Task Documentation

Task: JDK 17 Installation, Maven Project Setup, IntelliJ Configuration, Git Repository Setup, MySQL Installation
Student Name: Shubhangi Bhuyare
Date Completed: 10-09-2025

---

1. Objective

The purpose of this task was to prepare a complete Java development environment by performing the following steps:

- Setting up a Git repository for version control.
- Installing and configuring MySQL for database operations.
- Installing JDK 17 and setting up the necessary environment variables.
- Installing and configuring IntelliJ IDEA as the primary development tool.

---

2. Steps Followed

1. Created a new Git repository to manage project versions efficiently.
2. Installed MySQL on the system to enable database connectivity for upcoming projects.
3. Installed JDK 17 and configured system environment variables (`JAVA_HOME` and `PATH`) to ensure Java runs properly from the terminal.
4. Installed IntelliJ IDEA and set it up for Java application development.
5. Created a new Java project in IntelliJ IDEA.
6. Inside the project, added a package named `model`.
7. Developed a `Product` class within the `model` package, which included:
    - Defining attributes such as `id`, `name`, `quantity`, and `price`.
    - Adding methods to capture user input.
    - Implementing a `display()` method to show product details.
8. Executed the program in IntelliJ to test and confirm that the `Product` class functions as expected.

---

3. Challenges Faced

- Encountered difficulties while setting up JDK 17 environment variables, which were resolved by following online guides.
- Faced minor issues during MySQL installation and configuration, fixed with the help of official MySQL documentation.

---

4. Verification

- Verified the Git setup by reviewing the repository commits and push history.
- Confirmed successful MySQL installation by logging into the MySQL command-line client.
- Checked JDK 17 setup by executing the command `java -version`.
- Validated IntelliJ configuration by creating and running a simple Java project.
- Tested the `Product` class by taking user input and displaying product information correctly in the console.

Task: Implement Inventory Manager Class with CRUD Methods, and Push to GitHub
Student Name: Shubhangi Bhuyare
Date Completed: 11-09-2025

1. Objective

The objective of this task is to extend the existing inventory system by implementing methods in the InventoryManager class to manage products effectively. These methods allow the user to add new products, delete existing ones, update product details, and search products by name.

2. Steps Taken

1. Opened IntelliJ IDEA and created a new Maven project.
2. Verified the Maven folder structure: ○ src/main/java →
   for source code

- src/main/resources → for resources (optional)
- src/test/java → for test cases

3. Inside src/main/java, created a package: com.inventory.
4. Implemented three main classes inside the package:
   - Product.java → defines product attributes (id, name, quantity, price).
   - InventorySystem.java (Inventory Manager Class) → contains CRUD methods:
     - addProduct() → Create product
     - displayAll() → Read all products
     - updateProduct() → Update product quantity
     - removeProduct() → Delete product
     - (Plus searchProduct() for convenience)
   - Main.java → menu-driven program, displays welcome message at start and thank-you message at exit.
5. Added a menu in Main.java for user operations:
   - Add Product
   - Remove Product
   - Update Product
   - Search Product by Name
   - Show Inventory Report
   - Exit
6. Ran the Main class and tested all CRUD operations.
7. Used Git commands (git init, git add, git commit, git push) to push the complete project to GitHub.

3. Challenges Encountered

- Problem: Input mismatch when using Scanner after integers.
- Solution: Used nextLine() to handle string inputs properly.
- Problem: Maven dependencies not downloading automatically.
- Solution: Reloaded Maven project and ran mvn clean install.
- Learning: Gained hands-on practice in CRUD implementation, user interaction, Maven setup, and GitHub usage.

4. Verification

- Verified Add Product by entering details and checking if the product is added to the list.

- Verified Delete Product by removing a product using its id and confirming it no longer appears in the list.
- Verified Update Product by changing quantity and price and ensuring updated values are displayed.
- Verified Search Product by entering a name and checking if the correct product details are displayed.

