Kurdistan Region-Iraq

University of Salahaddin-Erbil

College of Engineering

Software and Informatics Departments

Inventory Management System

Prepared By

Aya Mumin Khallat Fatah Yousra Zahir

2024-2025

1. Introduction

Inventory Management System is a **JavaFX desktop application** integrated with a **MySQL** database to efficiently manage the inventory operations of a business. It offers functionality to add, edit, and monitor products, suppliers, users, and purchase records. This system is tailored for small to medium-sized businesses that need a simple and efficient way to keep track of stock.

---

2. Objective

- To provide an easy-to-use tool for managing inventory, purchases, suppliers, and users.
- To automate stock tracking and updates based on purchases.
- To maintain an organized database that improves operational efficiency.

---

3. Problem Statement

Managing inventory manually can be time-consuming, error-prone, and inefficient. Businesses often struggle with stock discrepancies, forgotten supplier details, and uncontrolled user access. This project addresses these challenges by offering a centralized and digital inventory management system.

---

4. Technology Stack

Component	Technology Used
Frontend	JavaFX
Backend	Java (JDBC for database communication)
Database	MySQL
IDEs	NetBeans / IntelliJ IDEA
Version Control	GitHub

---

5. Functional Requirements

- **User Authentication:** Login system with role-based access (e.g., Admin, User).
- **User Management:** Create, update, delete users; assign roles.
- **Supplier Management:** Add, edit, delete supplier records.
- **Product Management:** Add products with details like quantity, price, supplier info.
- **Purchase Management:** Record and manage purchases, automatically update product stock.
- **Reporting:** View basic reports on inventory and purchases.

---

6. Non-Functional Requirements

- **Performance:** The system must handle up to hundreds of products and users efficiently.
- **Security:** Password-protected login to prevent unauthorized access.
- **Usability:** Intuitive and user-friendly JavaFX interface.
- **Data Integrity:** Consistent and reliable data storage using MySQL transactions.
- **Portability:** System can be installed and used on any Windows/Linux machine with Java installed.

---

7. Modules Description

7.1 User Management

- Add new users
- Edit user information
- Delete users
- Assign roles (Admin, User)

7.2 Supplier Management

- Store supplier details (name, contact, address)
- Edit or delete supplier records

7.3 Product Management

- Add products with name, quantity, price, supplier reference
- Edit product details
- Delete products
- Monitor stock levels

7.4 Purchase Management

- Record new purchases
- Select supplier and products in the purchase form
- Automatically update product quantities based on purchases

7.5 Authentication

- Secure login page
- Password authentication
- Redirect users based on their role

---

8. Scope

8.1 In Scope

- Desktop-based inventory management system.
- Local database storage.
- Basic reporting for stock and purchases.
- CRUD operations for users, suppliers, products, and purchases.

8.2 Out of Scope

- Cloud-based or web-based access.
- Customer sales and invoicing system.
- Multi-warehouse stock management.
- Advanced analytics and prediction features.

---

9. System Architecture

- **Presentation Layer:** JavaFX UI
- **Application Logic Layer:** Java Classes for business logic
- **Database Layer:** MySQL database accessed through JDBC

---

10. Database Design Overview

Main Tables:

- `users` — user information and roles
- `suppliers` — supplier details
- `products` — product inventory details
- `purchases` — records of purchases made

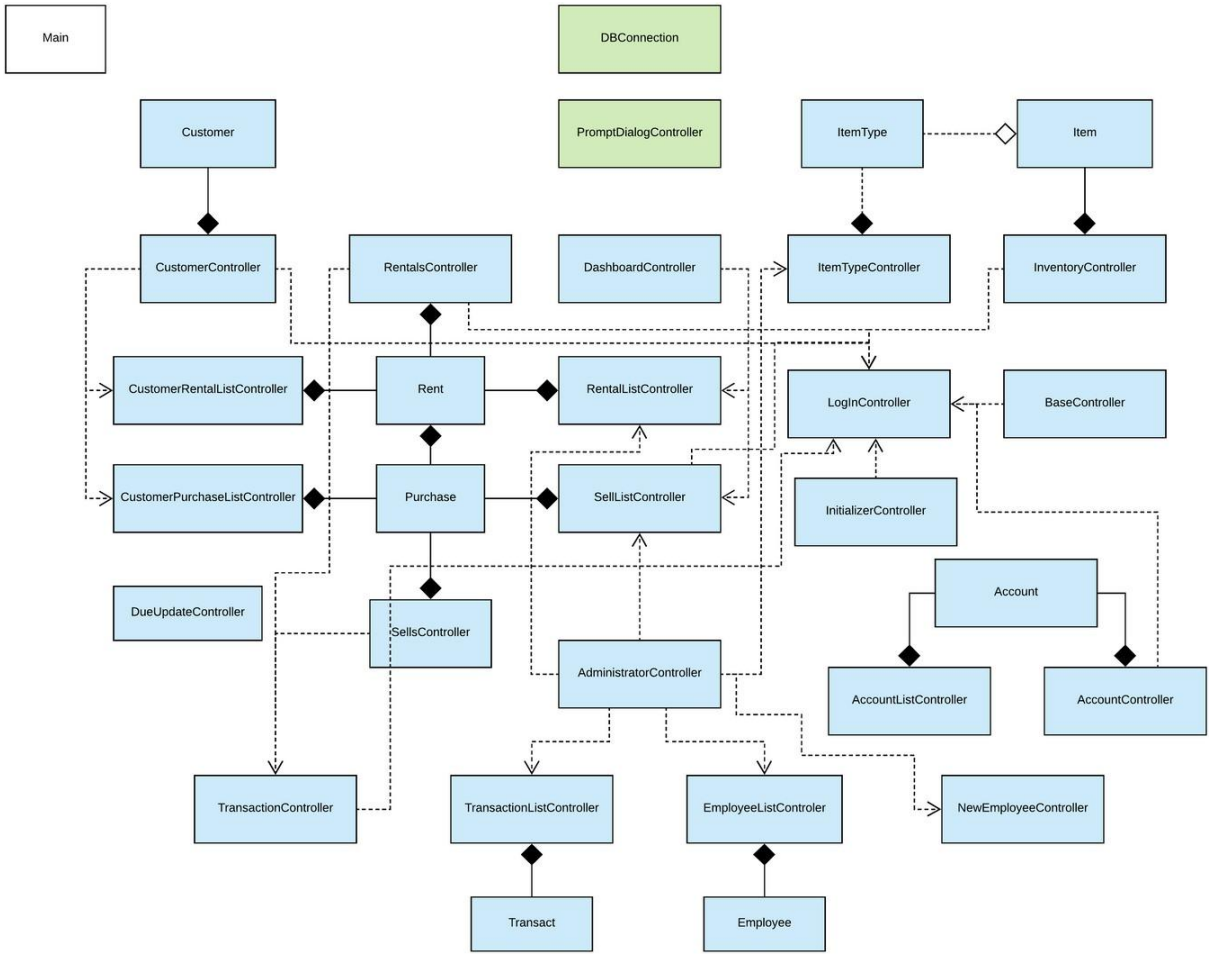
---

11. User Roles

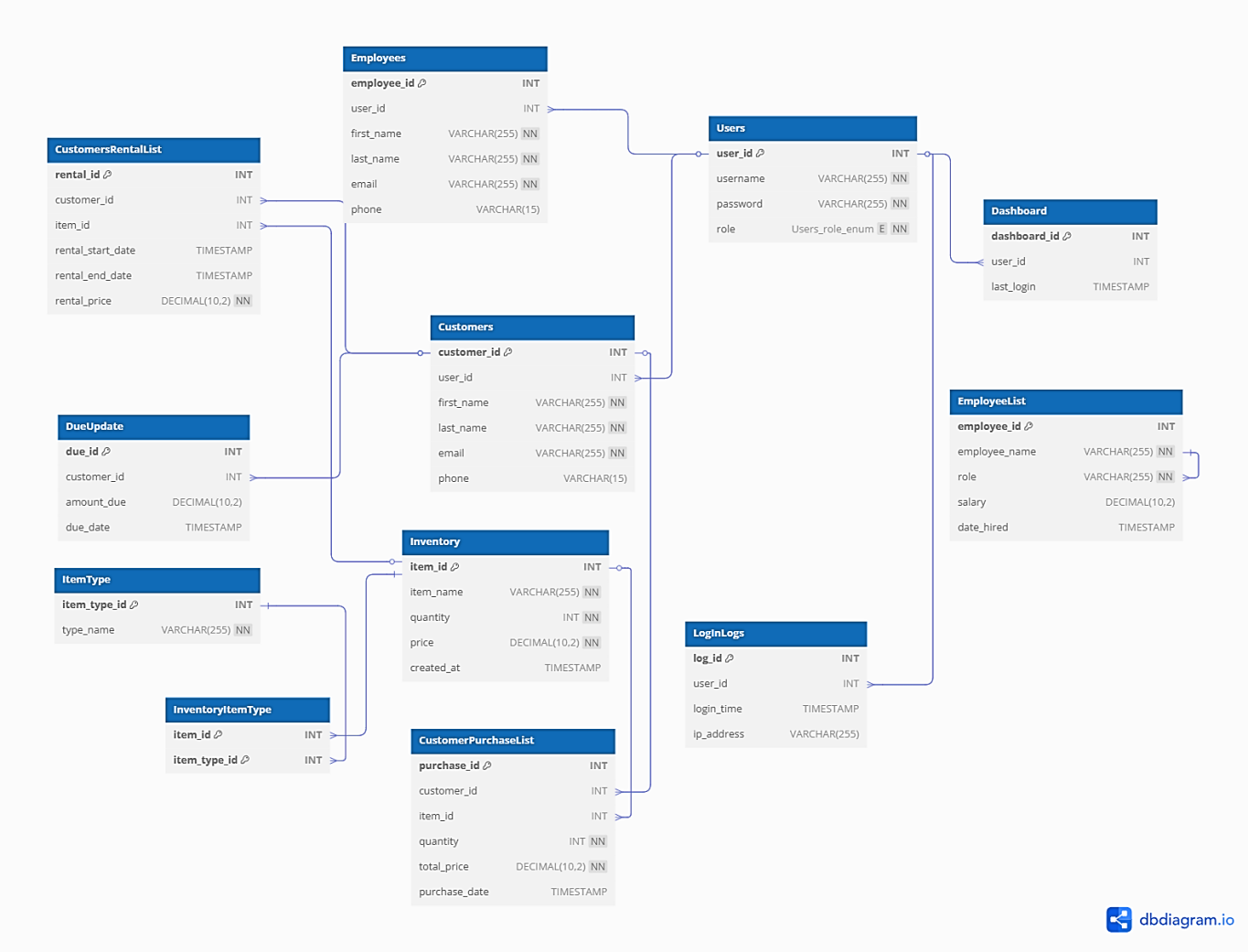
Role	Permissions
Admin	Full access (users, suppliers, products, purchases)
User	Limited access (view and manage products and purchases only)

