PROJECT DOCUMENTATION

STORE MANAGER

1. Introduction

• Project Title: Store manager

• Team ID :NM2025TMID44646

• Team Leader: KAVISRI S(skavisri162007@gmail.com)

• Team Members:

NAME: ARUL P(Arul69304@gmail.com)

NAME: ARAVINDH N(Aravinthn10@gmail.com)

NAME:SANJAI S M (Sanjaisdsanjaisd8@gmail.com)

PROJECT OVERVIEW:

PURPOSE AND FEATURES

PURPOSE OF STORE MANAGER APP

INVENTORY CONTROL: Maintain up-to-date records of products (name, price, quantity, supplier, expiry, etc.).

SALES & ORDERS TRACKING: Keep track of customer purchases, invoices, and order history.

REAL-TIME UPDATES: Ensure any change in stock or price is instantly reflected across the system.

DATA CENTRALIZATION: Use MongoDB as a single source of truth for all store data.

SCALABILITY: Designed to handle large inventories, multiple branches, and future features.

⚙ KEY FEATURES (BACKEND: Node.js + MongoDB, JSON API)

FEATURE DESCRIPTION

Product Management Stock Monitoring Add, update, delete, and view product information. Automatic stock level checks and alerts when quantity drops below threshold. Category & Supplier Management Organize products by category; store supplier info. Sales / Orders Module Create invoices, track orders, and store customer info. JSON REST API frontend. All data exposed as JSON through secure REST endpoints for the User Authentication & Roles stock). Admin (add/edit products) vs. Staff (view/update Reports Generation Daily, weekly, monthly sales & stock reports. Search & Filter Search products by name, category, or supplier from the frontend. Audit Logs Track who made changes to inventory or pricing.

?? FRONTEND FEATURES (REACT / ANGULAR / Vue)

Dashboard showing total products, low stock alerts, and sales summaries.

Interactive Product Table with sorting, filtering, and inline editing.

Order Entry Form to quickly create invoices and reduce stock automatically.

Reports Page to view/download sales and inventory reports.

Responsive Design for desktop, tablet, and mobile.

Charts & Graphs (sales trends, top products) using libraries like Chart.js or Recharts.

🛠️ TYPICAL TECH STACK

Backend: Node.js + Express.js

Database: MongoDB (with Mongoose ODM)

Data Format: JSON for API communication

Frontend: React.js / Angular / Vue.js

3.ARCHITECTURE:

Core Concepts & Architecture (The Theory)

At its heart, this program demonstrates the interaction between an application and a database, following a simple client-server model.

* Object-Oriented Design: The program is built around the GroceryManager class. This is a core concept of Object-Oriented Programming (OOP), where we bundle data (the database connection) and the operations that can be performed on that data (addProduct, deleteProduct, etc.) into a single, logical unit called an object.

* Database Driver: The program uses the official MongoDB Java Driver. This driver is a library that acts as a translator, allowing the Java application (the client) to send commands and data to the MongoDB database (the server) in a language it understands.

•CRUD OPERATIONS: This is the central theory behind most database applications. CRUD is an acronym for the four basic functions of data management

• CREATE: Adding new data. Implemented in the addProduct() method.

•READ : Retrieving existing data. Implemented in viewAllProducts() and viewLowStockProducts().

•UPDATE : Modifying existing data. Implemented in the updateProduct() method.

• DELETE: Removing existing data. Implemented in the deleteProduct() method.

•BSON DOCUMENTS: MongoDB is a NoSQL database that stores data in a format called BSON (Binary JSON). In the Java code, a BSON document is represented by the org.bson.Document class. You can think of a Document as a flexible container for key-value pairs, much like a Map in Java. For example,new Document("name", "Milk") creates a piece of data where the key is "name" and the value is "Milk".