Task: Upgrade Inventory System Using HashMap and Implement Robust Exception Handling

Student Name: Shubhangi Bhuyare
Date Completed: 12-09-2025

---

1. Objective

The primary aim of this task was to enhance the performance and stability of the inventory management system by:

- Replacing `ArrayList<Product>` with `HashMap<Integer, Product>` to achieve faster product lookup, modification, and deletion using product IDs.
- Incorporating comprehensive exception handling with `try-catch` blocks to manage invalid user inputs and prevent unexpected program crashes.
- Improving the application's reliability and making it more user-friendly.

---

2. Steps Followed

1. Opened the existing inventory management project in IntelliJ IDEA. 2. Refactored the **InventoryManager** class located in the `com.inventory.service` package by: ○ Changing the product data structure from `ArrayList<Product>` to `HashMap<Integer, Product>`.

○ Updating all relevant methods (`addProduct`, `removeProduct`, `updateProduct`, `searchProduct`, `displayAll`) to work seamlessly with `HashMap`.

3. Introduced `try-catch` blocks in input sections to handle invalid or incorrect user entries.
4. Added logic to prevent the addition of duplicate product IDs.
5. Implemented search functionality using `products.values()` to iterate through all product entries.
6. Conducted multiple test runs to confirm that exception handling worked correctly and that meaningful error messages were displayed.

---

3. Challenges Faced

- Initially encountered difficulties while handling invalid inputs such as alphabets instead of numbers; resolved by implementing proper exception handling using `try-catch`.

---

4. Verification

- **Add Product:** Confirmed that product details were successfully stored in the `HashMap`.
- **Delete Product:** Verified that products were removed correctly by ID.
- **Update Product:** Checked that changes to quantity and price were reflected accurately.
- **Search Product:** Validated that searching by name returned correct results.
- **Exception Handling:** Tested with invalid inputs and ensured the system showed appropriate messages without crashing.

Task: Integrate JDBC for Database Connectivity and Perform CRUD Operations

Student Name: Shubhangi Bhuyare
Date Completed: 14-09-2025

---

1. Objective

The objective of this task was to connect the inventory management system to a MySQL database using JDBC, enabling it to perform CRUD (Create, Read, Update, Delete) operations directly on the database for persistent data storage.

---

2. Steps Followed

1. Verified that MySQL was installed and created a database named `inventoryDB` with a table `products` having fields: `id`, `name`, `category`, `quantity`, and `price`.
2. Created a **dbConnection** class to establish a JDBC connection with the MySQL database using JDK 17.
3. Implemented a **ProductDAO** class to manage all database-related operations:
    - `addProduct(Product p)` → Inserts a new record into the database.
    - `getAllProducts()` → Retrieves all products from the database.
    - `getProductByName(String name)` → Fetches product details by name.
    - `updateProduct(int id, int qty, double price)` → Updates the quantity and price of a specific product.
    - `removeProduct(int id)` → Deletes a product based on its ID.
4. Updated the **InventoryManager** class to use DAO methods instead of inmemory data structures like `ArrayList` or `HashMap`.
5. Modified the **Main** class to provide a console-based menu with the following options: ○ Add Product ○ Remove Product ○ Update Product ○ Search Product by Name ○ Display All Products ○ Exit
6. Added exception handling to manage invalid user inputs and ensure smooth program execution.
7. Tested all CRUD functionalities to ensure proper interaction with the MySQL database.

---

3. Challenges Faced

- Faced issues while configuring JDBC connection parameters (URL, username, password), which were fixed after reviewing settings.

- Encountered SQL errors due to data type mismatches, resolved by ensuring correct parameter mapping in `PreparedStatement`.

---

4. Verification

- **Add Product:** Confirmed successful insertion of new products into the `products` table.
- **Remove Product:** Verified that records were deleted correctly using their IDs.
- **Update Product:** Ensured updated quantity and price values appeared accurately in MySQL.
- **Search Product:** Confirmed that searching by name returned accurate product details.
- **Display All Products:** Checked that all product records were displayed properly from the database.
- **Exception Handling:** Entered invalid inputs to verify that the program displayed clear error messages without terminating.