---

1. UML Class Diagram



1. Entity Relationship Diagram



Relationships Explanation

1. Users and Customers / Employees

- **Relationship**:
  - The `Users` table holds common user data (such as login credentials and roles). Each user can be either a `Customer` or an `Employee`.
  - The `Customers` and `Employees` tables reference the `Users` table through the `user_id` foreign key.
- **Explanation**:
  - A user can only have one role (either `Customer` or `Employee`), but a user can be one of these two roles. Hence, there is a **one-to-one** relationship between the `Users` and `Customers`/`Employees` tables. Each `Customer` or `Employee` is linked to exactly one `User`.

Correctness: The relationship is correct. Each user has a unique `user_id` which ties to either a `Customer` or an `Employee`.

2. Customers and Inventory (CustomerPurchaseList and CustomersRentalList)

- **Relationship**:
  - A `Customer` can make purchases and rentals.
  - The `CustomerPurchaseList` table records the items purchased by a customer, and the `CustomersRentalList` table records the items rented by a customer.
  - Both tables contain foreign keys to the `Customers` (`customer_id`) and `Inventory` (`item_id`) tables.
- **Explanation**:
  - **Customer to Purchase**: A customer can buy many items (many-to-many relationship between `Customers` and `Inventory` through `CustomerPurchaseList`). The `CustomerPurchaseList` table bridges this many-to-many relationship by holding multiple purchases for each customer.
  - **Customer to Rental**: Similarly, a customer can rent many items, so the `CustomersRentalList` table holds multiple rental entries for each customer, also representing a many-to-many relationship.
- **Correctness**: The relationships are correct. These are many-to-many relationships between `Customers` and `Inventory`, facilitated by the `CustomerPurchaseList` and `CustomersRentalList` tables.

3. Inventory and ItemType (Many-to-Many Relationship)

- **Relationship**:
  - An item in the `Inventory` can belong to multiple `ItemType`s (for example, a product could be both a "laptop" and an "electronics" item).
  - The `InventoryItemType` table is used to create a many-to-many relationship between the `Inventory` and `ItemType` tables.
- **Explanation**:
  - Each `Inventory` item can be categorized under multiple `ItemType` categories. For instance, a laptop can belong to both the "Electronics" and "Computers" types.
  - The `InventoryItemType` table serves as the bridge that connects `Inventory` and `ItemType`, linking each item to one or more types.
- **Correctness**: The relationship is correct. It's a many-to-many relationship between `Inventory` and `ItemType`.

4. Inventory Table

- **Relationship**:
  - The `Inventory` table has an `item_id` as the primary key. Each `Inventory` item can be part of a purchase or rental (as shown in the `CustomerPurchaseList` and `CustomersRentalList`).
- **Explanation**:
  - Each item in the `Inventory` table can be associated with many customers' purchases and rentals, forming a many-to-many relationship between `Inventory` and `Customers` through the aforementioned bridge tables (`CustomerPurchaseList` and `CustomersRentalList`).
- **Correctness**: The relationship is correct. The `Inventory` table functions as the central table that other entities (like `Customers`) reference through purchase and rental tables.

5. DueUpdate (Customer Due Payments)

- **Relationship**:
  - The `DueUpdate` table is related to the `Customers` table through the `customer_id` foreign key. It tracks any outstanding amounts that a customer needs to pay.
- **Explanation**:
  - This is a **one-to-many** relationship where each customer can have multiple due updates (e.g., outstanding payments). This allows the system to track any dues a customer has across different transactions.
- **Correctness**: The relationship is correct. Each customer can have multiple dues or outstanding payments.

6. Employees Table

- **Relationship**:
  - The `Employees` table is directly related to the `Users` table via the `user_id` foreign key. This relationship allows employees to log into the system using the same user credentials.
- **Explanation**:
  - This is a **one-to-one** relationship, as each employee has one entry in the `Users` table. The `Employees` table holds additional information about the employee (name, contact details, etc.), which is separate from their login information.
- **Correctness**: The relationship is correct. Each employee is linked to a unique user.

7. LogInLogs (Tracking User Logins)

- **Relationship**:
  - The `LogInLogs` table records each time a user logs in, and it's linked to the `Users` table by the `user_id`.
- **Explanation**:
  - This is a **one-to-many** relationship. A user can have multiple login attempts (or logs), but each log belongs to exactly one user.
- **Correctness**: The relationship is correct. It captures login events for each user.

Table of Contents

1. AccountController

Overview

Manages the **Accounts** section of a JavaFX application, interacting with a database to display, search, and add customer account data.

Key Features

- **UI Initialization**
  - Sets up autocomplete for customer and account search fields.
  - Binds database data to a TableView.
  - Fetches the next available account ID.
- **Auto-Fill ID**
  - Queries the database to find the next available account ID.
  - Clears fields for account creation.
- **Search Feature**
  - Enables searching accounts by name using SQL `LIKE` queries.
  - Updates TableView dynamically.
  - Toggles between search results and the full list.
- **Account Management**
  - Reloads all accounts created by the currently logged-in user.
  - Validates new account fields.
  - Inserts new accounts into the database.
  - Displays success or error messages via snackbar dialogs.

Main Components

- `TableView<Account>`: Displays recent accounts.
- `JFXTextField`: Inputs for Account ID, Customer ID, Payment Method, Search.
- `ObservableList<Account>`: Holds the list of displayed accounts.
- `DBConnection`: Custom database connection class.
- `Account`: Model class for account data.
- `PromptDialogController`: Handles user alerts and dialogs.

Used Libraries

- **JavaFX**: Core UI components.
- **JFoenix**: Material Design controls (e.g., `JFXTextField`, `JFXButton`).
- **ControlsFX**: Advanced UI features like autocompletion.
- **FontAwesomeFX**: FontAwesome icons in JavaFX.

2. AccountListController

Overview

Manages the **Account List** interface, supporting view, update, and delete operations on accounts.

Key Features

- **TableView Setup**
  - Binds columns to the `Account` model properties.
  - Fetches and displays accounts from the database.
- **Interactive Table**
  - Double-click to load selected account details into form fields.
  - Displays structured account information.
- **CRUD Operations**
  - Deletes accounts by `accID` using SQL.
  - Updates existing accounts with new information.
  - Displays success or error dialogs based on operation outcomes.

UI Components

- `JFXTextField`: Inputs for account information.
- `TableView<Account>`: Displays account listings.
- `JFXButton`: Actions like Add, Delete, Update.
- `FontAwesomeIconView`: Enhances UI with icons.

Database Interactions

- CRUD operations via prepared SQL statements.
- Immediate UI update after database changes.

Used Classes

- `Account`, `DBConnection`, `PromptDialogController`.

3. AdministratorController

Overview

Controls the **Administrative Dashboard**, managing employee records, accounts, transactions, and database operations.

Key Features

- **Window Navigation**
  - Loads and displays FXML windows.
  - Manages switching between views.
- **Account Management**
  - Opens account management interfaces.
  - Handles account updates and deletions.
- **Employee Management**
  - Opens employee addition forms.
  - Manages employee lists.
- **Transaction Viewing**
  - Displays transaction records.
  - Filters rental and sales data.
- **Database Management**
  - Provides database reset functionality with confirmation dialogs.

UI Components

- `JFXButton`: Triggers admin actions.
- FXML Windows: Separate views for different management modules.

Key Methods

- `loadWindow()`, `accUpdate()`, `itemTypeManage()`, `addNewEmployee()`, `empList()`, `showTransactions()`, `totalRents()`, `totalSell()`, `deleteAll()`.

Used Utility Classes

- `DBConnection`, `PromptDialogController`, `SellListController`, JavaFX components (`FXMLLoader`, `Stage`, `Scene`, `Alert`).

4. BaseController

Overview

Manages the **Main Dashboard Layout** and **Navigation**, serving as the central hub of the application.

Key Features

- **UI Initialization**
  - Maps navigation buttons to FXML views.
  - Displays logged-in user details.
  - Controls admin feature access.
  - Starts a live clock thread.
  - Loads the default dashboard view.
- **Dynamic View Loading**
  - Loads FXML files into the right-hand panel dynamically.
- **Navigation Handling**
  - Responds to navigation clicks.
  - Highlights the active button.
- **User Session Management**
  - Displays user session info.
  - Supports logout and returns to the login screen.

Main Components

- `AnchorPane`: `paneRight`, `paneLeft`.
- `JFXButton`: Navigation buttons.
- `Label`: Displays username, access level, real-time clock.
- `HashMap<String, String>`: Maps button labels to FXML paths.

Key Methods

- `initialize()`, `ctrlRightPane()`, `btnNavigators()`, `loadFXMLMap()`, `borderSelector()`, `runClock()`, `logOut()`.

Used Libraries

- JavaFX, JFoenix, ControlsFX, Platform threading utilities.

5. CustomerController

Overview

Controls the **Customer Management Interface**, supporting full CRUD operations, search functionality, and customer history viewing.