4. STEP INSTRUCTIONS: PREREQUISITES:

MONGODB

JSNODE NODE.JON

INSTALLATION STEPS:

⚙️ Code Explanation (Method by Method)

Let's break down how the Java code implements the concepts above. The GroceryManager Class

This class encapsulates the entire logic. When a new GroceryManager() is created, its constructor immediately establishes a connection to the MongoDB server using the provided connection string (uri) and gets a reference to the grocery_shop database and the products collection within it. A collection in MongoDB is like a table in a traditional SQL database. addProduct(String name, ...) - The "Create" Operation

This method takes product details as input, creates a new Document object, and uses .append() to add the key-value pairs (name, category, price, stock). The line productsCollection.insertOne(newProduct); then sends this document to the MongoDB server, which saves it to the products collection. viewAllProducts() & viewLowStockProducts(int threshold) - The "Read" Operations

•ViewAllProducts() uses productsCollection.find(). This command fetches all documents from the collection. The code then iterates through them and prints them as JSON strings.

• ViewLowStockProducts() demonstrates a filtered read. Instead of getting all products, it first builds a filter: Bson filter = Filters.lt("stock", threshold);. This creates a query that means "find all documents where the value of the 'stock' field is less than the threshold." This filtering happens on the database server, which is highly efficient. updateProduct(String productName, ...) - The "Update" Operation

THIS METHOD IS A TWO-STEP PROCESS:

* FIND: It first defines which document to update using a filter: Bson filter = Filters.eq("name", productName);. This means "find the document where the 'name' is equal to the given product name."

* MODIFY: It then defines what to change using Bson updates = Updates.combine(...). The Updates.set(...) command tells MongoDB to set the price and stock fields to new values. The productsCollection.updateOne(filter, updates); command sends both the "find" and "modify" instructions to modify

Similar to updating, this method first uses a filter (Filters.eq("name", productName)) to find the specific document to be removed. The line productsCollection.deleteOne(filter); then commands the

The main Method

This static method serves as the entry point and a test script for the GroceryManager class. It demonstrates the class in action by performing a clear, sequential set of

operations:

* It creates an instance of GroceryManager, establishing the database connection.

* It calls drop() to clear any old data, ensuring a clean start.

* It creates five sample products.

* It reads and displays all products.

* It performs a filtered read to find low-stock items.

* It updates the details for "Bread".

* It reads all products again to show the update.

* It deletes the "Milk" product.

* It reads the remaining products to show the deletion.

* Finally, it closes the database connection.

OUTPUT ANALYSIS

The program's output directly reflects the sequence of operations in the main method.

* Initial Additions: The first five lines of output are simple confirmations from the addProduct method. Product added: Milk ...

PRODUCT ADDED: masalas

* FIRST VIEWABLE IDIOTS():

The program then prints all five products that were just added. The _id and $oid are unique identifiers automatically generated by MongoDB for every document. This is why their values will be different each time you run the code.

--- All Products ---

{ "_id" : { "$oid" : "" }, "name" : "Milk", ... }

{ "_id" : { "$oid" : "" }, "name" : "Bread", ... } ...

* VIEWLOWSTOCKPRODUCTS(10):

This section shows the result of the filtered read. Only "Apples" (stock: 5) and "masalas" (stock: 9) have a stock level less than 10, so they are the only ones displayed.

--- Products with Low Stock (< 10) ---

{ "_id" : { "$oid" : "" }, "name" : "Apples", "stock" : 5 }

{ "_id" : { "$oid" : "" }, "name" : "masalas", "stock" : 9 }

* UPDATE AND SECOND VIEW:

After the confirmation Product updated: Bread, the program displays all products again. You can see that the "Bread" document now reflects the new price (1.75) and stock (15). All other documents are unchanged.

--- All Products ---

...

{ "_id" : { "$oid" : "" }, "name" : "Bread", "price" : 1.75, "stock" : 15 }

...

* DELETE AND FINAL VIEW:

Finally, after the confirmation Product deleted: Milk, the program displays the inventory one last time. The "Milk" document is now gone, and only four products remain.

