2) Real-time Order Management System with Node.js

REST API | Validation | Authentication | Mongo Db

Imagine you're part of a software development team at an e-commerce company. Your company is focusing on building an order management system (OMS) to handle customer orders, update inventory, and generate invoices. The system needs to provide real-time responses and interact with multiple backend services, including order validation, inventory management, and invoicing. In this scenario, you are required to create a set of APIs to handle order placement, order status updates, inventory searches, and reporting.

As a Node.js developer, you'll need to apply a variety of programming concepts such as working with request bodies, query parameters, path variables, sorting, searching, and relational comparisons to implement the API functionality efficiently.

You are required to create the following API endpoints for the Real-time Order Management System:

1. Placing an order.
2. Fetching order details.
3. Updating the order status.
4. Searching and sorting inventory.
5. Generating an order report based on various criteria.

For each of the APIs, you must handle path variables, query parameters, and HTTP headers effectively. You will also need to implement various algorithms like searching, sorting, and relational comparisons.

Steps to Solve

1. **Set up the Node.js project**:
    - Initialize a Node.js project using npm init.
    - Install necessary dependencies: Express.js, body-parser, and any other libraries you might need.
2. **Implement the following API endpoints**:

a) POST /orders

- **Description**: This API is used to place a new order. The order details are passed in the request body.
- **Input**: Request body with customer and order details.
- **Output**: Confirmation message with order ID and status.

b) GET /orders/:id

- **Description**: This API fetches the details of an order based on its ID.
- **Input**: Path variable id (order ID).
- **Output**: The order details (such as items, customer info, status).

c) PUT /orders/:id/status

- **Description**: This API updates the status of an order (e.g., "shipped", "delivered").
- **Input**: Path variable id (order ID) and request body with new status.
- **Output**: Confirmation message with the updated status.

d) GET /inventory

- **Description**: This API allows you to search for products in the inventory.
- **Input**: Query parameters for search (e.g., product name, category), sorting (e.g., by price, stock), and filtering (e.g., stock availability).
- **Output**: A list of products matching the search criteria, sorted based on the given parameters.

e) GET /orders/report

- **Description**: This API generates an order report with various filters (e.g., date range, order status).
- **Input**: Query parameters for filters such as start_date, end_date, status.
- **Output**: A summary of orders matching the filters.

3. **Considerations**:
   - Use path variables and query parameters effectively.
   - Implement sorting and searching algorithms in the inventory API.
   - Ensure that order statuses are updated based on relational comparisons (e.g., compare current and new status).
   - Apply proper HTTP headers for content-type and authorization, if required.

Sample Data: orders

```
{
  "order_id": 1001,
  "customer_id": 501,
  "items": [
    { "product_id": 302, "quantity": 2, "price": 20.0 },
    { "product_id": 411, "quantity": 1, "price": 15.5 }
  ],
  "total_amount": 55.5,
  "status": "pending"
}
```

Sample Data: Inventories

```
[
  { "product_id": 302, "name": "Wireless Mouse", "category": "Electronics", "price": 20.0, "stock": 120
},
  { "product_id": 411, "name": "Mechanical Keyboard", "category": "Electronics", "price": 45.0, "stock":
80 },
  { "product_id": 543, "name": "Gaming Headset", "category": "Accessories", "price": 30.0, "stock": 50 }
]
```

Sample Data: reports

```
[
  { "order_id": 1001, "customer_id": 501, "total_amount": 55.5, "status": "pending", "date": "2024-11-
20" },
  { "order_id": 1002, "customer_id": 502, "total_amount": 80.0, "status": "shipped", "date": "2024-11-
18" }
]
```

Sample Input and Output for Each API

1. POST /orders

Input:

```
{
  "customer_id": 501,
  "items": [
    { "product_id": 302, "quantity": 2, "price": 20.0 },
    { "product_id": 411, "quantity": 1, "price": 15.5 }
  ]
}
```

Output:

```
{
  "message": "Order placed successfully",
  "order_id": 1001,
  "status": "pending"
}
```

2. GET /orders/:id

Input: /orders/1001

Output:

```
{
  "order_id": 1001,
  "customer_id": 501,
  "items": [
    { "product_id": 302, "quantity": 2, "price": 20.0 },
    { "product_id": 411, "quantity": 1, "price": 15.5 }
  ],
  "total_amount": 55.5,
  "status": "pending"
}
```

3. PUT /orders/:id/status

Input:

```
{
  "status": "shipped"
}
```

Output:

```
{
  "message": "Order status updated successfully",
  "order_id": 1001,
  "status": "shipped"
}
```

4. GET /inventory?category=Electronics&sort=price

Input: /inventory?category=Electronics&sort=price

Output:

```
[
  { "product_id": 302, "name": "Wireless Mouse", "category": "Electronics", "price": 20.0, "stock": 120 },
  { "product_id": 411, "name": "Mechanical Keyboard", "category": "Electronics", "price": 45.0, "stock": 80 }
]
```

5. GET /orders/report?start_date=2024-11-01&end_date=2024-11-30&status=pending

Input: /orders/report?start_date=2024-11-01&end_date=2024-11-30&status=pending

Output:

```
[
  { "order_id": 1001, "customer_id": 501, "total_amount": 55.5, "status": "pending", "date": "2024-11-20" }
]
```

API Implementation Considerations

1. **Sorting**:
    - Implement sorting on the inventory endpoint based on query parameters. For example, if the sort=price is passed, the products should be sorted by price in ascending order.
2. **Searching**:
    - The inventory endpoint should support searching for products based on category or name. If a search term like category=Electronics is provided, only products in the electronics category should be returned.
3. **Relational Comparison**:
    - In the PUT /orders/:id/status API, compare the current status with the new status to ensure the update is valid (e.g., you can’t go from "shipped" back to "pending").