Task: Develop SQL Query Scripts for Product Data Management

Student Name: Shubhangi Bhuyare
Date Completed: 15-09-2025

---

1. Objective

The purpose of this task was to design and execute a SQL script file (`products_queries.sql`) containing essential queries for managing and analyzing product information in the `products` table of the `inventoryDB` database. The goal was to retrieve insights such as product listings, filtering based on conditions, sorting, and calculating summaries.

---

2. Steps Followed

1. Opened the `inventoryDB` database in MySQL.
2. Verified that the `products` table was present with the columns: `id`, `name`, `category`, `quantity`, and `price`.
3. Created a new SQL script file named **products_queries.sql**.
4. Added the following queries to the file:
    - Display all products from the inventory. ○ Show only product names and categories. ○ Retrieve products with quantity greater than 10. ○ Retrieve products priced below 5000. ○ List all products belonging to the “Electronics” category.
    - Sort products by price in descending order. ○ Show the top 3 most expensive products.
    - Calculate the total number of products (sum of quantity). ○ Find the average price of all products.
    - Retrieve the highest-priced product.
5. Executed the SQL script using the command:
6. `mysql -u username -p inventoryDB < products_queries.sql`

---

3. Challenges Faced

- Initially found it challenging to recall the exact SQL syntax for sorting and limiting query results, but this was resolved by referring to MySQL documentation.

---

4. Verification

- Executed all queries and reviewed results in the MySQL console.
- Confirmed the following outcomes:
    - All product records were displayed successfully. ○ Product names and categories appeared correctly. ○ Filtering by price and quantity worked as intended.
    - Products were sorted accurately by price (highest first).
    - Top 3 expensive items were shown correctly. ○ Aggregate functions (`SUM`, `AVG`, `MAX`) produced accurate results.

Task: Implement Custom Exceptions and Explore Java File Handling

Student Name: Shubhangi Bhuyare
Date Completed: 17-09-2025

---

1. Objective

The aim of this task was to enhance error handling in the inventory management system and explore Java file operations by:

- Creating a custom exception named `ProductNotFoundException` for handling missing product cases.
- Modifying DAO and Manager classes to throw and handle this exception appropriately.
- Learning Java I/O (Input/Output) operations for potential use in generating reports or logs.

---

2. Steps Followed

1. **Created a New Exception Package:**
   - Added a package `com.inventory.exception`. ○ Created a custom exception class `ProductNotFoundException` extending `RuntimeException`.
2. **Modified DAO Layer:**
   - Updated methods such as `removeProduct()`, `updateProduct()`, and `getProductByName()` in `ProductDAO` to throw `ProductNotFoundException` when a product doesn't exist.
3. **Updated Service Layer:**
   - In `InventoryManager`, added `try-catch` blocks to handle the custom exception and display user-friendly messages.
4. **Explored Java I/O Concepts:**
   - Studied file handling using classes such as `File`, `FileReader`, `FileWriter`, `BufferedReader`, and `BufferedWriter`.
   - Learned basic operations like creating, reading, and writing files along with handling `IOException` using `try-with-resources`.

3. Challenges Faced

- Needed a clear understanding of the difference between checked and unchecked exceptions before deciding to extend `RuntimeException`.
- Java File I/O was a new concept, which required additional practice to grasp reading and writing mechanisms properly.

---

4. Verification

- Tested the custom exception by performing remove, update, and search operations on non-existent products — verified that the system displayed custom error messages.
- Ensured that valid product operations still worked correctly.
- Confirmed understanding of Java I/O by writing small demo programs that successfully read from and wrote to text files.

Task: Introduce CSV-Based Save and Load Functionality in Inventory System

Student Name: Shubhangi Bhuyare
Date Completed: 18-09-2025

---

1. Objective

The goal of this task was to enhance the inventory management system with filebased persistence using CSV (Comma-Separated Values) files. This allowed saving and reloading product data easily. Specific objectives included:

- Implementing a `CSVHelper` utility class for reading and writing product data.
- Enabling CSV loading within the `InventoryManager` class.
- Adding a new menu option in the main program to load data from CSV files.
  - Resolving static and non-static method access issues in CSV operations.

