FAST National University of Computer and Emerging Sciences

Department of Computer Science, Peshawar Campus

(Spring 2025)

Instructor: Usama Musharaf

Project - Software Engineering

1. Application Overview:

The microservices-based application will simulate an **E-commerce Order Processing System** with the following services:

- **Front-End / User Service:** Provides a user friendly Interface
- **Auth Service** → Manages user authentication and profiles
- **Product Service** → Handles product listings and inventory
- **Order Service** → Manages orders and payments
- **Payment Service** → Handles transactions and payments
- **Shipping Service** → Manages shipping and tracking
- **Notification Service** → Sends notifications to users

Each service will be deployed in a **Docker container** and will interact with other services using **REST APIs.**

---

2. Technology Stack

Component	Technology
Frontend	Html, Css, Javascript / React
Backend	Node.js (Express) / Python (Flask/FastAPI) , PHP
Database	MongoDB / MYSQL or any other of your choice.
Containerization	Docker
Communication	REST APIs

3. System Architecture Overview

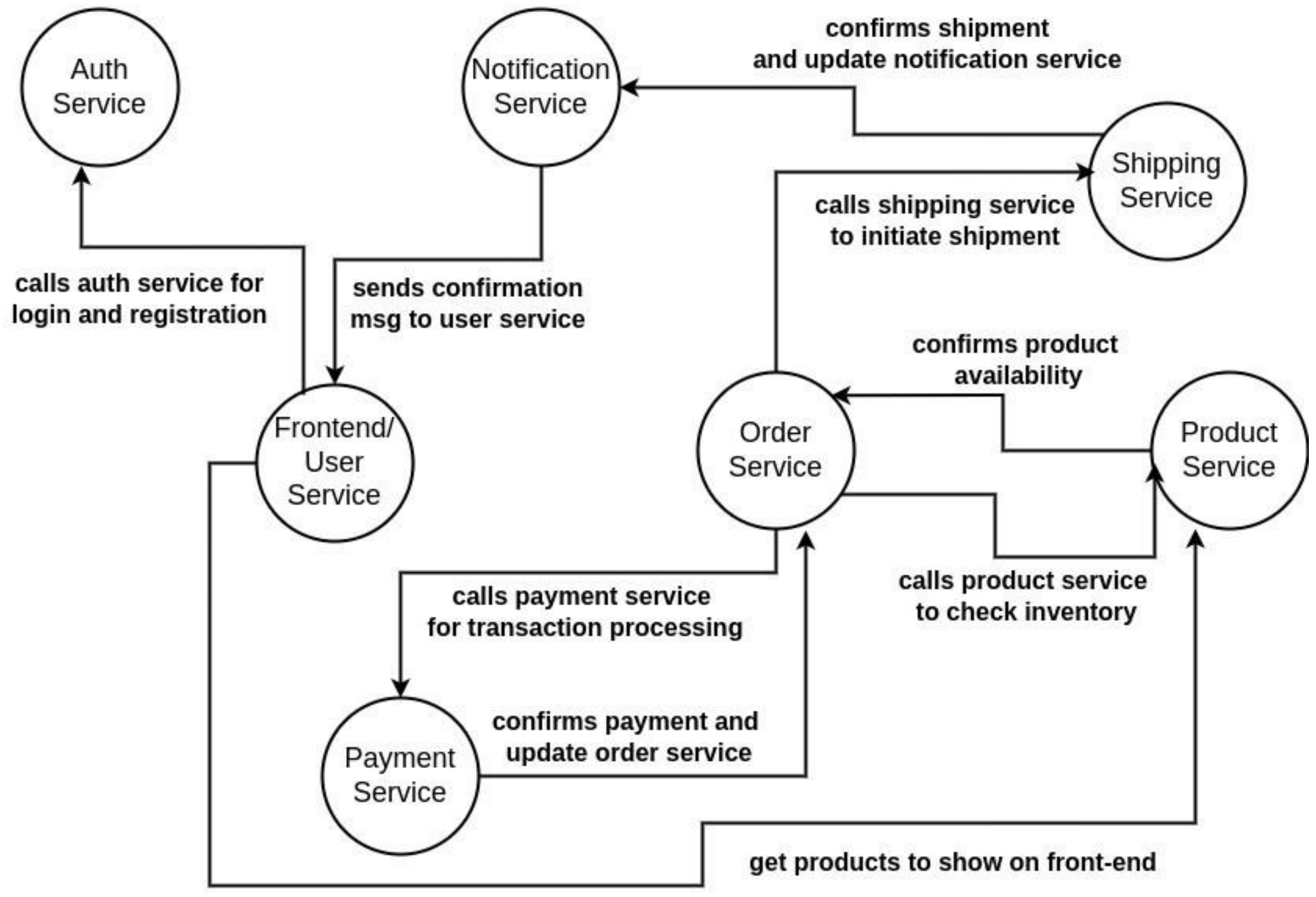
Service Dependency Flow:

1. **Front-End/ User Service:** User Friendly Interface
2. **Auth Service** → Handles user login and registration
3. **Product Service** → Fetches product details
4. **Order Service** → Takes orders from users, interacts with Product Service for inventory check
5. **Payment Service** → Processes payments for orders
6. **Shipping Service** → Manages order shipment after payment confirmation
7. **Notification Service** → Sends order confirmation and shipping status updates

Dependency Chain (Example Flow for a User Order):

1. **Auth Service** → Authenticates user
2. **Order Service** → Calls Product Service to check inventory
3. **Product Service** → Confirms product availability
4. **Order Service** → Calls Payment Service for transaction processing
5. **Payment Service** → Confirms payment and updates Order Service
6. **Order Service** → Calls Shipping Service to initiate shipment
7. **Shipping Service** → Confirms shipment and updates Notification Service
8. **Notification Service** → Sends confirmation msg to the user service

[Dependency Graph]


Auth Service
Notification Service
confirms shipment and update notification service
Shipping Service
calls shipping service to initiate shipment
calls auth service for login and registration
sends confirmation msg to user service
confirms product availability
Frontend/ User Service
Order Service
Product Service
calls product service to check inventory
calls payment service for transaction processing
Payment Service
confirms payment and update order service
get products to show on front-end

Teams:

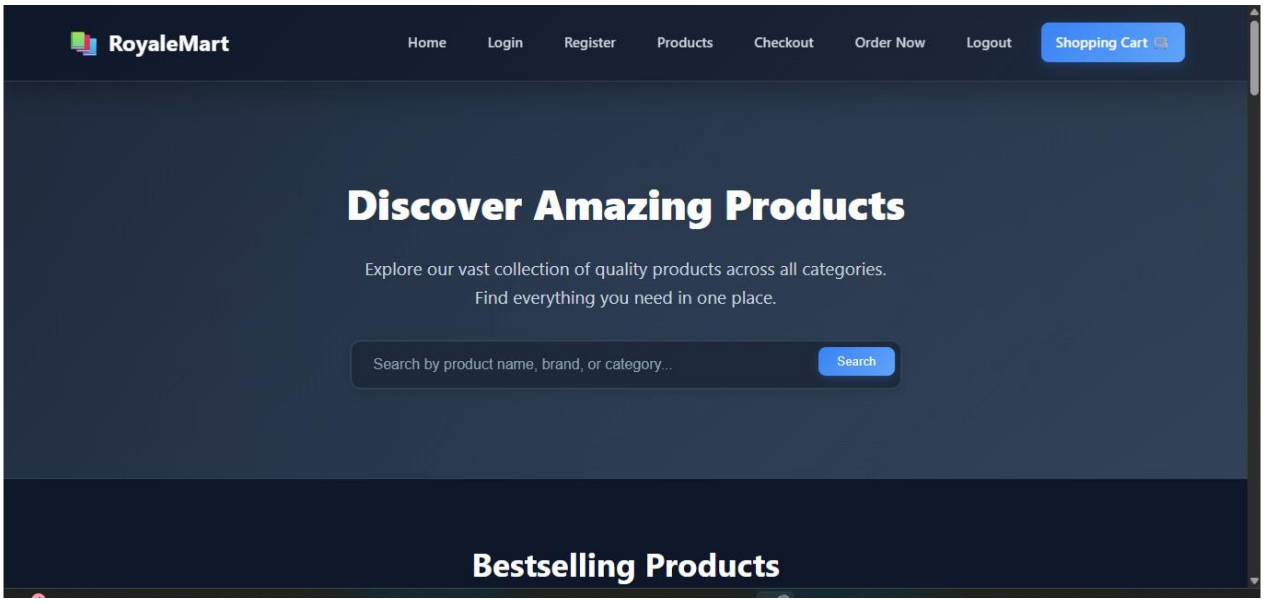
Team-A- Lead: **Moiz Ghazanfar Muhammad Nadeem**
Microservice: **Frontend-User Service**