Key Features

- **Customer Record Navigation**
  - Browse through customer records.
  - View details like personal info and financial summary.
- **CRUD Operations**
  - Create, read, update, delete customer records with validations.
- **Search Functionality**
  - Search by customer ID or name.
  - Reset search results.
- **UI Toggle Modes**
  - Navigation Mode: Read-only viewing.
  - Edit Mode: Field editing enabled.
  - Entry Mode: Adding new customers.
- **Customer Transaction Access**
  - View customer purchases and rental histories.
- **Tabular Customer List**
  - View customers in a table (ID, name, contact, gender).

UI Components

- **Main Panes**:
  - Customer detail view.
  - Table view of all customers.
- **Input Fields**:
  - First name, last name, address, phone, email, gender.
- **Display Elements**:
  - Customer ID, amount due, creation date, profile image, record indicator, mode indicator.

Buttons

- Toggle edit mode, navigate records, search/reset search, add new customer, save, delete, show/hide table, view purchases, view rentals.

Database Interaction

- Full database CRUD support with constraint checking.
- Real-time due balance calculation.

Key Methods

- **Initialization**: Sets up UI and event handlers.
- **Record Navigation**: Updates the UI with current customer details.
- **Data Operations**: Calculate dues, refresh records, create/update/delete customer data.
- **Search Operations**: By ID or name.
- **UI Event Handlers**: Manage modes, delete, search, save operations.
- **Access Control**: Delete restricted to admin-level users.

Static Fields

- Current record position, total records, primary customer list, temporary search list, autocomplete name list, flags for search/add modes, and image path.

6. DueUpdateController

Overview

DueUpdateController is responsible for managing the interface that updates outstanding payments (dues) for both purchases and rentals within a retail/rental management system. It allows users to search transactions by ID, view payment details, and record new payments against pending balances.

Key Features

1. **Dual Transaction Support**
    - Manages payment updates for purchases and rentals separately.
    - Provides distinct UI sections for each transaction type.
    - Processes each using its corresponding database table.
2. **Payment Processing**
    - Calculates updated payment totals and remaining balances.
    - Updates database records with the latest payment information.
    - Displays confirmation messages upon successful updates.
3. **Transaction Lookup**
    - Enables searching transactions by ID.
    - Retrieves and displays full transaction details.
    - Auto-populates form fields with existing data.
4. **Form Management**
    - Clears form fields after successful updates.
    - Utilizes date pickers to display transaction dates.
    - Formats monetary values for clear display.

Main Components

- **AnchorPane**: Main container for UI elements.
- **JFXTextField**: Input fields for transaction and payment details.
- **JFXDatePicker**: Date selectors for transactions.
- **JFXButton**: Action buttons for searching and processing payments.
- **FontAwesomeIconView**: Icons providing visual cues on buttons.
- **Label**: Displays titles and field descriptions.

UI Components

Purchase Payment Section

- `txtSellId`: Purchase transaction ID input.
- `txtCustomerId`: Customer ID display.
- `txtItemId`: Item ID display.
- `txtSellDate`: Purchase date picker.
- `txtPaid`: Amount already paid display.
- `txtDue`: Outstanding amount display.
- `txtNewPay`: New payment amount input.
- `btnSearch`: Button to search for a purchase transaction.
- `btnProcced`: Button to process payment for a purchase.

Rental Payment Section

- `txtRentalID`: Rental transaction ID input.
- `txtCustomerId1`: Customer ID display for rentals.
- `txtItemRental`: Rented item ID display.
- `txtRentalDate`: Rental date picker.
- `txtPaidRental`: Amount already paid for rental display.
- `txtRentalDue`: Outstanding rental amount display.
- `txtNewPayRental`: New rental payment amount input.
- `btnSearchRental`: Button to search for a rental transaction.
- `btnProceedRental`: Button to process rental payment.

Database Interactions

The controller interacts with the SQL database to:

- Retrieve purchase and rental transaction details by ID.
- Update payment and outstanding amounts for purchases and rentals.

Key Methods

Initialization

- `initialize(URL, ResourceBundle)`: Sets up the initial UI state.

Purchase Payment Methods

- `btnSearchAction(ActionEvent)`: Searches for purchase transaction by ID and fills form fields.
- `updateSellDue(ActionEvent)`: Processes additional payment, updates database, clears form fields, and shows a success message.

Rental Payment Methods

- `btnSearchActionRental(ActionEvent)`: Searches for rental transaction by ID and fills form fields.
- `updateRentalDue(ActionEvent)`: Processes additional rental payment, updates database, clears form fields, and shows a success message.

SQL Queries Used

- **Find purchase by ID:**

```
sql
CopyEdit
SELECT * FROM purchases WHERE purchaseID = [id];
```

- **Find rental by ID:**

```
sql
CopyEdit
SELECT * FROM rentals WHERE rentalID = [id];
```

- **Update purchase payment:**

```
sql
CopyEdit
UPDATE purchases SET payAmount = [totalPaid], amountDue = [newDue]
WHERE purchaseID = [id];
```

- **Update rental payment:**

```
sql
CopyEdit
UPDATE rentals SET paid = [totalPaid], amountDue = [newDue] WHERE
rentalID = [id];
```

Used Classes

- **DBConnection**: Utility class for database connections.
- **PromptDialogController**: Displays operation results.
- **JFXTextField, JFXButton, JFXDatePicker**: JFoenix UI components.
- **FontAwesomeIconView**: Icons for buttons.
- **AnchorPane**: JavaFX layout container.

Dependencies

- **JavaFX**: For UI components and layouts.
- **JFoenix**: For material-design enhanced UI components.
- **FontAwesomeFX**: For icon display.
- **Java SQL**: For database interactions.
- **Custom PromptDialogController**: For showing user prompts.

7. DashboardController

Overview

DashboardController manages the main dashboard interface of the retail/rental management system. It provides real-time business monitoring by displaying financial summaries, inventory status, and access to detailed transaction reports.

Key Features

1. **Financial Summary Display**
    - Shows today's sales and rental totals.
    - Displays counts and monetary values for transactions.
    - Tracks outstanding payments.
    - Monitors the overall financial health.
2. **Inventory Monitoring**
    - Displays the number of out-of-stock items.
    - Provides an overview of inventory status.
3. **Dynamic Data Refresh**
    - Manual refresh option for real-time updates.
    - Pulls the latest metrics from the database.
4. **Transaction Report Access**
    - Opens detailed reports for today's sales and rentals.

Main Components

- **Labels**: Display counts, totals, and outstanding amounts.
- **JFXButtons**: Allow refreshing data and opening reports.
- **Static Variables**: Store important business metrics.

UI Components

Display Labels

- `lblTodaySellCtr`: Today's sales transaction count.
- `lblTodaysSellAmount`: Total sales amount for today.
- `lblTodaysRentalCtr`: Today's rental transaction count.
- `lblTodaysRentalAmount`: Total rental amount for today.
- `lblOutOfStock`: Number of out-of-stock items.
- `lblTotalDueAmount`: Total outstanding amount.
- `lblTodaysDueAmount`: Today's outstanding payments.

Action Buttons

- `btnTodaySell`: Opens today's sales report.
- `btnTodayRental`: Opens today's rental report.
- `loadAgain`: Refreshes dashboard data.

Database Interactions

The controller retrieves:

- Sales and rental transactions (count and totals).
- Outstanding payments (due amounts).
- Out-of-stock item count.

Key Methods

Initialization

- `initialize(URL, ResourceBundle)`: Sets up UI components and loads initial data.
- `setFields()`: Updates UI labels based on current static variables.

Data Operations

- `loadAgain(ActionEvent)`: Refreshes metrics by reloading the latest data from the database.

Navigation

- `showRent(ActionEvent)`: Opens today's rental transaction list.
- `showSell(ActionEvent)`: Opens today's sales transaction list.

SQL Queries Used

- **Outstanding rental payments:**

```
sql
CopyEdit
SELECT COUNT(*), SUM(amountDue) FROM rentals WHERE amountDue <> 0;
```

- **Outstanding purchase payments:**

```
sql
CopyEdit
SELECT COUNT(*), SUM(amountDue) FROM purchases WHERE amountDue <> 0;
```

- **Today's sales:**

```
sql
CopyEdit
SELECT COUNT(*), SUM(payAmount) FROM purchases WHERE purchaseDate =
'[today's date]';
```

- **Today's rentals:**

```
sql
CopyEdit
SELECT COUNT(*), SUM(paid) FROM rentals WHERE rentalDate = '[today's
date]';
```

- **Out-of-stock items:**

```
sql
CopyEdit
SELECT * FROM item, itemtype WHERE itemTypeId = ItemType.itemTypeId AND
 stock = 0;
```

Static Fields

- `todaysRentalCtr`: Today's rental transaction count.
- `totalDueCtr`: Transactions with outstanding balances.
- `todaySellCtr`: Today's sales transaction count.
- `todaysTotalDue`: Sum of today's outstanding payments.
- `todaysTotalSell`: Sum of today's sales.
- `todayTotalRental`: Sum of today's rentals.
- `totalDueAmount`: Overall outstanding amount.
- `stockOut`: Out-of-stock item count.

Used Classes

- **DBConnection**: Manages database connections.
- **FXMLLoader**: Loads FXML views into new windows.
- **JFXButton**: Material-design buttons.
- **Label**: Displays information on the dashboard.
- **SellListController/RentalListController**: Manage detailed transaction views.

Dependencies

- **JavaFX**: For UI development.
- **JFoenix**: For enhanced material-design components.
- **Java SQL**: For database access.
- **Java Time**: For date operations.

8. CustomersRentalListController

Overview

The `CustomersRentalListController` manages the rental history interface for a specific customer in a JavaFX GUI application. It displays rental records, calculates financial summaries, and presents transaction history in a structured tabular format.