---

2. Steps Followed

1. **Enhanced Utility Layer:**
    - Created a new class `CSVHelper` under the `com.inventory.util` package.
    - Implemented `saveProduct(Product p)` to append product details into a CSV file.
    - Implemented `loadProducts()` to read product data from the file and return it to the system.
2. **Modified InventoryManager:**
    - Added `loadProductsFromCSV()` to invoke `CSVHelper.loadProducts()` and display the loaded products.
3. **Updated Main Class:**
    - Extended the user menu to include the "Load Products from CSV" option.
    - Adjusted `switch` statements for smooth menu navigation.
4. **Resolved Static Reference Issue:**
    - Fixed the "Non-static method cannot be referenced from a static context" error by creating a `csvDao` object instead of using static method calls.
5. **Tested Functionality:**
    - Verified that saving a product correctly writes details into `products.csv`.
    - Confirmed that loading from CSV reads data accurately and displays it in the console.

---

3. Challenges Faced

- Encountered the "Non-static method cannot be referenced from static context" error when calling methods incorrectly.
- Needed to simplify CSV logic to avoid unnecessary complexity such as header management or overwrite modes.

---

4. Verification

- **Add Product:** Successfully saved new product details to `products.csv`.
- **Load from CSV:** Verified that product details loaded correctly from the file.
- **Menu Update:** Ensured that the new option appeared and worked as expected.
- **Error Fix:** Confirmed that static vs non-static issues were resolved and the program compiled without errors.

Task: Add Search Submenu (Search by ID, Search by Name, View All Products)
Student Name: Shubhangi Bhuyare
Date Completed: 23-09-2025

1. Objective

The objective of this task was to improve the usability of the Inventory Management System by:

- Extending the Main menu to include a sub-menu for search operations.
- Allowing the user to choose between:
    1. Search product by ID
    2. Search product by Name
    3. View all products
    4. Back to main menu
- Adding a getProductById() method in the DAO layer.
- Updating InventoryManager with searchProductById() and searchProductByName() methods.

2. Steps Taken

1. **Modified Main class:**
    - Replaced the single searchProduct() call with a new search sub-menu loop.
    - Added options for searching by ID, name, or viewing all products.
2. **Updated InventoryManager class:**
    - Added searchProductById() method to handle search by product ID. ○ Renamed/refactored existing search method into searchProductByName().

3. **Updated ProductDAO class:**
   - Added `getProductById(int id)` method to query the database for a product by its ID.
   - Retained `getProductByName(String name)` for name-based searches.
4. Ensured proper exception handling with `ProductNotFoundException`.
5. Tested all new search options by adding sample data and running the menu.

3. Challenges Encountered

- Needed to restructure the Main menu logic to allow smooth navigation between main menu and sub-menu.
- Ensuring exception handling was consistent between ID-based and namebased searches.

4. Verification

- Verified Search by ID → Entered an existing product ID; details were displayed. Invalid ID showed `ProductNotFoundException`.
- Verified Search by Name → Entered an existing product name; details were displayed. Invalid name showed error message.
- Verified View All Products → Displayed all records from the database.
- Verified Back option → Returned to main menu without exiting the program.

Task: Add Search by Category & JUnit Dependencies in Inventory System
Student Name: Shubhangi Bhuyare
Date Completed: 29-09-2025

1. Objective

The objective of this task was to extend the inventory management system by:

- Adding a search by category functionality to allow users to filter products based on their category.

- Integrating JUnit dependencies into the project to support automated unit testing of core functionalities (CRUD operations and search features). **2. Steps Taken**

1. **Updated Main Menu:**
   - Modified the Search Sub-Menu in Main.java to include an option for Search by Category.
2. **DAO Layer Update (ProductDAO):**
   - Implemented getProductsByCategory(String category) to fetch products from the database by category.
   - Ensured it throws ProductNotFoundException if no products are found.
3. **Service Layer Update (InventoryManager):**
   - Added a new method searchProductByCategory() that calls DAO and displays matching products.
4. **JUnit Setup:**
   - Added JUnit 5 dependency in pom.xml. ○ Verified IntelliJ recognized the JUnit library.