#	Members
1	Faryal
2	Laiba
3	**Neelam**
4	**Subhan Shah Bukhari**

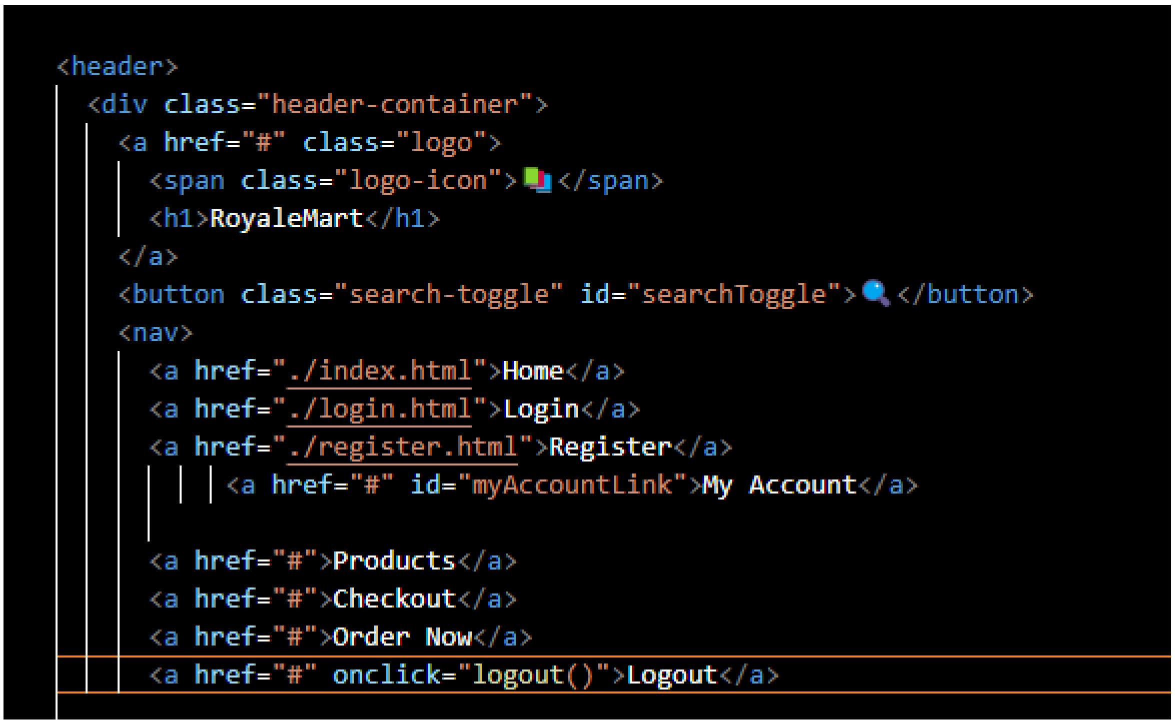
1. Frontend Development (HTML/CSS)

1.1 Home Page:

- Designed and implemented the responsive homepage layout with:
  - Header navigation bar with login/logout functionality
  - Product grid showcasing items with hover effects
  - Newsletter subscription section
  - Footer with multiple sections


RoyaleMart
Home
Login
Register
Products
Checkout
Order Now
Logout
Shopping Cart
Discover Amazing Products
Explore our vast collection of quality products across all categories.
Find everything you need in one place.
Search by product name, brand, or category...
Search
Bestselling Products

Key code snippet (header navigation):



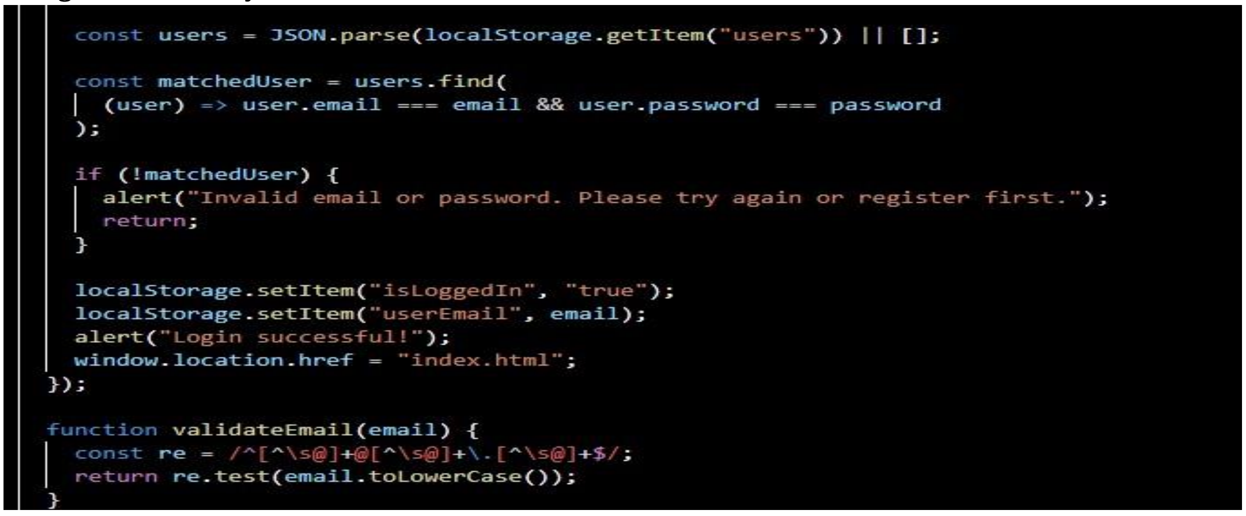
```html
<header>
  <div class="header-container">
    <a href="#" class="logo">
      <span class="logo-icon">📚</span>
      <h1>RoyaleMart</h1>
    </a>
    <button class="search-toggle" id="searchToggle">🔍</button>
    <nav>
      <a href="./index.html">Home</a>
      <a href="./login.html">Login</a>
      <a href="./register.html">Register</a>
          <a href="#" id="myAccountLink">My Account</a>

      <a href="#">Products</a>
      <a href="#">Checkout</a>
      <a href="#">Order Now</a>
      <a href="#" onclick="logout()">Logout</a>
```

1.2 User Authentication System

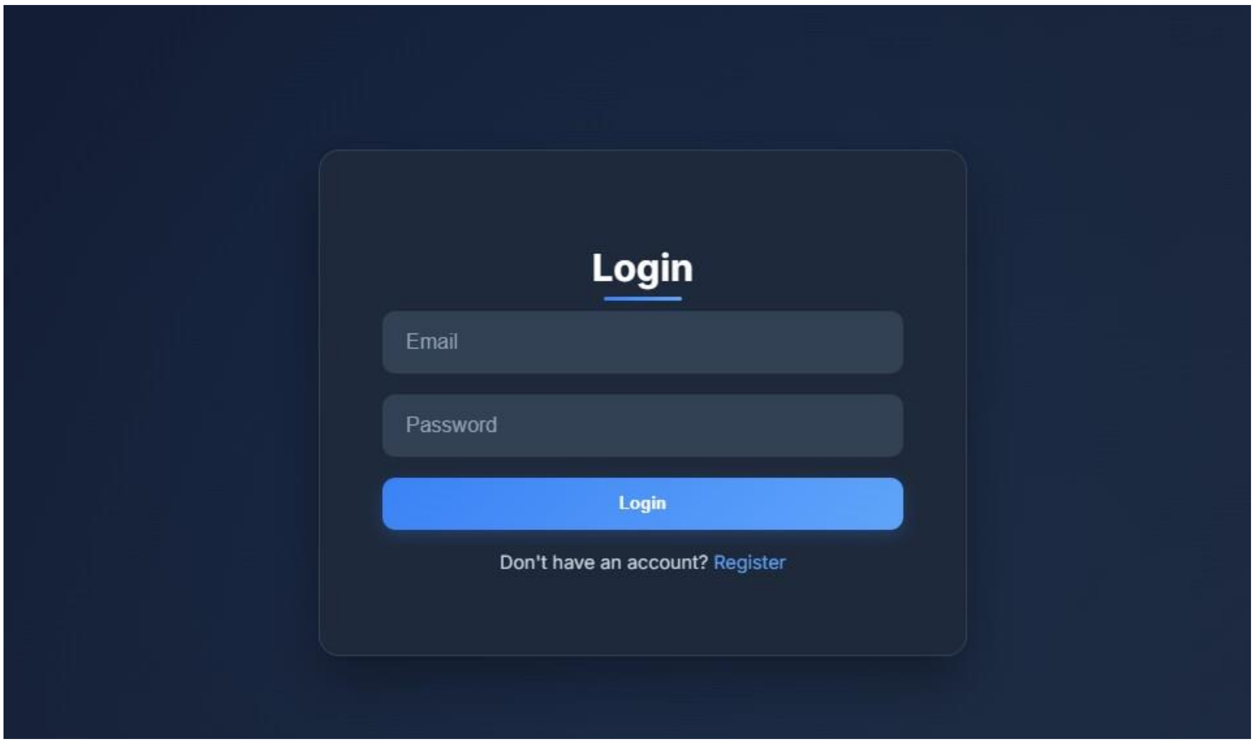
- Implemented login/registration functionality using localStorage
- Created form validation for email and password fields
- Added session management to track logged-in users

Key JavaScript code:

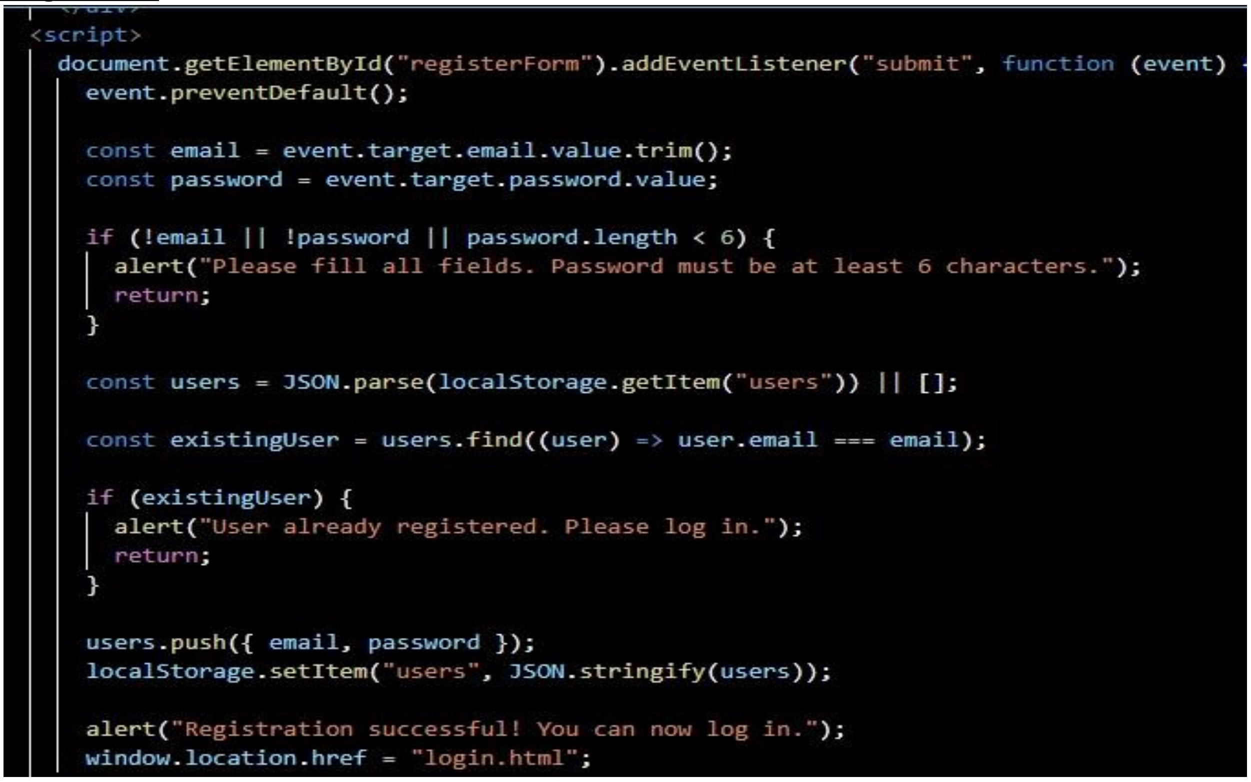
javascript
// Login functionality



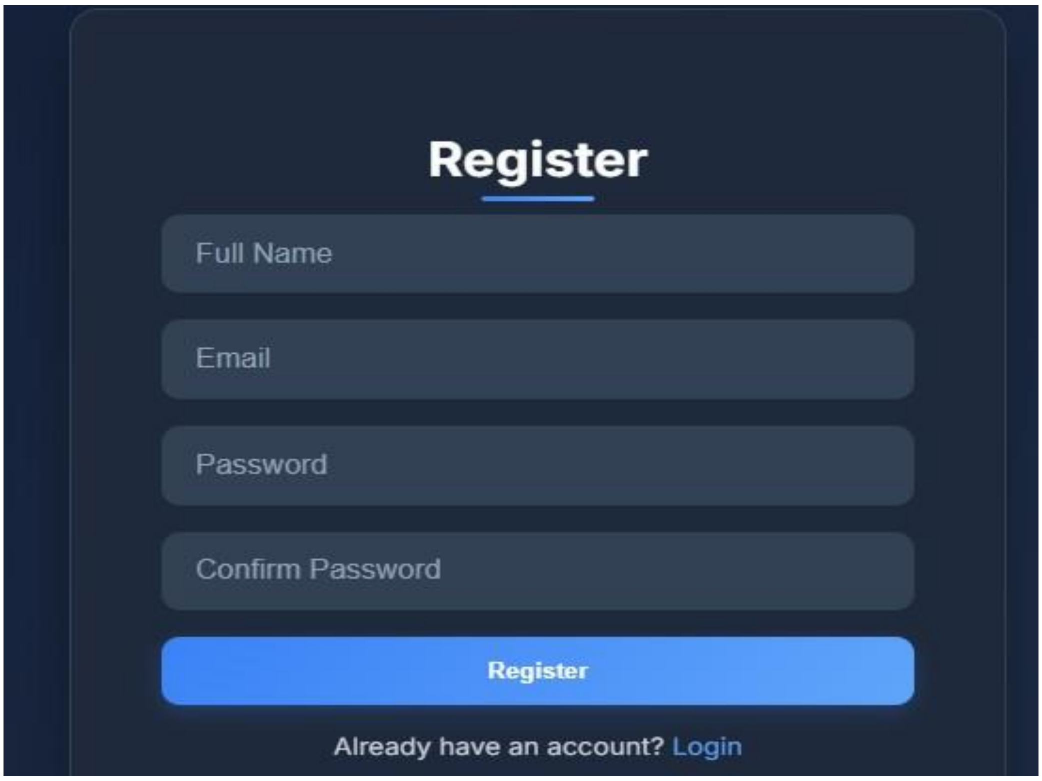
```javascript
    const users = JSON.parse(localStorage.getItem("users")) || [];

    const matchedUser = users.find(
      (user) => user.email === email && user.password === password
    );

    if (!matchedUser) {
      alert("Invalid email or password. Please try again or register first.");
      return;
    }

    localStorage.setItem("isLoggedIn", "true");
    localStorage.setItem("userEmail", email);
    alert("Login successful!");
    window.location.href = "index.html";
  });

  function validateEmail(email) {
    const re = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return re.test(email.toLowerCase());
  }
```



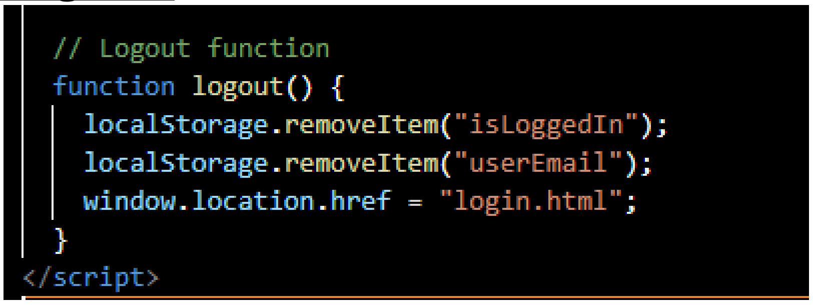
Registration



```
<script>
  document.getElementById("registerForm").addEventListener("submit", function (event)
    event.preventDefault();

    const email = event.target.email.value.trim();
    const password = event.target.password.value;

    if (!email || !password || password.length < 6) {
      alert("Please fill all fields. Password must be at least 6 characters.");
      return;
    }

    const users = JSON.parse(localStorage.getItem("users")) || [];

    const existingUser = users.find((user) => user.email === email);

    if (existingUser) {
      alert("User already registered. Please log in.");
      return;
    }

    users.push({ email, password });
    localStorage.setItem("users", JSON.stringify(users));

    alert("Registration successful! You can now log in.");
    window.location.href = "login.html";
```



LogOut:



```
    // Logout function
    function logout() {
      localStorage.removeItem("isLoggedIn");
      localStorage.removeItem("userEmail");
      window.location.href = "login.html";
    }
  </script>
```

1.3. Product Management

- Designed product card component with:
  - Image display
  - Pricing information (with discount badges)
  - Rating system
  - Add to cart functionality

Key CSS for product cards:

css

```
.book-card {
  background-color: var(--bg-
  secondary); border-radius: 16px;
  transition: transform 0.4s ease;
}

.book-card:hover {
  transform: translateY(-12px);
  border-color: var(--accent);}
```

Technical Implementation

Responsive Design
- Used CSS Grid and Flexbox for layout
- Implemented mobile-first approach with media queries
- Added hamburger menu for mobile navigation

Dark Theme
- Created comprehensive CSS variables for consistent theming
- Implemented gradient backgrounds and subtle shadows
- Added hover effects for better user interaction

Challenges & Solutions

1. **Challenge:** Maintaining user session across pages
   Solution: Used localStorage to track login status and user email

2. **Challenge:** Responsive product grid layout
   Solution: Implemented CSS Grid with auto-fill and minmax() for flexible columns

3. **Challenge:** Form validation
   Solution: Added JavaScript validation for email format and password matching

1.4. Product Page
- Product gallery with main image + thumbnails
- Detailed product info with price display
- Interactive quantity selector
- Add to cart functionality

```
html
<div class="product-container">
  <div class="product-gallery">...</div>
  <div class="product-details">
    <h1>Smart Watch Series 7</h1>
    <div class="price">$149.99 <span class="original-price">$199.99</span></div>
    <div class="quantity-selector">
      <button>-</button><input type="number" value="1"><button>+</button>
    </div>
    <button class="add-to-cart-btn">Add to Cart</button>
  </div>
</div>
```

1.5. Shopping Cart
- Dynamic cart item listing
- Editable quantities
- Real-time subtotal calculation
- Checkout progression

```
javascript
function updateCartTotal() {
  let total = 0;
  document.querySelectorAll('.cart-item').forEach(item => {
    total += parseFloat(item.dataset.price) * parseInt(item.querySelector('.qty-input').value);
  });
  document.querySelector('.subtotal').textContent = `Subtotal: $${total.toFixed(2)}`;
}
```

3. Checkout Page

- 3-step process (Shipping → Payment → Review)
- Form validation
- Order summary

1.6 Add / Edit Product

What is this?
This is a Product Management Form used to add, edit, or delete products in an inventory system.

How It Works

The user fills in the form fields: Product ID, Name, Stock, and Price.

On clicking "Save" or "Delete", JavaScript sends a REST API request to the backend.

The backend handles the request and updates the central database.

The user interface updates based on the response.