Key Features

1. **Rental Record Display**
    - Fetches and displays rental history for a specific customer.
    - Shows detailed rental transactions in a TableView.
    - Displays rental and return dates for each transaction.
2. **Financial Summary**
    - Calculates the total amount paid for rentals.
    - Tracks outstanding payments (amount due).
    - Shows the number of rental transactions completed.
3. **Date Information**
    - Displays the current date on the interface.
    - Shows rental and expected return dates for each transaction.
4. **Data Filtering**
    - Filters rental history by customer ID (stored as a static variable).
    - Organizes transactions in a consistent format.

Main Components

- **TableView<Rent>**: Displays rental transaction history.
- **TableColumn** elements: Represent individual transaction attributes (rental ID, customer ID, item ID, etc.).
- **Label** elements: Display summary information (total rentals, amount due, etc.).
- **ObservableList<Rent>**: Stores and updates rental records shown in the table.

UI Components

- **TableView**:
  - `tblRecent`: Main table for displaying rental history.
- **Table Columns**:
  - `rentID`: Rental identifier.
  - `cusID`: Customer identifier.
  - `itemID`: Item identifier.
  - `rentalDate`: Date when the item was rented.
  - `returnDate`: Expected return date.
  - `paid`: Amount paid for rental.
  - `due`: Outstanding balance.
  - `empName`: Employee who processed the transaction.
- **Display Labels**:
  - `lblSellCount`: Total number of rental transactions.
  - `lblDue`: Current amount due.
  - `lblAmount`: Total amount paid for rentals.
  - `today`: Current date.

Database Interactions

The controller interacts with an SQL database to:

- Retrieve rental records filtered by customer ID.
- Calculate financial summaries.
- Process transaction details into the `Rent` model object.

Key Methods

Initialization

- `initialize(URL, ResourceBundle)`:
  - Sets up UI components and loads customer rental data.
  - Initializes TableView columns with `PropertyValueFactory`.
  - Sets the current date in the `today` label.
  - Connects to the database.
  - Executes an SQL query to fetch rental records.
  - Calculates financial totals and updates the UI.

Data Operations

- **SQL Query** Example:

  ```
  sql
  CopyEdit
  SELECT * FROM rentals WHERE Customers.customerID = [customerID]
  ```

- **Financial Calculations**:
  - Accumulate total payments from the `paid` field.
  - Accumulate outstanding amounts from the `amountDue` field.
  - Count total number of rental transactions.

Used Classes

- `Rent`: Model class for rental transaction data.
- `DBConnection`: Utility for database connection.
- `PropertyValueFactory`: Links table columns to model properties.
- `FXCollections`: Creates observable lists for JavaFX.
- `TableView` / `TableColumn`: JavaFX components.
- `Label`: JavaFX component for text display.

Static Fields

- `customerID`: Stores the ID of the customer being viewed.

Dependencies

- **JavaFX**: UI components (TableView, Label, etc.).
- **Java SQL**: Database connection and queries.
- **Custom 'Rent' model**: Rental transaction data.
- **Custom 'DBConnection' utility**: Database management.
- **Java Time**: Displaying current date (`LocalDate`).

9. CustomerPurchaseListController

Overview

The `CustomerPurchaseListController` manages the customer purchase history interface. It displays purchase records, calculates financial totals, and presents transaction history in a structured table.

Key Features

1. **Purchase Record Display**
    - Fetches and displays purchase history for a specific customer.
    - Shows detailed purchase transactions in a TableView.
    - Displays financial summaries.
2. **Financial Summary**
    - Calculates total amount spent.
    - Tracks outstanding payments (amount due).
    - Displays the number of transactions completed.
3. **Date Information**
    - Displays the current date.
    - Includes purchase dates in transaction records.
4. **Data Filtering**
    - Filters purchase history by customer ID.
    - Organizes transactions consistently.

Main Components

- **TableView<Purchase>**: Displays purchase transaction history.
- **TableColumn** elements: Represent transaction attributes (purchase ID, customer ID, item ID, etc.).
- **Label** elements: Show financial summaries.
- **ObservableList<Purchase>**: Stores and updates purchase records.

UI Components

- **TableView**:
  - `tblRecent`: Main table for displaying purchase history.
- **Table Columns**:
  - `purID`: Purchase identifier.
  - `cusID`: Customer identifier.
  - `itemID`: Item identifier.
  - `date`: Purchase date.
  - `qty`: Quantity purchased.
  - `paidAmount`: Amount paid.
  - `dueAmount`: Outstanding balance.
  - `empName`: Employee who processed the transaction.
- **Display Labels**:
  - `lblSellCount`: Total number of transactions.
  - `lblDuetoday`: Current amount due.
  - `lblAmount`: Total transaction amount.

Database Interactions

- Retrieves purchase records by customer ID.
- Calculates financial totals.
- Processes records into the `Purchase` model object.

Key Methods

Initialization

- `initialize(URL, ResourceBundle)`:
  - Sets up UI components.
  - Binds columns to model properties.
  - Connects to the database.
  - Executes SQL query to retrieve purchases.
  - Updates financial totals.

Data Operations

- **SQL Query** Example:

```
sql
CopyEdit
SELECT * FROM purchases WHERE Customers.customerID = [customerID]
```

Used Classes

- `Purchase`: Model class for purchase data.
- `DBConnection`: Utility for database connection.
- `PropertyValueFactory`: Links columns to model properties.
- `FXCollections`: JavaFX observable list utility.
- `TableView` / `TableColumn`: JavaFX components.
- `Label`: JavaFX component for text display.

Dependencies

- **JavaFX**: UI components.
- **Java SQL**: Database interaction.
- **Custom 'Purchase' model**: Purchase data.
- **Custom 'DBConnection' utility**: Manages database connections

10. InitializerController

Overview

The `InitializerController` manages the system startup process. It loads essential data from the database before launching the main application interface, displaying progress and task information.

Key Features

1. **Progressive Data Loading**
    - Displays real-time loading progress.
    - Updates current loading task name.
    - Performs background loading to keep the UI responsive.
2. **Comprehensive Data Initialization**
    - Loads customers, inventory, sales, rentals, and account data.
    - Loads dashboard statistics.
3. **Session Management**
    - Maintains session username from login.
    - Passes session information to the main application.
4. **Application Launch**
    - Closes the initialization window after loading.
    - Opens and configures the main application window.

Main Components

- **JFXProgressBar**: Displays loading progress.
- **Label**: Shows current loading task name.
- **LoadRecords (Task)**: Background thread for loading operations.
- **Scene**: Main application window.

UI Components

- `progressIndicator`: Loading status indicator.
- `taskName`: Current loading task label.

Database Interactions

- Loads customers, inventory items, sales, rentals, accounts, and dashboard statistics:
    - Out-of-stock items.
    - Amounts due.
    - Today's sales and rentals.
    - Today's due amounts.

Key Methods

Initialization

- `initialize(URL, ResourceBundle)`:
    - Configures UI elements and binds them to loading tasks.
    - Starts the `LoadRecords` task to retrieve all necessary data.
    - Launches the main application window on completion.

Application Launch

- `loadApplication()`:
    - Opens the `base.fxml` main application window.
    - Configures window properties: title, icon, maximization.

LoadRecords Inner Class

- `call()`:
    - Connects to the database.
    - Executes multiple SQL queries.
    - Loads retrieved data into appropriate collections.
    - Updates progress messages dynamically.

Data Structures

- **ObservableList**: For customers, items, sales, rentals, and accounts.
- **ArrayList**: For IDs, names, and reference data.
- **TreeMap**: For item type mappings.

Data Flow

1. Connect to database.
2. Load customer and inventory data.
3. Load sales and rental transactions.
4. Load account information.
5. Calculate dashboard statistics.
6. Complete initialization and launch the application.

Dependencies

- **JavaFX**: UI components.
- **JFoenix**: Enhanced UI elements.
- **SQL Database**: Data source.
- **Model Classes**: `Customer`, `Item`, `Purchase`, `Rent`, `Account`.

11. EmployeeListController

Overview

The **EmployeeListController** manages the employee interface within a retail/rental management system. It provides functionality to view, edit, and delete employee accounts across different access levels (Employee and Admin). The controller offers a tabular display of employees and supports detailed editing of user credentials and permissions.

Key Features

1. **Employee Record Display**
    - Shows all employee accounts in a table format.
    - Displays usernames, passwords, email addresses, and access levels.
    - Enables selection of records for detailed editing.
2. **Employee Management**
    - Creates new employee accounts.
    - Updates existing employee information.
    - Deletes employee accounts.
    - Manages access level permissions.
3. **Interactive Table**
    - Double-click functionality to load employee details into form fields.
    - Visual presentation of all employees in the system.
    - Quick access to employee records for management.
4. **Form Management**
    - Input fields for employee credentials.
    - Access level selection via dropdown menu.
    - Validation of employee information.

Main Components

- **TableView<Employee>**: Displays the list of all employees.
- **JFXTextField/JFXPasswordField**: Input fields for employee details.
- **JFXComboBox**: Dropdown for selecting access levels.
- **JFXButton**: Action buttons for updating and deleting.
- **ObservableList**: Data structures for table data and dropdown options.

UI Components

Input Fields

- **txtUser**: Username input field.
- **txtPass**: Password input field (masked).
- **txtEmail**: Email address input field.
- **cboAccessLevel**: Dropdown for access level (Employee/Admin).

Table Components

- **tbl**: Main TableView for employee records.
- **username**: Column for usernames.
- **pass**: Column for passwords.
- **email**: Column for email addresses.
- **access**: Column for access levels.

Action Buttons

- **btnAddNew**: Add new employee.
- **btnDelete**: Delete selected employee.
- **btnAddIcon**: Icon for add button.

Database Interactions

- Retrieves all employee records from the `user` table.
- Updates employee information with new values.
- Deletes employee accounts by username.
- Stores employee credentials with appropriate access levels.

Key Methods

Initialization

- **initialize(URL, ResourceBundle)**:
    - Sets up UI components and loads employee data.
    - Populates access level dropdown (Employee, Admin).
    - Configures double-click event on table rows.
    - Loads employee data into the table.