3. Challenges Encountered

- Initially forgot to update the search menu numbering after inserting the new Search by Category option.
- Faced dependency conflict when both JUnit 4 and JUnit 5 were present; resolved by using only JUnit 5 (junit-jupiter).

4. Verification

- **Search by Category:** Entered category "Electronics" and confirmed only matching products displayed. Invalid category showed ProductNotFoundException.
- **JUnit Setup:** Verified IntelliJ recognized JUnit 5.
- **Test Execution:** All unit tests (ProductDAOTest) passed successfully in IntelliJ's test runner.

Task: JUnit Tests for CRUD Operations

Student Name: Shubhangi Bhuyare
Date Completed: 01-10-2025

1. Objective

The objective of this task was to test CRUD operations (Create, Read, Update, Delete) on the products table using the ProductDAO class with JUnit. These tests ensure database integration and validate methods like addProduct, getProductById, updateProduct, removeProduct, and getProductsByCategory.

2. Tasks Completed

- Wrote JUnit test cases for:
  - **Create:** addProduct() ○ **Read:** getProductById(), getProductByName(), getProductsByCategory() ○ **Update:** updateProduct() ○ **Delete:** removeProduct()
- Added @Before setup to insert sample data before each test.
- Added @After cleanup to remove test data after execution. □ Verified exception handling with ProductNotFoundException.

3. Sample Code Snippets

```
@Test
public void testAddAndGetProduct() {
   Product p = new Product(101, "Mouse", 20, 450.0, "Electronics");
dao.addProduct(p);

   Product fetched = dao.getProductById(101);
   Assert.assertNotNull(fetched);
   Assert.assertEquals("Mouse", fetched.getName());
}

@Test(expected = ProductNotFoundException.class)
public void testRemoveNonExistingProduct() {
dao.removeProduct(999); // should throw exception }
```

4. Output Screenshot (JUnit)

✓ All CRUD tests passed successfully in IntelliJ JUnit runner.

5. Learning Outcome

- Learned how to test DAO classes with JUnit.
- Gained hands-on practice with database CRUD validation. □ Understood how to test exception scenarios in JUnit (@Test(expected=...)).
- Improved ability to write repeatable tests using setup and teardown (@Before, @After).

Task: Add Pagination Feature and Implement Mockito & Integration Test Cases in Inventory System
Student Name: Shubhangi Bhuyare
Date Completed: 02-10-2025

1. Objective

The objective of this task was to:

- Extend the inventory management system by adding a pagination feature for product listing.
- Ensure efficient database queries using SQL LIMIT and OFFSET.
- Write Mockito-based unit tests for DAO and service layer methods.
- Implement integration test cases to validate end-to-end functionality with the database.

2. Steps Taken

1. **Pagination Implementation:**
    - Updated ProductDAO with getProductsPaginated(int page, int pageSize) using SQL LIMIT and OFFSET.
    - Added displayPaginated() in InventoryManager to allow user input for page and size.
    - Updated Main menu with a new option: "View Products with Pagination".
2. **Mockito Unit Tests:**
    - Added test cases to mock DAO behavior using Mockito. ○ Verified service layer methods (e.g., add, update, search) without hitting the real database.

3. Integration Tests:

- Wrote integration tests to connect with MySQL and validate real CRUD + pagination queries.
- Used sample data setup and teardown to ensure database consistency across tests.

3. Challenges Encountered

- Initially faced `Table 'inventorydb.products' doesn't exist` error; resolved by creating the `products` table in MySQL.
- Mockito setup required adding correct dependencies (`mockito-core`) in the project.
- Had to carefully structure integration tests to avoid polluting the database with test data.

4. Verification

- **Pagination:** Verified by fetching products with `page=2`, `pageSize=3`, ensuring correct subset of data displayed.
- **Mockito Tests:** All service-level unit tests passed successfully with mocked DAO.
- **Integration Tests:** CRUD + pagination tests passed when connected to MySQL database.