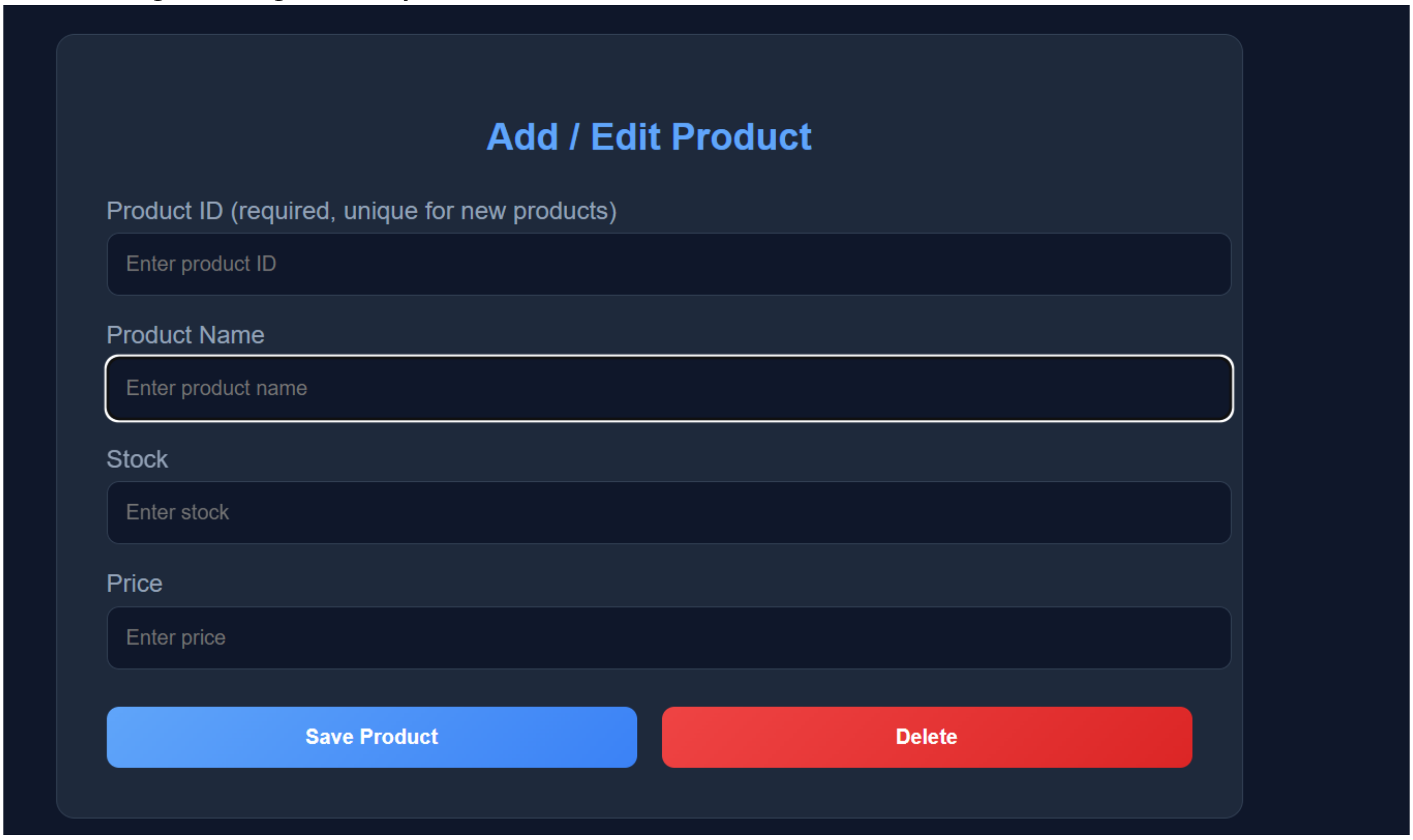
How They Made It

Frontend: Created using plain HTML, CSS, and JavaScript. It includes a simple form with input fields and buttons for saving or deleting a product.

Backend: A REST API (built separately) handles all product operations like add, update, and delete.

Database: A central database (such as MySQL or PostgreSQL) stores all product data.

Containerization: Docker is used to run the frontend, backend, and database in separate containers. Docker Compose is used to manage them together easily.



1.7 User Account:

What is this?
This is a User Account Dashboard that displays user profile information and their order history.

How It Works

When a user logs in, their profile and past orders are fetched using REST API requests. The backend retrieves this data from a central database.
The frontend uses JavaScript to dynamically show the profile and order details in a structured format.

Each order has a "Track" button that can be used to fetch tracking information.

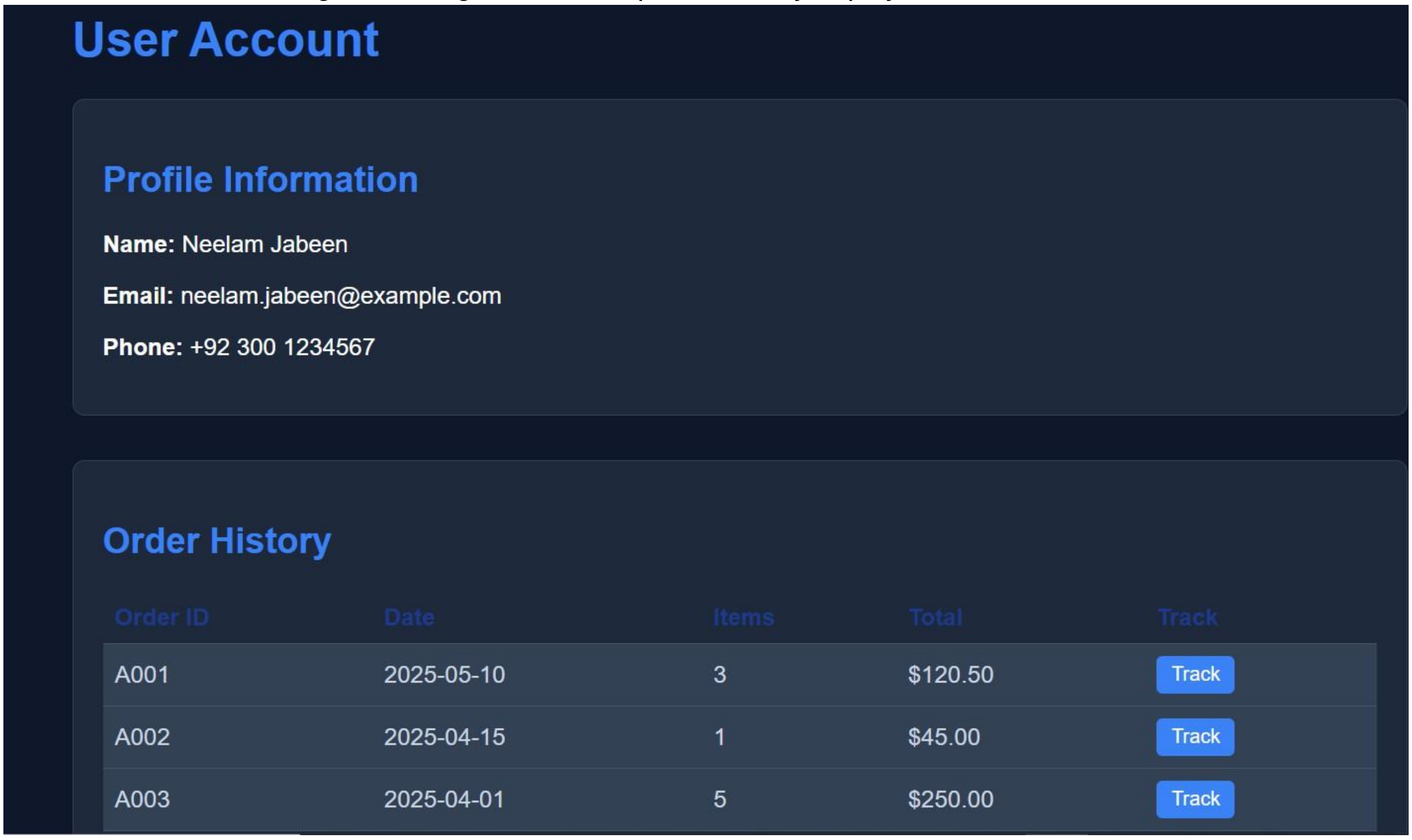
How They Made It

Frontend: Built using plain HTML, CSS, and JavaScript to design the layout and handle user interaction.

Backend: A REST API handles requests such as fetching user details and order history.

Database: A centralized database (like MySQL or PostgreSQL) stores user information and orders.

Containerization: The application components (frontend, backend, and database) are each placed in Docker containers, and run together using Docker Compose for easy deployment and communication.



User Account

Profile Information

Name: Neelam Jabeen

Email: neelam.jabeen@example.com

Phone: +92 300 1234567

Order History

Order ID	Date	Items	Total	Track
A001	2025-05-10	3	$120.50	Track
A002	2025-04-15	1	$45.00	Track
A003	2025-04-01	5	$250.00	Track

1.8 Checkout

What is this?
The Checkout Page allows users to review their cart, enter shipping and payment information, and place their order.

How It Works

The user fills out the form with shipping and payment details.

On clicking "Place Order", JavaScript sends a REST API request to the

backend.The backend verifies the data and saves the order in the central database.

A success response is returned, leading the user to the Order Confirmation Page.

How They Made It Frontend: Built using HTML, CSS, and JavaScript. JavaScript handles input validation and API communication.

Backend: A REST API endpoint receives the order data and processes it.

Database: A central database stores order details, including user info, product IDs, quantities, and total cost.

Containerization: All components (frontend, backend, and database) are run in separate Docker containers and managed via Docker Compose.

1.9 Order Confirmation Page:

What is this?
The Order Confirmation Page shows a summary of the successfully placed order.

How It Works

After placing an order, the backend returns an order ID and related details.

This data is shown to the user through the JavaScript-driven UI.

The confirmation page may also fetch live tracking or estimated delivery using the REST API.

How They Made It

Frontend: Developed using HTML, CSS, and JavaScript to display order details.

Backend: Responds to the order submission with confirmation data through a REST API.

Database: Order information is retrieved from the central database using the order ID.

Containerization: The backend, frontend, and database are containerized with Docker, enabling consistent deployment.