Data Operations

- **loadTableData()**:
    - Fetches employee records from database and fills the table.
    - Configures table column cell factories.
    - Creates Employee objects and binds them to the table.
- **loadClickedContents()**:
    - Populates form fields with selected employee details.

12. InventoryController

Overview

The **InventoryController** handles inventory management within a retail/rental system. It provides functionalities to view, create, edit, and delete inventory items, managing various attributes like pricing, stock levels, and categorization. It includes a detailed form view for item management and a tabular display of inventory.

Key Features

1. **Inventory Item Display**
    - Shows all inventory items in a table view.
    - Displays detailed item information in form view.
    - Navigation through item records.
    - Visual indicators for low-stock items.
2. **Inventory Management**
    - Creates new inventory items.
    - Updates existing item information.
    - Deletes inventory records.
    - Manages item photographs.
3. **Search and Filter**
    - Searches items by ID or name.
    - Lists out-of-stock items.
    - Auto-completion for item name search.
4. **Item Categorization**
    - Assigns items to categories.
    - Designates items for rental and/or sale.
    - Sets pricing for sale and rental separately.

Main Components

- **TableView<Item>**: Displays inventory list.
- **JFXTextField**: Input fields for item information.
- **JFXCheckBox**: Selection for rental/sale designation.
- **JFXComboBox**: Dropdown for item types.
- **Circle**: Displays item image.
- **JFXButton**: Action buttons.
- **ObservableList**: Data structure for table data.

UI Components

Input Fields

- **txtItemName**: Item name input.
- **txtType**: Dropdown for item type selection.
- **txtRentRate**: Rental rate input.
- **txtStock**: Stock quantity input.
- **txtPrice**: Sales price input.
- **txtSearch**: Search input field.
- **chkRent**: Checkbox for rental designation.
- **chkSale**: Checkbox for sale designation.

Display Components

- **itemID**: Displays item ID.
- **imgCustomerPhoto**: Displays item image.
- **lblPageIndex**: Displays record navigation status.
- **lblMode**: Displays current mode (Entry/Navigation).
- **lblSearchResults**: Displays search results status.

Table Components

- **tbl**: Main TableView for inventory items.
- **columnItemID**: Item ID column.
- **columnItemName**: Item name column.
- **columnItemType**: Item type column.
- **columnForRent**: Rental availability column.
- **columnForSale**: Sale availability column.
- **columnRentalRate**: Rental rate column.
- **columnPrice**: Sale price column.
- **columnStock**: Stock quantity column.

Action Buttons

- **btnAddIcon**: Add/edit mode toggle button.
- **btnDelete**: Delete selected item.
- **btnSearch**: Search items.
- **btnListAll**: List all items.
- **btnOutOfStock**: List out-of-stock items.
- **btnPrevEntry**: Navigate to previous item.
- **btnNextEntry**: Navigate to next item.
- **btnGoBack**: Return from list to form view.

Database Interactions

- Retrieves all inventory records from `item` and `itemtype` tables.
- Adds new inventory items.
- Updates existing item information.
- Deletes inventory items by ID.
- Searches items by name or ID.
- Filters for out-of-stock items.

Key Methods

Initialization

- **initialize(URL, ResourceBundle)**:
    - Sets up UI components and loads inventory data.
    - Populates item type dropdown.
    - Configures search auto-completion.
    - Initializes default view settings.

Data Operations

- **reloadRecords()**: Refreshes inventory data from the database.
- **recordNavigator()**: Displays selected item details in the form view.
- **addRecordToDatabase()**: Adds new item to the database.
- **updateRecord()**: Updates an existing item.
- **btnDelAction()**: Deletes selected item.

UI Management

- **setView()**: Prepares form view for item details.
- **listView()**: Prepares table view for item listing.
- **btnAddMode()**: Toggles between entry and navigation modes.
- **listAllItems()**: Displays all items.
- **outOfStockList()**: Displays only out-of-stock items.

Search Functions

- **searchWithID()**: Search by item ID.
- **searchWithName()**: Search by item name.
- **btnSearchAction()**: Handles search actions and displays results.

Validation

- **checkFields()**: Validates input fields before database operations.

Data Structures

- **ObservableList<Item>**: Lists for current items and search results.
- **ArrayList<String>**: List of item names for auto-completion.
- **TreeMap<String, Integer>**: Maps item type names to IDs.

User Access Control

- Deleting items requires Admin access.
- Non-Admin users can view and edit, but not delete.

Data Flow

1. Initialize controller and load inventory data.
2. Display items in form view with navigation controls.
3. Allow searching, filtering, and listing operations.
4. Enable detailed editing of item information.
5. Process user actions for adding, updating, or deleting items.
6. Validate input before database operations.
7. Refresh inventory data after modifications.

Item Attributes

- **ID** (auto-generated).
- **Name**.
- **Type** (predefined categories).
- **Stock Quantity**.
- **Sale Price** and **Rental Rate**.
- **Photo** (optional).

13.ItemTypeController

Overview

The `ItemTypeController` manages the item type interface within a retail/rental management system. It enables viewing, creating, editing, and deleting item types used for categorizing inventory. It offers both a form view for detailed management and a tabular view for listing all item types alongside their assigned item counts.

Key Features

1. Item Type Display

- Displays all item types in a tabular format.
- Provides a form view with detailed type information.
- Shows the count of items assigned to each type.
- Supports navigation between records.

2. Item Type Management

- Creates new item types.
- Updates existing type information.
- Deletes type records with safeguards (e.g., foreign key constraints).
- Auto-generates new type IDs.

3. Data Operations

- Auto-refreshes type listings after modifications.
- Prevents deletion of types that have associated items.
- Enables double-click selection for editing.

Main Components

- `TableView<ItemType>`: Displays all item types.
- `JFXTextField`: Input fields for type information.
- `JFXButton`: Action buttons for operations.
- `ObservableList`: Data structure for managing table data.

UI Components

Input Fields

- `txtItemType`: Input field for type name.

Display Components

- `lblType`: Label displaying the type ID.

Table Components

- `tbl`: Main TableView for item types.
- `typeID`: Column for type IDs.
- `typeName`: Column for type names.
- `totalItems`: Column showing the number of assigned items.

Action Buttons

- `btnUpdate`: Update item type (with `btnAddIcon`).
- `btnDelete`: Delete item type (with `btnAddIcon1`).
- `btnRefresh`: Refresh the data display.

Database Interactions

- Retrieves all item type records from the `itemtype` table.
- Counts associated items from the `item` table.
- Adds new item types.
- Updates existing types.
- Deletes types by ID, respecting foreign key constraints.

Key Methods

Initialization

- `initialize(URL, ResourceBundle)`: Sets up UI components and loads data.
- Configures table double-click behavior.

Data Operations

- `setTableData()`: Loads item types with item counts.
- `reload()`: Refreshes data and resets input fields.
- `addOrUpdateItemType()`: Adds or updates an item type.
- `deleteItemType()`: Deletes the selected item type.
- `loadContents()`: Populates form fields based on selection.

Data Flow

1. Initialize controller and load item data.
2. Display types in a table view with item counts.
3. Enable detailed selection and editing.
4. Allow adding, updating, or deleting types.
5. Validate operations via database constraints.
6. Refresh the view after modifications.

Item Type Attributes

- **ID** (auto-generated).
- **Name**.
- **Total Items Count** (calculated dynamically).

Error Handling

- SQL error code management.
- Special handling for foreign key constraint violations (error 1451).
- User-friendly error messages via `PromptDialogController`.

14.NewEmployeeController

Overview

The `NewEmployeeController` manages the interface for adding new employees to the system. It allows users to input employee details, validate information, and save records into the database. It ensures the system maintains complete and accurate employee profiles.

Key Features

1. **Employee Registration**
    - Collects detailed information about employees.
    - Validates inputs before submission.
    - Saves new employee records into the database.
    - Supports uploading and previewing employee photos.
2. **Data Validation**
    - Checks for mandatory fields like name, email, and mobile number.
    - Validates email format and mobile number length.
    - Prevents duplicate entries by checking existing records.
3. **User Interaction**
    - Provides clear prompts and error messages.
    - Clears the form after successful submission or reset.

Main Components

- `JFXTextField`: Input fields for employee details.
- `JFXButton`: Action buttons for submitting or clearing the form.
- `ImageView`: Displays the employee's photo.
- `FileChooser`: Selects an image from the local file system.

UI Components

- **Input Fields**
    - `txtName`: Employee's full name.
    - `txtMobile`: Employee's mobile number.
    - `txtEmail`: Employee's email address.
    - `txtAddress`: Employee's address.
- **Image Components**
    - `imgProfile`: Image view for displaying employee's photo.
- **Action Buttons**
    - `btnSave`: Save new employee information.
    - `btnClear`: Reset all form fields.

Database Interactions

- Insert a new employee record into the `employee` table.
- Optionally store the employee photo path.

Key Methods

- **Initialization**
  - `initialize(URL, ResourceBundle)`: Sets up initial UI configurations.
- **Form Actions**
  - `saveEmployee()`: Validates and saves employee data.
  - `clearForm()`: Clears all input fields.
  - `uploadImage()`: Handles selecting and displaying an employee photo.

Data Flow

1. User inputs employee data.
2. System validates the entered data.
3. System saves validated data to the database.
4. The form clears for a new entry.

Error Handling

- Validation messages for missing or incorrect inputs.
- Database error catching during insert operations.
- Image upload error management.

15.LogInController

Overview

The `LogInController` manages the login functionality for the system. It handles user authentication by verifying the username and password against stored records. Upon successful login, it navigates the user to the main application dashboard.

Key Features

1. **User Authentication**
    - Accepts username and password input.
    - Verifies credentials against database records.
    - Grants access if the credentials are correct.
2. **Access Control**
    - Redirects authenticated users to the home screen.
    - Displays error messages for invalid login attempts.
3. **Security Measures**
    - Encrypts or hashes passwords before validation (if applied).
    - Limits the number of login attempts (optional enhancement).