---

5. Sample Code Snippets

DAO Pagination Method:

```
public List<Product> getProductsPaginated(int page, int pageSize) {
   List<Product> list = new ArrayList<>();
   String sql = "SELECT * FROM products LIMIT ? OFFSET ?";
try (Connection conn = dbConnection.getConnection();
PreparedStatement stmt = conn.prepareStatement(sql)) {
      int offset = (page - 1) * pageSize;
stmt.setInt(1, pageSize);       stmt.setInt(2,
offset);

      ResultSet rs = stmt.executeQuery();
```

```
        while (rs.next()) {
list.add(new Product(
rs.getInt("id"),
rs.getString("name"),
rs.getInt("quantity"),
rs.getDouble("price"),
rs.getString("category")
            ));
        }
    } catch (SQLException e) {
        System.out.println("Error fetching paginated products: " + e.getMessage());
    }
    return list; }
```

Mockito Test Example:

```
@Test
public void testSearchProductByName() {
    ProductDAO mockDao = Mockito.mock(ProductDAO.class);
    InventoryManager manager = new InventoryManager(mockDao);

    Product p = new Product(101, "Keyboard", 15, 850.0, "Electronics");
    Mockito.when(mockDao.getProductByName("Keyboard")).thenReturn(p);

    Product result = manager.searchProductByName("Keyboard");
assertNotNull(result);
    assertEquals("Keyboard", result.getName()); }
```

6. Learning Outcome

- Learned how to implement pagination using SQL LIMIT & OFFSET.
- Gained experience writing Mockito unit tests to mock DAO dependencies.
- Understood how to design integration tests that validate real database operations.
- Improved ability to balance between fast unit tests and realistic integration tests.

Task: Add Get Products by Price Range Functionality
Student Name: Shubhangi Bhuyare

Date Completed: 06-10-2025

1. Objective

The objective of this task was to extend the Inventory Management System by adding a new functionality that retrieves products within a specified price range. This improves search flexibility and allows filtering of products based on budget constraints.

2. Steps Taken

1. **DAO Layer Update (ProductDAO):**
    - Implemented a new method getProductsByPriceRange(double min, double max) that executes a SQL query with BETWEEN clause.
    - Used PreparedStatement to prevent SQL injection and handle dynamic parameters.

```
public List<Product> getProductsByPriceRange(double minPrice, double maxPrice) {
   List<Product> list = new ArrayList<>();
   String sql = "SELECT * FROM products WHERE price BETWEEN ? AND ?";
   try (Connection conn = dbConnection.getConnection();
PreparedStatement stmt = conn.prepareStatement(sql)) {
stmt.setDouble(1, minPrice);        stmt.setDouble(2,
maxPrice);        ResultSet rs = stmt.executeQuery();
       while (rs.next()) {
list.add(new Product(
rs.getInt("id"),
rs.getString("name"),
rs.getInt("quantity"),
rs.getDouble("price"),
              rs.getString("category")
          ));
       }
   } catch (SQLException e) {
      System.out.println("Error fetching products by price range: " + e.getMessage());
   }
   return list;
}
```

2. **Service Layer Update (InventoryManager):**

```
public void searchProductsByPriceRange(double min, double max) {
List<Product> products = dao.getProductsByPriceRange(min, max);     if
(products.isEmpty()) {
       System.out.println("No products found in this price range.");
   } else {
      products.forEach(System.out::println);
   }
}
```

3. **Main Menu Update (Main.java):**
    - Extended the menu with a new option: "Search Products by Price Range". ○ Took user input for minimum and maximum price, then called manager.searchProductsByPriceRange(min, max);.
4. **Testing:**
    - Inserted multiple products with different price ranges into the database. ○ Verified correct filtering when entering different ranges (e.g., 500– 2000, 2000–10000).