Team-B- Lead: **Saud Nasir & Abdullah**
Microservice: **Auth Service**

#	Members
1	Abdul Hadi
2	Talha Zia
3	**Asim Shakee;**
4	**Atta ur Rehman**

Overview

This document explains the structure and implementation of a user authentication microser- vice built using Flask and MongoDB. The microservice provides user registration and login functionality with secure password hashing and proper input validation.

Technologies Used
• Flask: A lightweight Python web framework used to build the REST API.
• MongoDB: A NoSQL database used to store user information.
• PyMongo: MongoDB client for Python.
• Werkzeug: Used for securely hashing and verifying passwords.
• UUID: Used to generate unique user IDs.

Project Structure
• userservice.py: Main Python script containing the API logic.
• /register route: Registers a new user.
• /login route: Authenticates existing users.

Implementation Details
MongoDB Setup
A MongoDB Atlas cluster was used to host the database. The connection string is provided in the code and connects to the UserService database and users collection.

```
client = MongoClient ( " mongodb + srv :// < username >: < password >@ < cluster - url > " )
 db = client [ " UserService " ]
 users_collection = db [ " users " ]
```

Listing 1: MongoDB Connection
User Registration
The /register endpoint handles new user registrations by validating input, checking for existing users, hashing the password, and inserting the user record into MongoDB.

```
@app . route ( '/ register ' , methods =[ ' POST ' ])
def register () :
data = request . get_json ()
hashed_password = g e n e r a t e _ p a s s w o r d _ h a s h ( password )
user_id = str ( uuid . uuid4 () )
users_collection . insert_one ({' _id ': user_id ,' username ': username ,' email ': email ,' password ':
hashed_password})
return jsonify ({  ' message ':  ' User registered successfully  ' }) , 201
```

Listing 2: Register Route
User Login
The /login endpoint authenticates users by either email or username. It verifies the password using check password hash and returns user info upon success.

```
@app . route ( '/ login ' , methods =[ ' POST ' ])
def login () :
data = request . get_json ()
if not user or not ch e c k_ p a s sw o r d_ h a sh ( user [ ' password '] , password ) :
return jsonify ({ ' message ': ' Invalid credentials ' }) , 401
return jsonify ({' message ': ' Login successful ' ,' user_id ': user [ ' _id '] ,' username ': user [ ' username ']
,' email ': user [ ' email ']}) , 200
```

Security Considerations
• Passwords are hashed using Werkzeug’s generate password hash function before storing them in the database.
• Input validation is performed to ensure required fields are present.
• MongoDB queries ensure uniqueness of both email and username.

Running the Microservice
The Flask app runs on port 5002:

```
if __name___== ' __main___':
app . run ( debug = True , port =5002)
```

Use tools like Postman or CURL to send POST requests to /register and /login endpoints for testing.

Future Improvements
• Implement JWT for session management and authentication.
• Add email verification for new users.
• Create user update and delete endpoints.
• Add rate limiting and input sanitization for better security.

Team-C- Lead: **Muhammad Abdullah and Rohail Nawaz**
Microservice: **Product Service**

#	Members
1	**Sultan Mehmood Mughal**
2	Hafiz Zarar
3	**Abdullah Bilal**
4	**Faran Ahmad**

Product Service:

Introduction

This report analyzes a product microservice built with FastAPI, MongoDB, and Pydantic, focusing on its functionality for product and inventory management, with an emphasis on admin-specific operations for adding, updating, and deleting products. The microservice integrates with a frontend service for product display and an order service for inventory checking, price, and quantity extraction. The report includes code snippets, highlights admin-specific features, and incorporates images of the frontend interface to illustrate the system.

Microservice Overview

The product microservice is a RESTful API developed using FastAPI, with MongoDB Atlas for data storage and Pydantic for data validation. It supports CRUD operations for products and inventory management, with CORS enabled for frontend integration. And product service is used by Order service for order processing task.

```
order-service:
build: ./Order_service
ports:
- "5000:5000"
environment:
MONGO_URI: mongodb://mongo:27017/OrderService
depends_on:
- mongo
- product-service
- shippingservice
```

Key Features

- • Product Management: Admin operations for creating, updating, and deleting products.
- • Inventory Management: Checking and updating stock levels for order processing.
- • Integration: Supports frontend display and order service integration.
- • Logging: Tracks operations for debugging and monitoring.

Code Analysis

The microservice provides endpoints for product and inventory operations, with specific endpoints for admin tasks. Below are key snippets for admin-related functionality.

Adding Products (Admin)

Admins can add new products via the POST /products endpoint, which validates input using the Product Pydantic model:

Listing 1: Add Product Endpoint (Admin)

```python
class Product(BaseModel):
    id: str
    name: str
    stock: int
    price: float

@app.post("/products")
async def add_product(product: Product):
    existing_product = products_collection.find_one({"id":
       product.id})
    if existing_product:
        logger.warning(f"Product ID {product.id} already exists")
        raise HTTPException(status_code=400, detail="Product ID
           already exists")
    products_collection.insert_one(product.dict())
    logger.info(f"Product added: {product.id}")
    return {"id": product.id, "message": "Product added"}
```

This endpoint ensures no duplicate product IDs and logs the operation. Admin access should be secured with authentication (e.g., JWT).

Updating Products (Admin)

Admins can update product details (name, stock, price) via the PUT /products/{product id} endpoint:

Listing 2: Update Product Endpoint (Admin)

Listing 2: Update Product Endpoint (Admin)

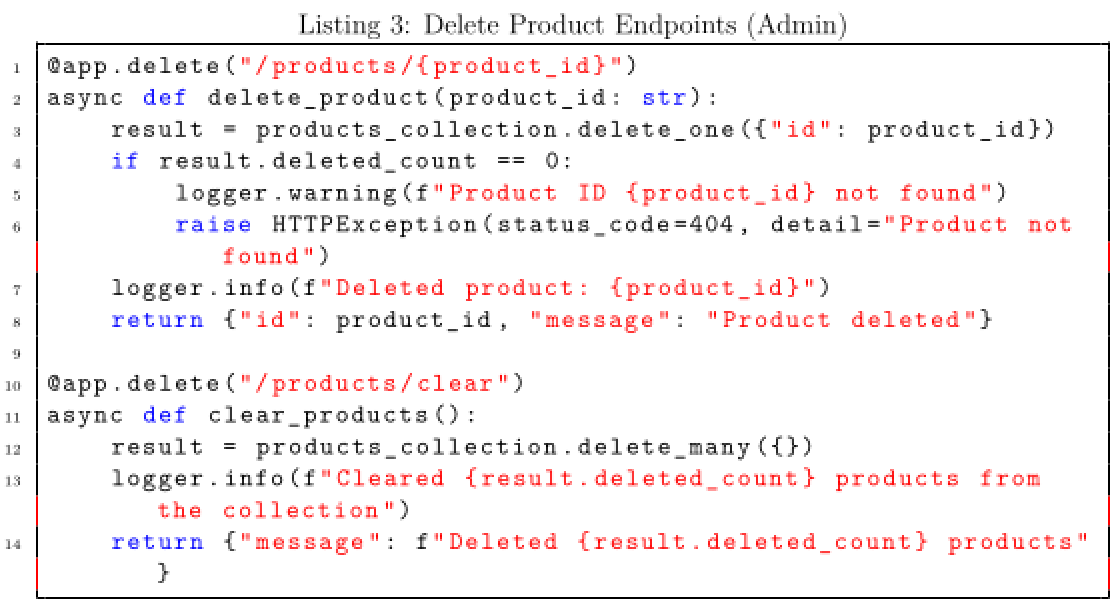
```python
@app.put("/products/{product_id}")
async def update_product(product_id: str, product: Product):
    existing_product = products_collection.find_one({"id":
       product_id})
    if not existing_product:
        logger.warning(f"Product ID {product_id} not found")
        raise HTTPException(status_code=404, detail="Product not
           found")
    update_data = {k: v for k, v in product.dict().items() if k
       in ["name", "stock", "price"]}
    result = products_collection.update_one({"id": product_id}, {
       "$set": update_data})
    if result.modified_count == 0:
        logger.info(f"No changes made to product {product_id}")
        return {"id": product_id, "message": "No changes detected
           "}
    logger.info(f"Product updated: {product_id}")
    return {"id": product_id, "message": "Product updated"}
```

This endpoint updates specified fields and logs changes. Authentication is needed to restrict this to admins.

Deleting Products (Admin)

Admins can delete individual products or clear all products using two endpoints:

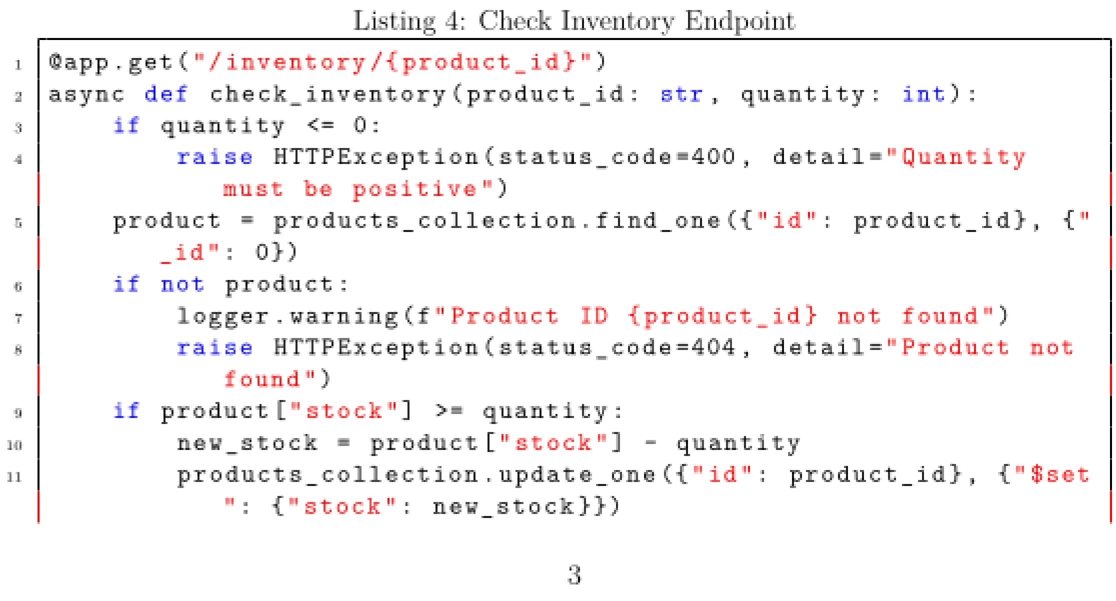
Listing 3: Delete Product Endpoints (Admin)