Main Components

- `JFXTextField`: For entering the username.
- `JFXPasswordField`: For entering the password securely.
- `JFXButton`: Button to submit the login form.

UI Components

- **Input Fields**
    - `txtUsername`: Text field for entering username.
    - `txtPassword`: Password field for entering password.
- **Action Button**
    - `btnLogin`: Button to trigger login verification.

Database Interactions

- Fetch user records from the `users` table based on the entered username.
- Compare the entered password with the stored password.

Key Methods

- **Initialization**
  - `initialize(URL, ResourceBundle)`: Prepares the login form (e.g., clearing fields).
- **Login Process**
  - `handleLogin()`:
    - Validates that both fields are not empty.
    - Queries the database to check credentials.
    - Loads the main application window if successful.
    - Shows an error alert if login fails.
- **Navigation**
  - Loads the dashboard/home screen upon successful login.

Data Flow

1. User enters username and password.
2. System validates inputs are not empty.
3. System checks credentials against the database.
4. If correct, system opens the dashboard.
5. If incorrect, displays an error message.

Error Handling

- Empty field warnings.
- Invalid username or password error message.
- Database connection error handling.

16.RentalListController

Overview

The **RentalListController** manages the rental report interface within a retail or rental management system. It enables users to view rental transaction records for the current day or across all time periods. This controller displays detailed rental information, including financial summaries and transaction counts, in a structured table with aggregated statistics.

Key Features

1. Rental Report Display

- Shows rental transactions in a tabular format.
- Displays detailed information for each transaction.
- Filters rentals by the current date when specified.
- Calculates and presents financial summaries.

2. Financial Tracking

- Tallies the total payments received.
- Calculates outstanding amounts due.
- Counts the total number of rental transactions.
- Presents all monetary values with currency indicators.

3. Date-Based Filtering

- Option to view only today's rentals.
- Option to view all historical rental records.
- Displays the current date for reference.

Main Components

- **TableView<Rent>**: Displays the list of rental transactions.
- **TableColumn**: Various columns representing rental data fields.
- **Label**: Components for displaying aggregated financial and transaction statistics.
- **ObservableList**: Data structures used to manage table data dynamically.

UI Components

Display Components

- **lblHeader**: Label displaying the report title (changes based on selected filter).
- **lblSellCount**: Label showing the total number of rental transactions.
- **lblAmount**: Label displaying the total payments received.
- **lblDue**: Label showing the total amounts still due.
- **today**: Label displaying the current date.

Table Components

- **tblRecent**: Main `TableView` for rental transactions.
- **rentID**: Column for rental transaction IDs.
- **cusID**: Column for customer IDs.
- **itemID**: Column for item IDs.
- **rentalDate**: Column for rental start dates.
- **returnDate**: Column for expected return dates.
- **empName**: Column for employee usernames who processed the rentals.
- **paid**: Column for amounts paid.
- **due**: Column for outstanding balances.

Database Interactions

The controller communicates with the SQL database to:

- Retrieve all rental records from the `rentals` table.
- Filter rental records by the current date when needed.
- Extract transaction details for display and calculation of statistics.

Key Methods

Initialization

- **initialize(URL, ResourceBundle)**:
    - Sets up UI components and binds data to the table.
    - Displays the current date.
    - Determines whether to show today's rentals or all rentals.
    - Executes the appropriate database query based on the selected report type.

Data Operations

- **showReport()**:
    - Processes rental data retrieved from the database.
    - Configures the table columns for proper data display.
    - Calculates summary statistics such as totals and counts.
    - Populates the table with individual rental records.
    - Updates the summary labels with the calculated values.

Static Variables

- **todayFlag**: A boolean flag that indicates whether the controller should display only today's rental transactions.

Data Flow

1. Initialize the controller and determine the report type (today's rentals or all-time).
2. Execute the appropriate database query to fetch rental records.
3. Process the retrieved records and calculate aggregate statistics.
4. Populate the `TableView` with detailed rental records.
5. Update the summary labels with the calculated financial and transaction data.

Rental Transaction Attributes

- **Rental ID**: Unique identifier for each rental.
- **Customer ID**: Reference to the renting customer.
- **Item ID**: Reference to the rented item.
- **Rental Date**: Start date of the rental period.
- **Return Date**: Expected end date of the rental period.
- **Paid Amount**: Total payment received from the customer.
- **Amount Due**: Remaining balance owed by the customer.
- **Employee Username**: Username of the employee who processed the rental transaction.

Error Handling

- Proper logging of SQL exceptions to assist with debugging and maintain system stability.

17.PromptDialogController

Overview

The **PromptDialogController** is responsible for managing the display of modal dialog boxes within the retail/rental management system. It provides a standardized method to present information, alerts, and error messages to users. By creating top-level, application-modal windows, it ensures that critical information is acknowledged before users can continue their workflow.

Key Features

1. Modal Dialog Display

- Creates application-modal windows that must be dismissed before proceeding.
- Ensures that important messages are clearly seen and acknowledged by users.
- Maintains always-on-top behavior to guarantee visibility.

2. Flexible Message Presentation

- Supports custom header text to provide context for each dialog.
- Offers a multi-line text area for displaying detailed messages.
- Utilizes a simple single-button interface for user acknowledgment.

3. Consistent User Experience

- Provides a standardized dialog appearance across the entire application.
- Displays dialogs in non-resizable windows for controlled and predictable presentation.
- Implements a simple interaction model with a single close action.

Main Components

- **Label**: Displays the header or context text for the dialog.
- **JFXTextArea**: Shows the main content/message of the dialog.
- **JFXButton**: Acts as the acknowledgment and close button.
- **Stage**: JavaFX window that represents the modal dialog.

UI Components

Display Components

- **lblHeader**: Label for displaying the dialog's title or context.
- **txtError**: Text area containing the main message or error information.
- **btnClose**: Button allowing users to acknowledge the message and close the dialog.

Key Methods

Constructor

- **PromptDialogController(String header, String error)**:
    - Creates and displays a new dialog window.
    - Initializes a new `Stage` with appropriate modal properties.
    - Loads the dialog layout from an FXML file.
    - Sets the header and main message content dynamically.
    - Configures the close button’s event handler.
    - Displays the dialog centered on the screen.

Dialog Properties

- Always displayed on top of other application windows.
- Application-modal, blocking interactions with other windows until closed.
- Non-resizable, maintaining a consistent and clean appearance.
- Automatically centered on the user's screen for easy access.

Usage Patterns

This controller is used throughout the application for:

- **Success Confirmations**: e.g., "Operation Successful."
- **Error Notifications**: e.g., "Authentication Error!"
- **Warning Messages**: e.g., "Fields cannot be empty!"
- **Information Displays**: e.g., database constraint violation messages.

Error Handling

- Internal exception handling for FXML loading errors.
- Outputs stack traces for debugging purposes when necessary.

Implementation Notes

- Uses component lookup to dynamically find and bind UI elements from the FXML file.
- Dynamically attaches event handlers to the close button at runtime.
- Each dialog instance is independent with no persistent state between dialogs.

18.TransactionController

Overview

The **TransactionController** manages the transaction processing interface within the retail management system. It oversees the finalization of both sales and rental transactions, handling customer accounts, processing payments, updating inventory, and recording financial operations. This controller ensures the validity of transaction details, interacts with the database to persist transaction records, and provides immediate feedback to the user upon successful completion.

Key Features

1. Transaction Processing

- Finalizes both purchase and rental transactions.
- Records financial transactions with appropriate type codes.
- Automatically updates inventory stock levels.
- Links transactions directly to customer accounts.
- Manages transaction ID generation and tracking.

2. Account Integration

- Enables selection of customer accounts for transaction association.
- Validates customer account existence and accuracy.
- Links each transaction to the selected customer account.

3. Transaction Type Classification

- Supports multiple transaction type codes.
- Categorizes transactions appropriately for financial reporting.
- Maintains a mapping between transaction types and database codes.

4. Dual Transaction Support

- Handles both sales (purchases) and rental transactions.
- Applies distinct processing logic based on transaction type.
- Maintains accurate records for each transaction category.

Main Components

- **JFXButton**: Action buttons for transaction operations.
- **JFXComboBox**: Dropdown menus for selecting customer accounts and transaction types.
- **JFXDatePicker**: Component for choosing the transaction date.
- **Label**: Fields displaying transaction and customer details.
- **AnchorPane**: Layout container for UI organization.
- **FontAwesomeIconView**: Icons representing button states.

UI Components

Display Components

- **lblTrID**: Displays the generated transaction ID.
- **lblCategory**: Shows the transaction category.
- **lblCusID**: Displays the customer's name.
- **lblPurID**: Shows the purchase or rental ID.
- **lblPurchaseOrRent**: Indicates whether the transaction is a sale or rental.
- **lblAmounPaid**: Displays the payment amount.
- **lblDue**: Shows the remaining due amount.
- **trPane**: Main container for the transaction window components.
- **btnIcon**: Icon reflecting the current button state.

Input Components

- **txtTrDate**: Date picker for selecting the transaction date.
- **cboChooseAccount**: Dropdown for selecting a customer account.
- **cboTrType**: Dropdown for selecting the transaction type.

Action Components

- **btnCancel**: Cancels the current transaction process.
- **btnProceed**: Finalizes and processes the transaction.

Database Interactions

The controller interacts with the SQL database to:

- Retrieve customer information by ID.
- Generate sequential transaction IDs.
- Access customer account details.
- Fetch transaction type codes.
- Insert new purchase or rental records.
- Record corresponding financial transactions.
- Update inventory stock levels accordingly.

Key Methods

Initialization

- **initialize(URL, ResourceBundle)**: Sets up UI elements and event handlers.
- **loadTransactionWindowContents()**:
    - Loads transaction window data.
    - Retrieves customer details.
    - Generates and displays transaction IDs.
    - Populates dropdown menus with account and transaction type options.
    - Initializes default field values.

