.0	260.0	260.0	0.0
1	2024-03-22	ABC Company	5.0	250.0	10.0	260.0	260.0	0.0

Updating the values of a sale





Value updated in Sales Details

Add Purchase
Purchase Detail
Add Purchase Return
Purchase Return Detail
Add Sales
Sales Detail

Order Id	Invoice Date	Party	Currency	Taux	Quantity	Total Amount	Other Amount	Payble Amount	Paid Amount	Due Amount
369	2024-04-25	null	INR	0.0	14.0	610.0	0.0	610.0	610.0	0.0
368	2024-04-24		INR	0.0	2.0	400.0	0.0	400.0	400.0	0.0
367	2024-04-24		INR	0.0	1.0	200.0	0.0	200.0	200.0	0.0
366	2024-04-24		INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0
365	2024-04-24		INR	1.0	1.0	100.0	0.0	100.0	100.0	0.0
364	2024-04-24	sss	INR	0.0	1.0	100.0	0.0	100.0	100.0	0.0

Purchase Return reflected in Database

```
select b.* from pos.sale_returns a, pos.sale_return_details b, pos.items c
where a.order_id = b.order_id and b.item_id = c.item_id
order by a.order_id desc
```

Results 1

select b.* from pos.sale_returns a, pos.: Enter a SQL expression to filter results (use Ctrl+Space)

	ORDER_ID	ITEM_ID	ITEM_NAME	LOCATION	UOM	QUANTITY	PRICE	AMOUNT
1	61	101	Product A	[NULL]	Nos.	1	100	100
2	61	101	Product A	[NULL]	Nos.	1	100	100
3	41	101	Product A	[NULL]	Meter	1	100	100