Listing 3: Delete Product Endpoints (Admin)

```
@app.delete("/products/{product_id}")
async def delete_product(product_id: str):
    result = products_collection.delete_one({"id": product_id})
    if result.deleted_count == 0:
        logger.warning(f"Product ID {product_id} not found")
        raise HTTPException(status_code=404, detail="Product not
           found")
    logger.info(f"Deleted product: {product_id}")
    return {"id": product_id, "message": "Product deleted"}

@app.delete("/products/clear")
async def clear_products():
    result = products_collection.delete_many({})
    logger.info(f"Cleared {result.deleted_count} products from
       the collection")
    return {"message": f"Deleted {result.deleted_count} products"
       }
```

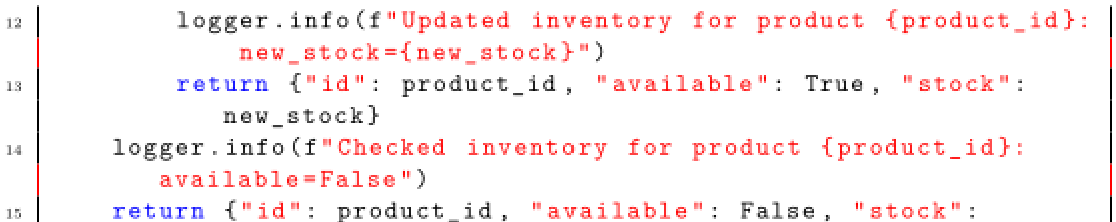
The DELETE /products/{product id} endpoint removes a single product, while DELETE /products/clear clears all products. Both should be protected with admin authentication.

Inventory Management
The GET /inventory/{product id} endpoint supports the order service by checking stock availability:
Listing 4: Check Inventory Endpoint


Listing 4: Check Inventory Endpoint

```
@app.get("/inventory/{product_id}")
async def check_inventory(product_id: str, quantity: int):
    if quantity <= 0:
        raise HTTPException(status_code=400, detail="Quantity
           must be positive")
    product = products_collection.find_one({"id": product_id}, {"
       _id": 0})
    if not product:
        logger.warning(f"Product ID {product_id} not found")
        raise HTTPException(status_code=404, detail="Product not
           found")
    if product["stock"] >= quantity:
        new_stock = product["stock"] - quantity
        products_collection.update_one({"id": product_id}, {"$set
           ": {"stock": new_stock}})
```

3



```
        logger.info(f"Updated inventory for product {product_id}:
            new_stock={new_stock}")
        return {"id": product_id, "available": True, "stock":
           new_stock}
    logger.info(f"Checked inventory for product {product_id}:
       available=False")
    return {"id": product_id, "available": False, "stock":
```

This endpoint ensures atomic stock updates and is used by the order service to validate inventory before order processing.

Integration with Frontend and Order Service

The microservice provides product data to the frontend via /products and inventory status via /inventory/{product id}. The order service uses the check inventory end-point to validate stock and extract price and stock for order processing. The frontend includes a product display page and an admin interface for managing products, as shown in the images below.

Frontend Product Page

The frontend product page displays a grid of products with details fetched from the /products endpoint. This is illustrated in Figure 1.

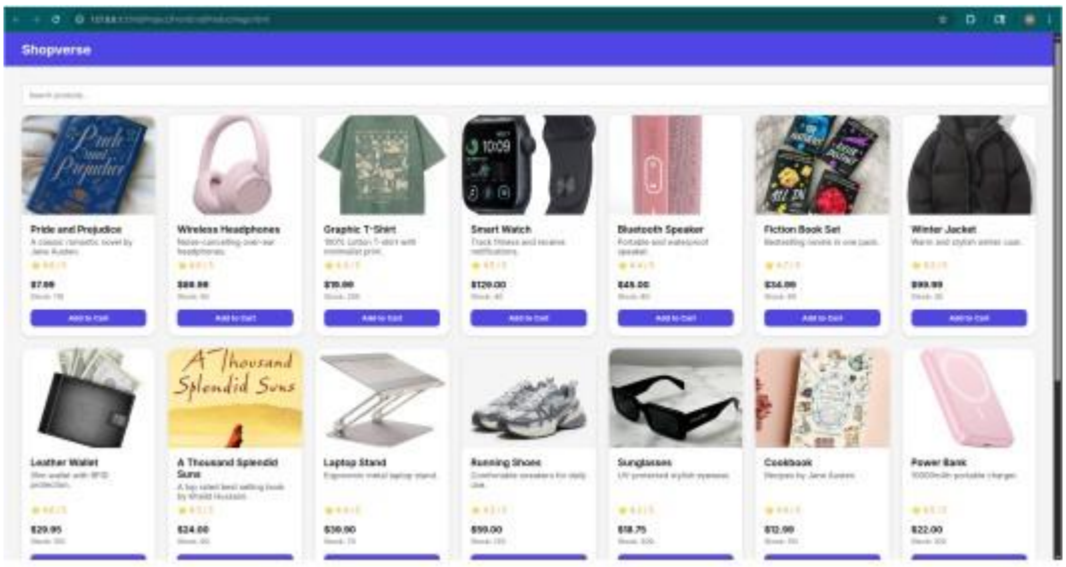


Figure 1: Frontend Product Page Displaying Available Products

Frontend Admin Page

The admin interface allows adding, updating, and deleting products, interacting with the respective microservice endpoints. This is shown in Figure 2.

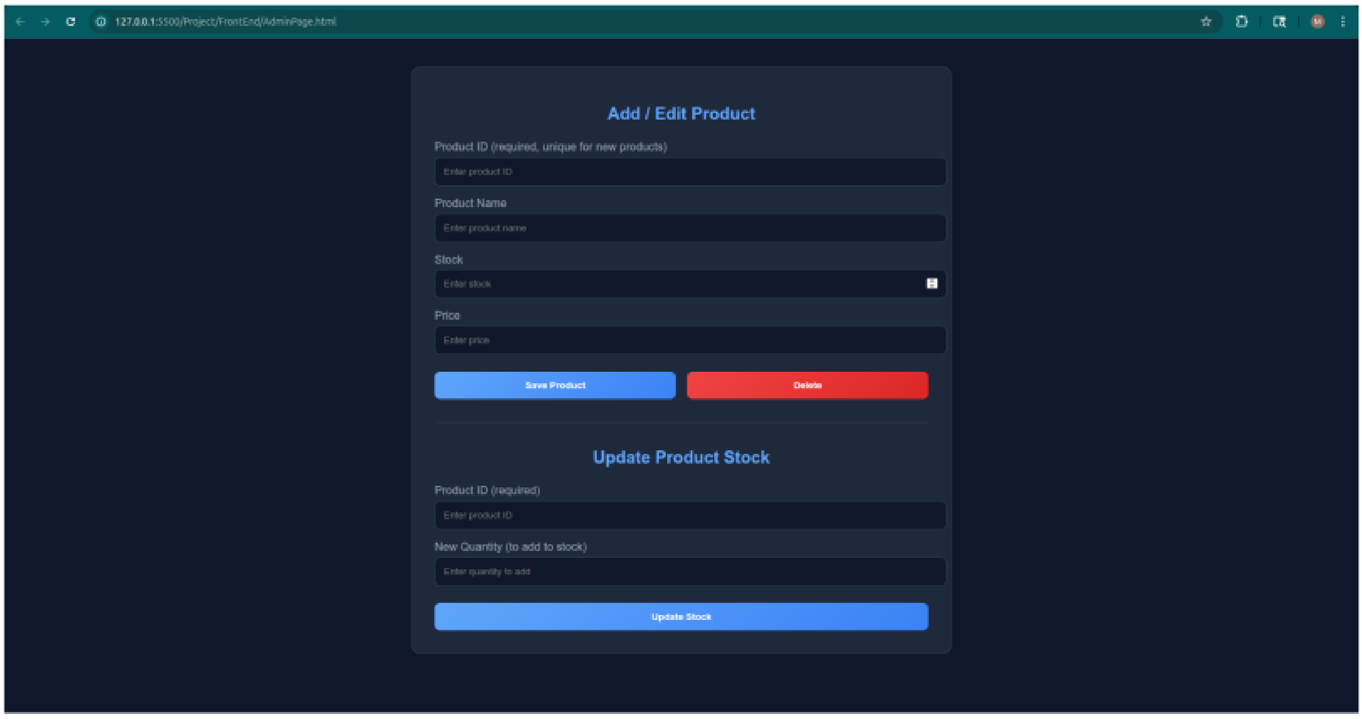


Figure 2: Frontend Admin Page for Product Management

Strengths and Potential Improvements

Strengths

• Robust Validation: Pydantic ensures data integrity for admin inputs.
• Asynchronous Design: FastAPI supports scalable admin operations.
• Comprehensive Logging: Tracks admin actions for auditing.

• CORS Support: Enables seamless frontend integration.
• User-Friendly UI: The frontend provides intuitive product and admin interfaces.