Transaction Processing

- **btnProceedAction(ActionEvent)**:
    - Determines transaction type and delegates processing.
- **finalizePurchase()**:
    - Inserts purchase records.
    - Creates financial transaction entries.
    - Updates stock levels.
    - Provides completion confirmation.
- **finalizeRental()**:
    - Inserts rental records.
    - Records corresponding financial transactions.
    - Updates stock inventory.
    - Displays success feedback.

UI Management

- **hideWindow()**: Closes the transaction window upon completion.

Static Variables

- **cusName**: Current customer name.
- **purchaseId**: ID associated with a purchase transaction.
- **rentalId**: ID associated with a rental transaction.
- **purchaseQty**: Quantity of items purchased.
- **itemID**: Identifier of the transacted item.
- **cusID**: Current customer ID.
- **payAmount**: Payment amount received.
- **rentalReturnDate**: Expected return date for rentals.
- **rentOrSale**: Boolean flag indicating sale (true) or rental (false).
- **due**: Amount remaining due after transaction.
- **trType**: Mapping of transaction type names to their codes.
- **stock**: Current stock level for the item.

Data Flow

1. Controller is initialized and loads necessary data from the database.
2. Transaction details are populated via static variables.
3. User selects customer account and transaction type.
4. User finalizes the transaction by clicking the proceed button.
5. System determines whether the transaction is a purchase or rental.
6. Appropriate database records are inserted.
7. Inventory stock levels are updated automatically.
8. User receives confirmation upon successful transaction.
9. The transaction window closes automatically.

Validation

- Verifies that the customer exists in the database.
- Confirms the customer has an active account.
- Ensures sufficient inventory is available for the transaction.
- Validates transaction details before inserting into the database.

Error Handling

- Handles SQL errors gracefully with descriptive messages.
- Utilizes **PromptDialogController** for user-facing error notifications.
- Rolls back transaction data on encountering critical errors.

19.SellListController

Overview

The **SellListController** manages the display of sales transaction records within the retail management system. It offers a clear, tabular visualization of purchase history, enabling users to filter transactions by today's sales or view all historical sales. The controller also calculates and displays summary statistics such as the total number of transactions, total paid amounts, and outstanding dues. It interacts with the database to retrieve and organize sales data in a structured and user-friendly format.

Key Features

1. Sales Transaction Display

- Displays a complete history of sales transactions in a table.
- Provides filtering to view only today's transactions.
- Presents detailed transaction information including customer, item, and payment data.
- Calculates and displays summary statistics for easy review.

2. Data Aggregation

- Counts the total number of sales transactions.
- Calculates total sales revenue.
- Computes outstanding amounts still due.
- Displays the current date as a reference for daily reports.

3. Visual Representation

- Structures sales data clearly within a TableView.
- Provides multiple columns for detailed transaction attributes.
- Clearly labels summary statistics for quick reference.

Main Components

- **TableView<Purchase>**: Displays the list of sales transactions.
- **TableColumn**: Represents different transaction details (ID, customer, item, amounts, etc.).
- **Label**: Displays summary statistics such as total sales and outstanding dues.
- **PreparedStatement**: Prepares SQL queries for fetching sales data.
- **ObservableList**: Holds data for populating the TableView.

UI Components

Display Components

- **lblHeader**: Displays the report title ("Today's Sales" or "All Sales").
- **lblSellCount**: Shows the total number of sales transactions.
- **lblDue**: Displays the total outstanding amount.
- **lblAmount**: Shows the total amount of sales.
- **today**: Displays the current date.

Table Components

- **tblRecent**: TableView for listing sales transactions.
- **purID**: Column for purchase IDs.
- **cusID**: Column for customer IDs.
- **itemID**: Column for item IDs.
- **date**: Column for transaction dates.
- **qty**: Column for quantity of items purchased.
- **paidAmount**: Column for amounts paid.
- **dueAmount**: Column for due amounts.
- **empName**: Column for employee names associated with transactions.

Database Interactions

The controller interfaces with the SQL database to:

- Retrieve all historical sales records.
- Filter sales transactions based on the current date.
- Calculate aggregate sales statistics such as totals and counts.
- Map SQL result sets into **Purchase** objects for easy display in the TableView.

Key Methods

Initialization

- **initialize(URL, ResourceBundle)**:
    - Sets up the TableView columns and binds them to **Purchase** object properties.
    - Establishes the database connection.
    - Checks the **todayFlag** to determine whether to load today’s sales or all sales.
    - Prepares and executes the corresponding SQL query.
    - Calls the method to populate the table and update summary statistics.

Data Management

- **showReport()**:
    - Prepares and executes the SQL query based on the selected filter (today or all).
    - Maps the result set to **Purchase** objects.
    - Calculates summary statistics including total sales, paid amount, and due amount.
    - Updates UI labels with calculated statistics.
    - Populates the TableView with sales transaction records.

Static Variables

- **todayFlag**: Boolean variable indicating whether to display only today's sales (`true`) or all sales (`false`).

Data Flow

1. Controller initializes and establishes a connection to the database.
2. **todayFlag** is checked to determine which sales data to retrieve.
3. An appropriate SQL query is prepared and executed.
4. Sales transaction data is retrieved from the database.
5. The controller calculates and updates summary statistics (sales count, total amount, and dues).
6. UI labels are updated to reflect current statistics.
7. The TableView is populated with the retrieved sales transaction records.

20. SellsController

Overview

The **SellsController** manages the sales transaction interface in the retail management system. It facilitates the creation of new sales transactions by validating customer and item information, calculating costs, processing payments, and displaying sales history. The controller offers both tabular and chart-based views of sales data and integrates with the transaction processing system to finalize sales.

Key Features

1. Sales Transaction Processing

- Creates new sales transactions, ensuring customer and item validity.
- Calculates total cost based on item price and quantity.
- Processes full or partial payments, tracking outstanding balances.
- Verifies product availability and customer account status.
- Integrates with the transaction system for finalizing sales.

2. Sales History Display

- Displays user-specific sales history in a structured table format.
- Visualizes sales revenue over time using a line chart.
- Provides the ability to toggle between table and chart views.
- Displays detailed information about each transaction.

3. Input Validation

- Validates customer IDs against the database.
- Ensures item availability for sale.
- Confirms valid quantity values.
- Provides visual and textual feedback on validation errors.

4. Auto-Completion Support

- Provides auto-completion for customer IDs.
- Offers auto-completion for inventory item IDs.
- Enhances data entry efficiency and accuracy.

Main Components

- **JFXTextField**: Input fields for transaction details.
- **JFXDatePicker**: Date selection for sale date.
- **JFXButton**: Action buttons for operations.
- **TableView<Purchase>**: Displays sales transaction history.
- **LineChart**: Visualizes sales revenue over time.
- **JFXSnackbar**: Notification display for validation feedback.
- **ObservableList**: Data structure for table data.
- **ArrayList**: Stores collections for auto-completion data.

UI Components

Input Fields

- **txtCustomerId**: Customer ID input field with auto-completion.
- **txtItemId**: Item ID input field with auto-completion.
- **txtPayAmount**: Payment amount input field.
- **txtQty**: Quantity input field.
- **txtDate**: Date picker for purchase date.

Display Components

- **lblId**: Displays purchase transaction ID.
- **lblCost**: Displays the calculated cost of the transaction.
- **lblVerify**: Indicates the verification status of inputs.
- **rightPane**: AnchorPane container for layout.
- **btnIcon**: Icon view for the proceed button state.
- **btnChartIcon**: Icon view for chart toggle button.

Table Components

- **tblRecent**: TableView for displaying recent sales transactions.
- **purID**: Column for purchase IDs.
- **cusID**: Column for customer IDs.
- **itemID**: Column for item IDs.
- **date**: Column for purchase dates.
- **qty**: Column for quantities purchased.
- **paidAmount**: Column for paid amounts.
- **dueAmount**: Column for due amounts.

Chart Components

- **lineChart**: LineChart for visualizing sales revenue over time.
- **dateAxis**: X-axis for dates in the chart.
- **amountAxis**: Y-axis for monetary values.

Action Components

- **btnProceed**: Button to proceed with the sales process.
- **btnBarchart**: Button to toggle between table and chart views.

Database Interactions

The controller interacts with an SQL database to:

- Verify the existence and account status of customers.
- Check item availability, stock levels, and prices.
- Record new sales transactions.
- Retrieve user-specific sales history.
- Fetch aggregated sales data for chart visualization.
- Generate sequential purchase IDs for transactions.

Key Methods

Initialization

- **initialize(URL, ResourceBundle)**:
    - Sets up UI components and loads initial data.
    - Configures auto-completion fields for customer and item IDs.
    - Retrieves the next purchase ID.
    - Configures table columns and binds data to properties.

Transaction Processing

- **btnProceedAction(ActionEvent)**:
    - Handles two-phase sales processing:
        - Phase 1: Validates inputs and calculates total costs.
        - Phase 2: Prepares transaction data and launches the transaction UI.
    - Handles various validation scenarios and provides feedback as necessary.

Data Management

- **loadAgain(ActionEvent)**:
    - Refreshes the sales data, updates purchase ID generator.
    - Reloads user-specific sales history and clears input fields.
    - Updates chart data if the chart is visible.

Visualization

- **generateLineChart()**:
    - Queries aggregated sales revenue data by date.
    - Populates the line chart with time-series data.
    - Limits display to recent transactions for better readability.

UI Management

- **btnBarchartAction(ActionEvent)**:
    - Toggles between table view and chart view for sales visualization.

Static Variables

- **toggleTable**: Controls the display mode (table vs chart).
- **startTransaction**: Tracks the sales transaction state.
- **purchaseList**: Shared list of purchase transactions.
- **customerIDName**: Collection of customer IDs and names for auto-completion.
- **inventoryItem**: Collection of inventory items for auto-completion.
- **customerID**: Collection of valid customer IDs.
- **itemIDForSale**: Collection of items available for sale.