3. Challenges Encountered

- Initially faced Table 'inventorydb.products' doesn't exist error; resolved by ensuring table is created and populated.
- Needed to handle the case when no products matched the range → displayed a user-friendly message.

4. Verification

- Verified by entering various price ranges and confirmed that only products within the specified range were displayed.
- Verified system displays a clear message when no products are found.

Task: Implement Login Functionality

Student Name: Shubhangi Bhuyare
Date Completed: 08-10-2025

1. Objective

1. To implement a secure login mechanism allowing registered users and admins to access the Inventory Management System based on verified credentials.
2. To ensure only verified users can log in after successful email verification.

2. Steps Followed

- Added a new “Login” option in the main menu.
- Integrated credential validation using email and password from the database.
- Displayed color-coded console messages for success and failure feedback.
- Redirected successfully logged-in users to role-based dashboards.

3. Challenges Faced

- Managed incorrect login attempts and null object handling in DAO results.
- Ensured login was blocked for unverified accounts.

4. Verification

- Tested multiple login attempts with valid and invalid credentials.
- Confirmed redirection to correct dashboard based on user role.

5. Learning Outcome

- Understood authentication flow using DAO methods and secure credential validation.
- Learned structured handling of verified and unverified users during login.

Snippet Example (Login Validation)

```
User user = userDAO.login(email, password);
if (user != null && user.isVerified()) {
   System.out.println(GREEN + "Login successful!" + RESET);
   if (user.getRole().equals("admin")) showAdminMenu();
   else showUserMenu();
} else {
   System.out.println(RED + "Invalid credentials or unverified account." + RESET);
}
```

Task: Implement Signup Feature for User and Admin

Student Name: Shubhangi Bhuyare
Date Completed: 13-10-2025

1. Objective

1. To develop a role-based signup feature that registers users as either “Admin” or “User”.
2. To implement secure form validation and store hashed credentials in the database.

2. Steps Followed

- Added a “Signup” option to the main menu.
- Extended user table with a role column to differentiate admin and user accounts.
- Added validation for email, password, and contact number format.
- Stored details securely using password hashing.

3. Challenges Faced

- Resolved duplicate registration issues using unique email validation.
- Ensured proper role assignment and database consistency after signup.

4. Verification

- Successfully registered both admin and user accounts.
- Verified that roles were correctly assigned and stored in the database.

5. Learning Outcome

- Learned secure user registration practices with input validation.
- Understood role-based registration and login flow in Java applications.

Snippet Example (Password Hashing & Signup)

```
String hashedPassword = PasswordUtil.hashPassword(password);
userDAO.saveUser(new User(name, email, hashedPassword, role));
System.out.println(GREEN + "Signup successful! Please verify your email." + RESET);
```

Task: Provide Different Menu Options for User and Admin

Student Name: Shubhangi Bhuyare
Date Completed: 16-10-2025

1. Objective

1. To customize the system interface by providing separate menus for Admin and User roles.
2. To enhance user experience through role-based access control.

2. Steps Followed

- Created two separate menu structures for Admin and User.
- Linked menu options dynamically after login based on the user's role.
- Admin menu included options for product add, update, delete, and stock alerts.
- User menu provided limited operations like search and view products.

3. Challenges Faced

- Structured control flow to avoid overlap between menus.
- Ensured smooth navigation between user dashboards and main menu.

4. Verification

- Logged in with both Admin and User accounts and verified correct menu display.
- Confirmed restricted access for non-admin users.

5. Learning Outcome

- Learned implementation of role-based UI and access management.
- Improved understanding of logical flow control within console-based systems.

Task: Implement OTP Email Verification during Signup

Student Name: Shubhangi Bhuyare
Date Completed: 21-10-2025

1. Objective

1. To add OTP-based email verification ensuring user authenticity before login.
2. To maintain a secure registration system integrated with real-time email verification.

2. Steps Followed

- Integrated OTP generation and email sending through EmailUtil.
- Added a separate "Verify Email" option in the main menu.
- Introduced isVerified column in the user table for tracking verification status.

- Restricted login for unverified users until successful OTP validation.