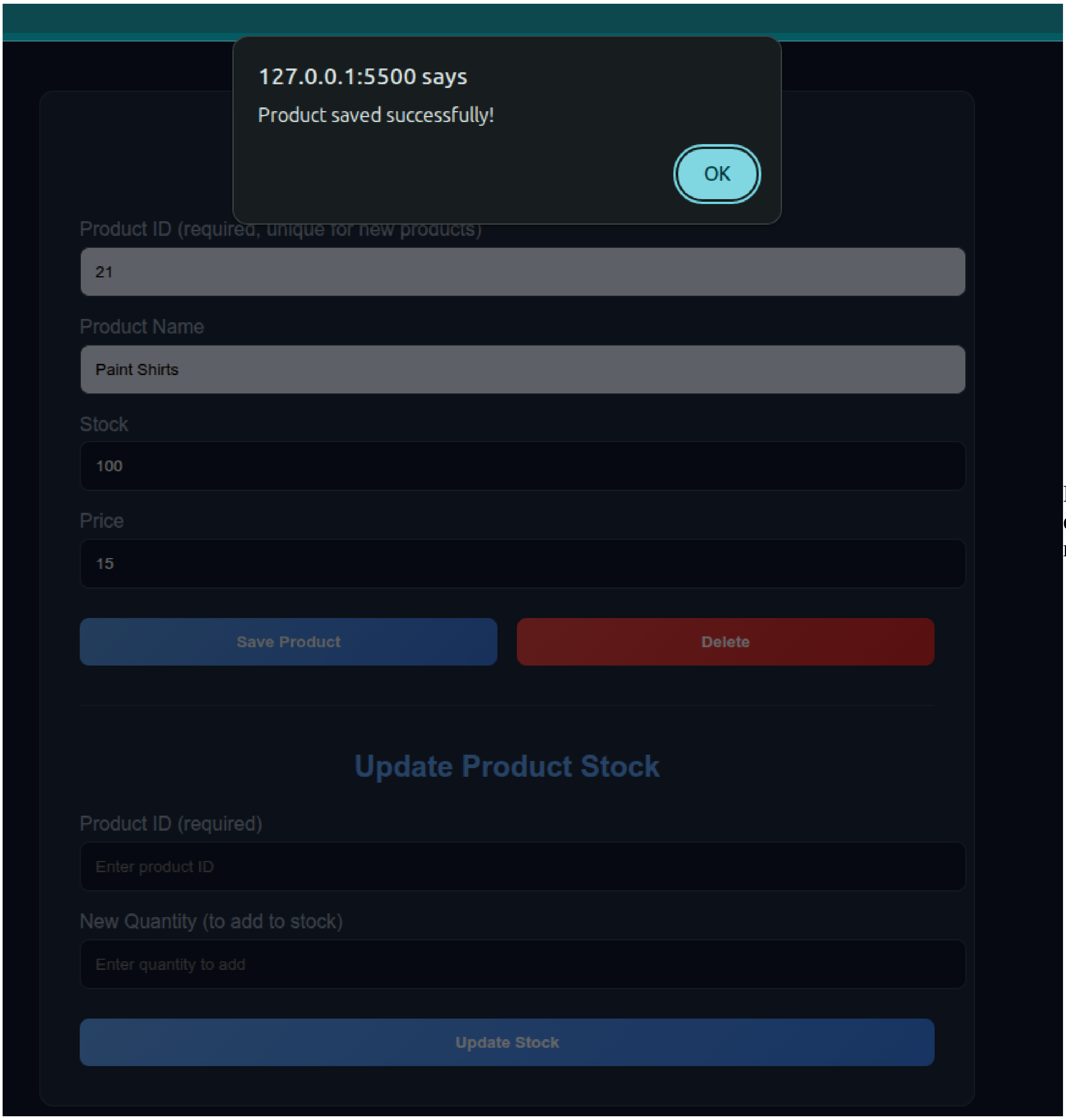
Potential Improvements
• Authentication: Implement JWT or OAuth to restrict admin endpoints.
• Authorization Roles: Differentiate admin and user roles in the API.
• Database Indexing: Add indexes on id for faster queries.
• Unit Tests: Test concurrent admin operations and edge cases.
• Audit Logs: Store admin actions in a separate collection for traceability.
• Image Handling: Add support for product images in the microservice and frontend.

Conclusion
The product microservice effectively supports admin operations for adding, updating, and deleting products, alongside inventory management for order processing. The integrated frontend provides a user-friendly product display and admin interface, as shown in the images. And provides services to the Order service.

All the communication is done through REST api as every service is running in a container using docker compose.

DockerFile:
```
FROM python:3.12.10-slim

WORKDIR /app

COPY requirements.txt .
RUN apt-get update && apt-get install -y --no-install-recommends gcc libc-dev && \
pip install --no-cache-dir -r requirements.txt
COPY . .

EXPOSE 8000

CMD ["python", "product_service.py"]
```

Snippets:


127.0.0.1:5500 says

Product saved successfully!

Product ID (required, unique for new products)

21

Product Name

Paint Shirts

Stock

100

Price

15

Update Product Stock

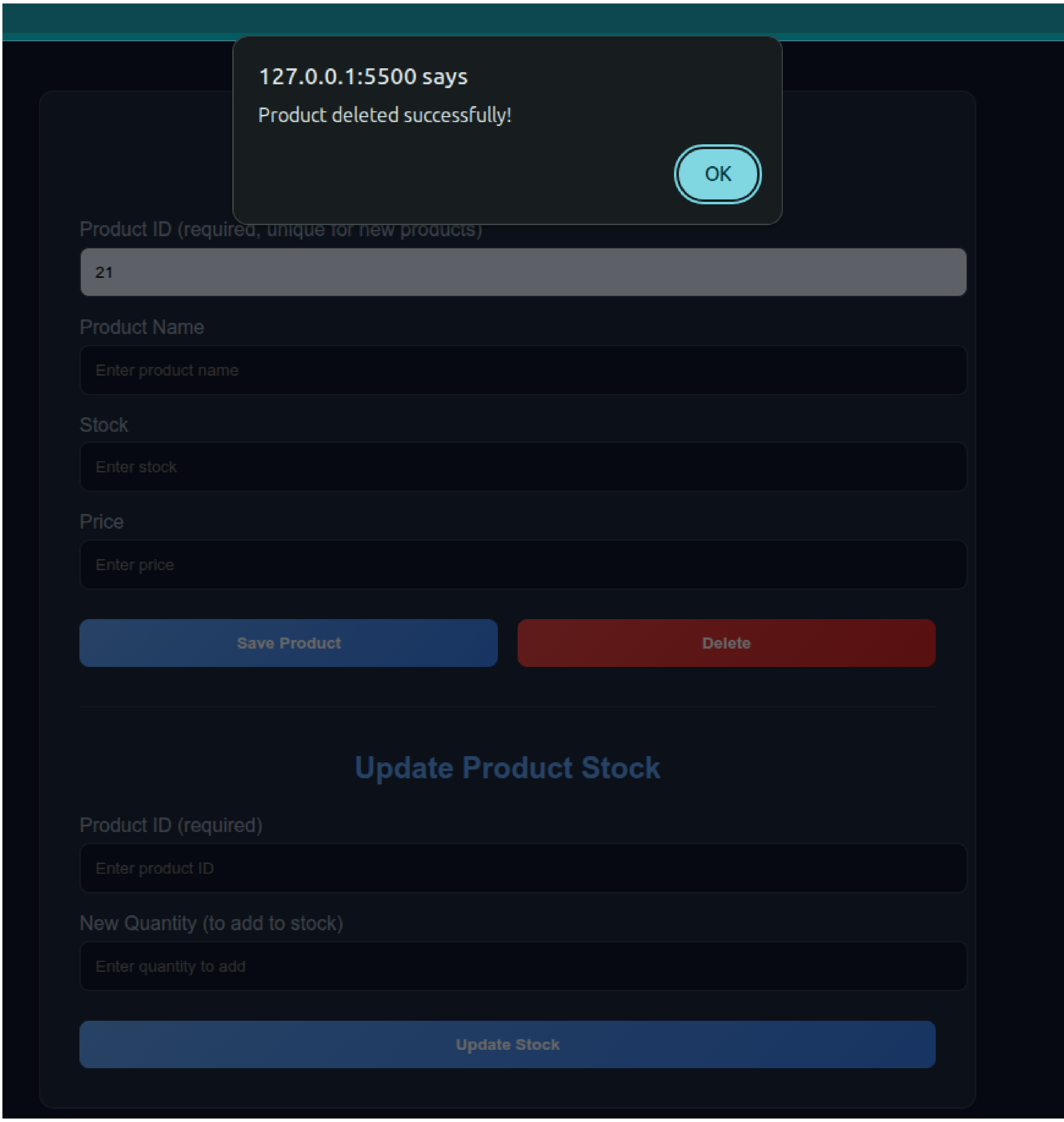
Product ID (required)

Enter product ID

New Quantity (to add to stock)