Data Flow

1. The controller initializes and loads customer and item data for auto-completion.
2. The user enters customer, item, and quantity information.
3. The system validates inputs and calculates the total cost.
4. The user can adjust the payment amount.
5. The system checks stock availability and customer account status.
6. The transaction is finalized through the transaction processing system.
7. Sales history is updated and displayed in the table or chart view.
8. The user can toggle between the two views or process new sales.

Validation Rules

- Customer ID must exist in the database.
- Item ID must be valid and available for sale.
- Quantity must be positive and non-zero.
- Stock levels must be sufficient for the requested quantity.
- The customer must have an account in the system.

Error Handling

- Visual feedback is provided for invalid inputs (e.g., red underlining).
- Snackbar notifications are used for validation errors.
- Critical issues (e.g., insufficient stock, missing accounts) prompt dialogs.
- SQL errors are handled with descriptive messages.

21.RentalsController

Overview

The **RentalsController** is responsible for managing the rental transaction interface within a retail/rental management system. It allows the creation of new rental transactions by validating customer and item information, calculating rental costs, processing payments, and visualizing rental history. The controller supports both tabular and chart-based views of rental data and integrates with the transaction system for finalizing rentals.

Key Features

1. Rental Transaction Processing

- Creates new rental transactions, ensuring customer and item validity.
- Calculates rental costs based on the rental duration.
- Processes full or partial payments, tracking any outstanding balance.
- Verifies the availability of rental items and customer account status.
- Integrates with the transaction system for finalizing rentals.

2. Rental History Display

- Displays user-specific rental history in a tabular format.
- Visualizes rental revenue over time using a time-series line chart.
- Allows toggling between table and chart views.
- Shows detailed information for each rental transaction.

3. Input Validation

- Validates customer IDs against the database.
- Verifies the availability of items for rental.
- Ensures valid date ranges for rental periods.
- Provides visual and textual feedback for validation errors.

4. Auto-Completion Support

- Provides auto-completion for customer IDs.
- Offers auto-completion for inventory item IDs.
- Enhances data entry efficiency and accuracy.

Main Components

- **JFXTextField**: Input fields for transaction details.
- **JFXDatePicker**: Date selection for rental periods.
- **JFXButton**: Action buttons for operations.
- **TableView<Rent>**: Displays rental transaction history.
- **LineChart**: Visualizes rental revenue over time.
- **JFXSnackbar**: Notification display for validation feedback.
- **ObservableList**: Data structure for table data.
- **ArrayList**: Stores collections for auto-completion data.

UI Components

Input Fields

- **txtCustomerId**: Customer ID input field with auto-completion.
- **txtItemId**: Item ID input field with auto-completion.
- **txtPayAmount**: Payment amount input field.
- **txtSearch**: Search input field for searching rentals.
- **txtRentalDate**: Date picker for rental start date.
- **txtReturnDate**: Date picker for expected return date.

Display Components

- **lblId**: Displays rental transaction ID.
- **lblCost**: Displays the calculated rental cost.
- **lblVerify**: Indicates the verification status of inputs.
- **lblCategory**: Displays the rental item category.
- **rightPane**: AnchorPane container for layout.
- **btnIcon**: Icon view for the proceed button state.
- **btnChartIcon**: Icon view for the chart toggle button.

Table Components

- **tblRecent**: TableView displaying recent rental transactions.
- **rentID**: Column for rental IDs.
- **cusID**: Column for customer IDs.
- **itemID**: Column for item IDs.
- **rentalDate**: Column for rental start dates.
- **returnDate**: Column for expected return dates.
- **paid**: Column for paid amounts.
- **due**: Column for due amounts.

Chart Components

- **lineChart**: LineChart for visualizing rental revenue over time.
- **dateAxis**: X-axis for dates in the chart.
- **amountAxis**: Y-axis for monetary values.

Action Components

- **btnSearch**: Button to initiate rental search.
- **btnRefresh**: Button to refresh the rental date to the current date.
- **btnProceed**: Button to proceed with the rental process.
- **btnRentalReturned**: Button for rental returns.
- **btnBarchart**: Button to toggle between table and chart views.

Database Interactions

The controller interacts with an SQL database to:

- Verify the existence and account status of customers.
- Check the availability and rental rates of items.
- Record new rental transactions.
- Retrieve rental history for the current user.
- Fetch aggregated rental data for chart visualization.
- Generate sequential rental IDs for transactions.

Key Methods

Initialization

- **initialize(URL, ResourceBundle)**:
    - Sets up UI components and loads initial data.
    - Configures auto-completion fields for customer and item IDs.
    - Retrieves the next rental ID.
    - Configures table columns and binds data to properties.

Transaction Processing

- **btnProceedAction(ActionEvent)**:
    - Handles two-phase rental processing:
        - Phase 1: Validates inputs and calculates rental costs.
        - Phase 2: Prepares transaction data and launches the transaction UI.
    - Handles validation scenarios and provides feedback accordingly.

Data Management

- **loadAgain(ActionEvent)**:
    - Refreshes rental data and updates the rental ID generator.
    - Reloads user-specific rental history and clears input fields.
    - Updates chart data if the chart is visible.

Visualization

- **generateLineChart()**:
    - Queries aggregated rental revenue data by date.
    - Populates the line chart with time-series data.
    - Limits display to recent transactions for better clarity.

UI Management

- **btnBarchartAction(ActionEvent)**:
    - Toggles between the table view and chart view for rental visualization.
- **ctrlRefreshAction(ActionEvent)**:
    - Resets the rental date to the current date.

Static Variables

- **toggleTable**: Controls display mode (table vs chart).
- **startTransaction**: Tracks the rental transaction state.
- **rentalList**: Shared list of rental transactions.
- **customerIDName**: Collection of customer IDs with names for auto-completion.
- **inventoryItem**: Collection of inventory items for auto-completion.
- **customerID**: Collection of valid customer IDs.
- **itemIDForRent**: Collection of items available for rental.

Data Flow

1. The controller initializes and loads customer and item data for auto-completion.
2. The user enters customer, item, and date range information.
3. The system validates inputs and calculates rental costs.
4. The user can adjust the payment amount.
5. The system checks the availability of items and customer account status.
6. The rental is finalized through the transaction processing system.
7. Rental history is updated and displayed in the table or chart view.
8. The user can toggle between the two views or process new rentals.
9.

Validation Rules

- Customer ID must exist in the database.
- Item ID must be valid and available for rent.
- The rental date must be earlier than the return date.
- Rental duration must be a positive value.
- Stock levels must be sufficient for the requested rental.
- The customer must have an active account in the system.

Error Handling

- Visual feedback is provided for invalid inputs (e.g., red underlining).
- Snackbar notifications are used for validation errors.
- Critical issues (e.g., insufficient stock, missing accounts) prompt dialogs.
- SQL errors are handled with descriptive messages.

22.TransactionListController

Overview

The **TransactionListController** manages the display of financial transactions in a retail management system. It provides a dual-table interface for viewing both purchase and rental transactions, each with relevant details. The controller retrieves transaction data from the database and presents it in two separate tables, ensuring clear distinction between the transaction types.

Key Features

1. Transaction History Display

- Displays purchase transactions in a dedicated table.
- Displays rental transactions in a separate dedicated table.
- Shows the current date for user reference.
- Presents transaction details in a tabular format.

2. Dual Transaction Type Support

- Separates purchase and rental transactions for clarity.
- Maintains a consistent data structure between transaction types.
- Provides parallel views for different transaction categories.

3. Database Integration

- Retrieves transaction records from financial transaction tables in the database.
- Populates table views with transaction data.
- Maps database fields to table columns.

Main Components

- **TableView<Transact>**: Displays transaction data.
- **TableColumn<Transact, ?>**: Columns for various transaction attributes.
- **Label**: Displays the current date.
- **ObservableList<Transact>**: Data structure for managing table data.

UI Components

Display Components

- **today**: Label displaying the current date.
- **tblP**: TableView for displaying purchase transactions.
- **tblR**: TableView for displaying rental transactions.

Table Components for Purchases

- **trIDP**: Column for purchase transaction IDs.
- **trDatePurch**: Column for purchase transaction dates.
- **trAccP**: Column for account IDs related to purchases.
- **purchID**: Column for purchase IDs.
- **trIssueP**: Column for transaction issuer information.

Table Components for Rentals

- **trIDR**: Column for rental transaction IDs.
- **trDateR**: Column for rental transaction dates.
- **trAccR**: Column for account IDs related to rentals.
- **rentalID**: Column for rental IDs.
- **trIssuedR**: Column for transaction issuer information.

Database Interactions

The controller interacts with an SQL database to:

- Retrieve purchase transaction records from the `financialtronpurchase` table.
- Retrieve rental transaction records from the `financialtronrental` table.
- Map database records to **Transact** model objects for display.

Key Methods

Initialization

- **initialize(URL, ResourceBundle)**:
    - Sets up UI components and loads data.
    - Configures table columns with property value factories.
    - Displays the current date in the `today` label.
    - Retrieves transaction data from the database.
    - Populates purchase and rental tables with respective transaction data.

Data Flow

1. The controller is initialized, and the current date is displayed in the `today` label.
2. A connection to the database is established.
3. Purchase transaction records are retrieved and mapped to **Transact** objects.
4. Rental transaction records are retrieved and mapped to **Transact** objects.
5. The tables are populated with the corresponding transaction data.
6. The data is displayed to the user in a clear, tabular format.

Error Handling

- SQL exceptions are handled with stack trace printing for debugging purposes.

Dependencies

- **Transact**: The model class for transaction data, representing fields like transaction ID, date, account ID, purchase/rental ID, and the issuer.
- **DBConnection**: A utility class for managing database connections.
- **JavaFX components**: TableView, TableColumn, Label for UI management.
- **Java SQL components**: Connection, PreparedStatement, ResultSet for interacting with the database.

Data Structure

The controller uses the **Transact** model class to represent transaction data with the following fields:

- **trID**: Transaction ID.
- **date**: Transaction date.
- **accID**: Account ID associated with the transaction.
- **purchaseOrRentID**: ID of the purchase or rental transaction.
- **issuedBy**: Username of the user who issued the transaction.