--- All Products ---

{ "_id" : { "$oid" : "" }, "name" : "Bread", ... }

{ "_id" : { "$oid" : "" }, "name" : "Apples", ... }

...

PROGRAM OVERVIEW

This Java program creates a simple web server using com.sun.net.httpserver. It connects to a MongoDB database to manage two collections: products and sales.

Think of it like a small-scale, internal web application for a grocery store. It has the following key functionalities:

DATABASE MANAGEMENT: Connects to a MongoDB instance to store and retrieve product and sales information.

API ENDPOINTS: Exposes several web "endpoints" that a web browser or another application can call to get data.

/: Serves a simple HTML page with tables for products and sales.

/products: Returns a list of all products in the inventory as JSON data.

/sales: Returns a list of all sales transactions as JSON data.

/SEED: A special endpoint to randomly generate new product and sales data and save it to the database, wiping out any existing data.

Front-end Interface: The / endpoint provides a basic web page that automatically fetches data from the /products and /sales endpoints and displays it in organized tables. It even includes a "Seed New Random Data" button to trigger the /seed endpoint.

Expected Program Output and Explanation

When you run this Java program, you will see a series of messages in your console (like the command prompt or terminal) and can interact with the program by opening a web browser.

1. CONSOLE OUTPUT This is the first thing you'll see after executing the GroceryServer.java file.

CONNECTED SUCCESSFULLY TO MONGODB!: This message confirms that the program successfully established a connection with your local MongoDB database. If you don't have MongoDB running on your machine, this step will fail and the program will throw an error.

⚠ Server started. Open your browser to http://localhost:8080: This indicates the web server is up and running, listening for requests on port 8080 of your computer. You now know exactly where to go to view the program's output. ◆ ◆

2. WEB BROWSER OUTPUT (The User Interface)

When you navigate in your web browser, you will see a web page that looks like a simplified dashboard for a grocery store.

THE PAGE WILL BE DIVIDED INTO TWO MAIN SECTIONS:

Product Inventory Table: This table displays all the items available in the store. Since the database is initially empty, this table will be blank at first, showing an "Error loading data" message until you seed the database.

RECENT SALES TABLE: This table shows a history of all sales transactions. Similar to the products table, it will be empty initially.

Here's an example of what the tables would look like after clicking the "Seed New Random Data" BUTTON:

Product Inventory Table Example ◆ ◆

This table is populated with 15 unique, randomly generated products. EXPLANATION:

TIMESTAMP: The date and time the sale occurred. The program generates dates that are in the past (up to 30 days ago).

PRODUCT ID & NAME: The specific product that was sold.

QUANTITY: The number of units of that product sold in this transaction.

TOTAL SALE: The total revenue for that transaction, calculated by multiplying the product's selling price by the quantity sold.

3. Output after "Seeding" the Database

When you click the "Seed New Random Data" button, you will see a new message in your console, indicating that the program is actively writing to the database.

EXPLANATION:

Seeding database with new data...: This confirms that the program has received your request and is starting the process. It will first delete all existing data from the products and sales collections.

Successfully updated stock for 10 products.: This message indicates that after creating the new sales, the program went back and correctly subtracted the sold quantities from the stock count of the

corresponding products in the products collection. The number 10 will vary depending on how many unique products were included in the generated sales.

CONCLUSION

The Store Manager Inventory System built with JavaScript (Node.js) and MongoDB provides a simple, scalable, and real-time way to manage store operations. By using a JSON-based API and a non-relational database, the system can efficiently store and retrieve large amounts of product and sales data.

THIS APPROACH ALLOWS STORE OWNERS OR MANAGERS TO:

Keep accurate, up-to-date inventory records.

Track sales, orders, and stock levels seamlessly.

Generate reports for better decision-making.

Offer a secure and role-based user experience for staff and admins.

Because it's built on widely used technologies, the app is easy to extend with new features—such as dashboards, analytics, and mobile access—and can grow alongside the business. In short, it's a cost-effective and modern solution for keeping track of inventory and store operations.

THANK YOU