3. Challenges Faced

- Managed real-time OTP validation and ensured time-bound usage.
- Prevented OTP reuse and handled expired OTP cases.

4. Verification

- Tested signup → OTP sent → OTP entered → successful verification → login.
- Confirmed unverified users could not log in until OTP validation.

5. Learning Outcome

- Gained hands-on experience with Java Mail API for OTP handling.
- Understood implementation of secure account verification using dynamic OTPs.

(OTP Email Verification)

```
String otp = EmailUtil.generateOTP();
EmailUtil.sendEmail(userEmail, "Verification OTP", "Your OTP is: " + otp);
System.out.print("Enter OTP: ");
if (enteredOtp.equals(otp)) {
   userDAO.verifyEmail(userEmail);
   System.out.println(GREEN + "Email verified successfully!" + RESET);
}
```

Task: Implement Stock Alert Email Notification with Threshold Column

Student Name: Shubhangi Bhuyare
Date Completed: 24-10-2025

1. Objective

1. To automate stock-level monitoring by introducing a threshold alert system.
2. To notify admins through email when product quantity drops below a defined limit.

2. Steps Followed

- Added a new threshold column in the products table.
- Developed StockAlertService to check product quantity after each operation.
- Configured the system to send alerts only once per product to the logged-in admin.
- Integrated alert checks within InventoryManager after updates and purchases.

3. Challenges Faced

- Avoided duplicate email notifications for the same product shortage.
- Ensured that only admins received the alert mails, not users.

4. Verification

- Manually reduced stock levels to below threshold and verified alert emails.
- Confirmed single-time email sending per product shortage case.

5. Learning Outcome

- Learned how to automate notifications using threshold-based triggers.
- Understood integration of service classes for event-driven actions in Java.

🔑 (Stock Alert Trigger)

```
if (product.getQuantity() < product.getThreshold() && !alertSent) {
   EmailUtil.sendStockAlert(adminEmail, product);
   alertSent = true;
}
```

Task: Package and Deploy Full Inventory Management System

Student Name: Shubhangi Bhuyare
Date Completed: 28-10-2025

---

1. Objective

The objective of this task was to:

1. Package the entire Inventory Management System for deployment and easy execution on any machine.
2. Create a deployable .jar file using Maven build tools.
3. Ensure smooth configuration of database connections and environment setup.
4. Test the packaged system to confirm successful startup and functionality on a clean environment.

---

2. Steps Followed

1. Project Packaging using Maven:
    - Configured pom.xml with the maven-assembly-plugin to generate a runnable JAR file.

- Included all dependencies within the JAR using the “fat jar” approach.
- Executed the Maven command:
- mvn clean compile assembly:single

  to generate the final build under the /target directory.

2. Database Configuration:
   - Created a db.properties file containing database connection credentials (URL, username, password).
   - Modified DBConnection.java to read credentials dynamically from the configuration file.
3. Deployment Steps:
   - Copied the generated inventory-system.jar file and db.properties to the deployment machine.
   - Ensured MySQL service was running and the required tables were created.
   - Executed the JAR file using:
   - java -jar inventory-system.jar
   - Verified system launched successfully and menus worked as expected.
4. Post-Deployment Verification:
   - Checked all CRUD operations, stock alerts, and login/signup modules.
   - Verified email services and pagination features on the deployed environment.

---

3. Challenges Faced

- Initially faced dependency resolution issues during Maven build; resolved by cleaning local repository cache.
- Adjusted relative paths for reading configuration files after packaging.
- Handled ClassNotFoundException for JDBC driver by ensuring it was bundled inside the JAR.

---

4. Verification

- Successfully ran the application on a new machine without an IDE.
- All modules (CRUD, login/signup, email alerts, pagination) worked as intended.
- Verified database connectivity through console logs and functional test cases.

---

5. Learning Outcome

- Learned how to build and deploy a Java project using Maven.
- Understood the structure of executable JAR files and dependency management.
- Gained practical knowledge in configuring environment-independent applications.
- Experienced the end-to-end process of preparing a console-based system for real deployment.