Enter quantity to add

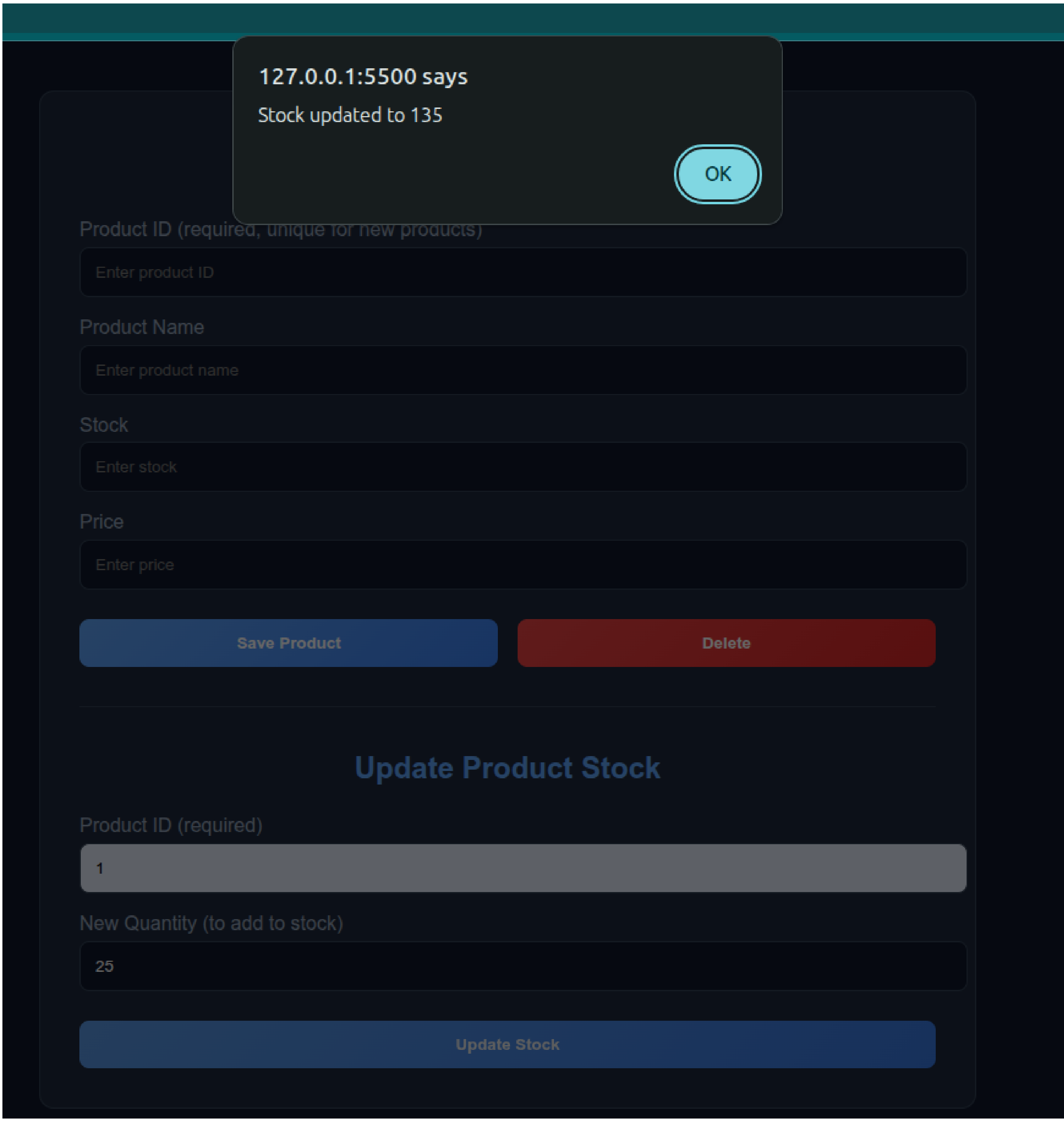
Del
etio
n:



127.0.0.1:5500 says

Product deleted successfully!

Stock updation:



127.0.0.1:5500 says

Stock updated to 135

Product ID (required, unique for new products)

Enter product ID

Product Name

Enter product name

Stock

Enter stock

Price

Enter price

Update Product Stock

Product ID (required)

1

New Quantity (to add to stock)

25

Team-D- Lead: **Mustafa & Hamza**
Microservice: **Order Service**

#	Members
1	Awais Bin Abdul Khaliq
2	Faris Ahmed
3	Munhib Baig
4	**Usman**
5	**Rohail Iqbal**

Order Service API Documentation

Overview

The **Order Service** is a microservice responsible for:

- Validating product availability from the **Product Service**
- Processing payment via the **Payment Service**
- Saving order details into **MongoDB**
- Initiating shipping via **Shipping Service**

⍰ Technologies Used

- **Flask** for the API
- **MongoDB (Atlas)** for order storage
- **Requests** for service communication
- **Docker** compatible internal service URLs
- **CORS** enabled for cross-origin access

⍰ Endpoints

POST /products

➢*Description*

Places an order by:

1. Verifying inventory for each product
2. Processing payment

3. Saving order data

4. Initiating shipping

➢*Request Body (JSON)*

```
{
  "customerid": "12345",
  "customername": "John Doe",
  "product": [
    ["prod01", "Laptop", 1, 750],
    ["prod02", "Mouse", 2, 25]
  ],
  "shipping_address": "123 Main Street, City, Country"
}
```

➢Fields

Field	Type	Description
customerid	string	Customer ID
customername	string	Name of the customer
product	list	List of products: [product_id, name, quantity, price]
shipping_address	string	Address to ship the products

➢Response (on success)

```
{
  "payment": {
    "status": "success",
    "transaction_id": "txn_abc123"
  },
  "order_id": "665f1234abcd5678efgh9012",
  "shipping": {
    "status": "success",
    "tracking_id": "SHIP123456789"
  }
}
```

➢Response (on failure)

```
{
  "status": "failed",
  "message": "Product prod01 not available in requested quantity."
}
```

Internal Function Details

save_order(customerid, customername, products, total_cost, payment_status)

Stores the order in MongoDB with a timestamp.

Returns the MongoDB _id as the order_id.

product_available(product_id, quantity)

Checks inventory from Product Service.

URL used: http://product-service:8000/inventory/<product_id>?quantity=<quantity>

process_payment(id, name, cost, method="cod", payment_details=None)

Sends payment info to Payment Service.

URL used: http://localhost:5001/pay

Supports future payment_details expansion.

shipping_service(order_id, customer_id, products, shipping_address)

Sends shipping request to Abdullah's Shipping PHP microservice.

URL used: http://shippingservice:5001/ship

? MongoDB Structure

Database: OrderService
Collection: orders

Document Schema

```
{
  "customer_id": "12345",
  "customer_name": "John Doe",
  "products": [
    {
      "id": "prod01",
      "name": "Laptop",
      "quantity": 1,
      "price": 750
    }
  ],
  "total_cost": 800,
  "payment_status": {
  "status": "success",
    "transaction_id":  "txn_abc123"
  },
  "timestamp":  "2025-06-09T12:00:00Z"
}
```

? Running the Service

Prerequisites

Python 3.8+

MongoDB Atlas connection string

Product, Payment, and Shipping services running (use Docker internal names)

DEMO IN FORM OF A VIDEO

Demo Video of Order Service.mp4

Team-E- Lead: **Saim Haider & Zaighum Zarawar**
Microservice: **Payment Service**

#	Members
1	**Usman**
2	Muzammil Waheed
3	Muhammad Hamza
4	**Hafiz Muhammad Abdullah**

Payment Service:

The Payment Service is a dedicated microservice responsible for processingtransactions within the E-commerce Order Processing System. Developed using Flask (and integrated with MongoDB for persistent storage, it ensures that all payment-related operations are handled securely and efficiently. This service supports multiple payment methods: Stripe, PayPal, and Cash on Delivery .

Upon receiving a request from the Order Service, the Payment Service verifies the transaction details and processes the payment through the specified provider. Stripe payments are handled using tokenized card data, while PayPal payments are confirmed using payer credentials. In the case of COD, the transaction is marked as pending and assigned a custom transaction ID. Once the payment is processed (or marked pending), the service logs the transaction with a timestamp, customer and order IDs, status, amount, and transaction identifier in the MongoDB collection.

This service plays a pivotal role in the dependency chain by confirming successful transactions and updating the Order Service, which in turn triggers shipping and notification flows. The Payment Service exposes its functionality via a restful API endpoint (/pay), enabling seamless integration across the system’s containerized architecture managed via Docker. The service ensures transactional integrity, auditability, and supports future scalability by design.

Team-F- Lead: **Muhammad Abdullah Amir**
Microservice: **Shipping Service**

#	Members
1	Bakht Nasir
2	Abdul Wakeel
3	Ahmed Ali
4	**Abdullah Khan**
5	**Hafiz Abdur Rehman**

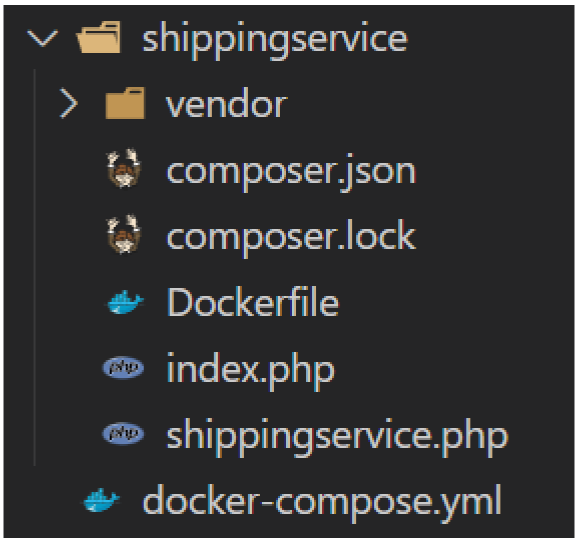
Shipping Service Documentation:

Service Overview

This is a **PHP-based microservice** that:

- Accepts shipping requests via HTTP `POST /ship`
- Saves shipment info to MongoDB
- Sends notifications via REST (to Hashim’s notification service)
- Runs inside Docker using a custom Dockerfile

� Directory Structure



```
shippingservice/
├── composer.json              # PHP dependencies
├── composer.lock              # Lock file
├── Dockerfile                 # Docker config for PHP environment
├── shippingservice.php        # Main app logic
├── index.php                  # (Optional) Entry or test point
├── vendor/                    # Composer autoload dependencies
```

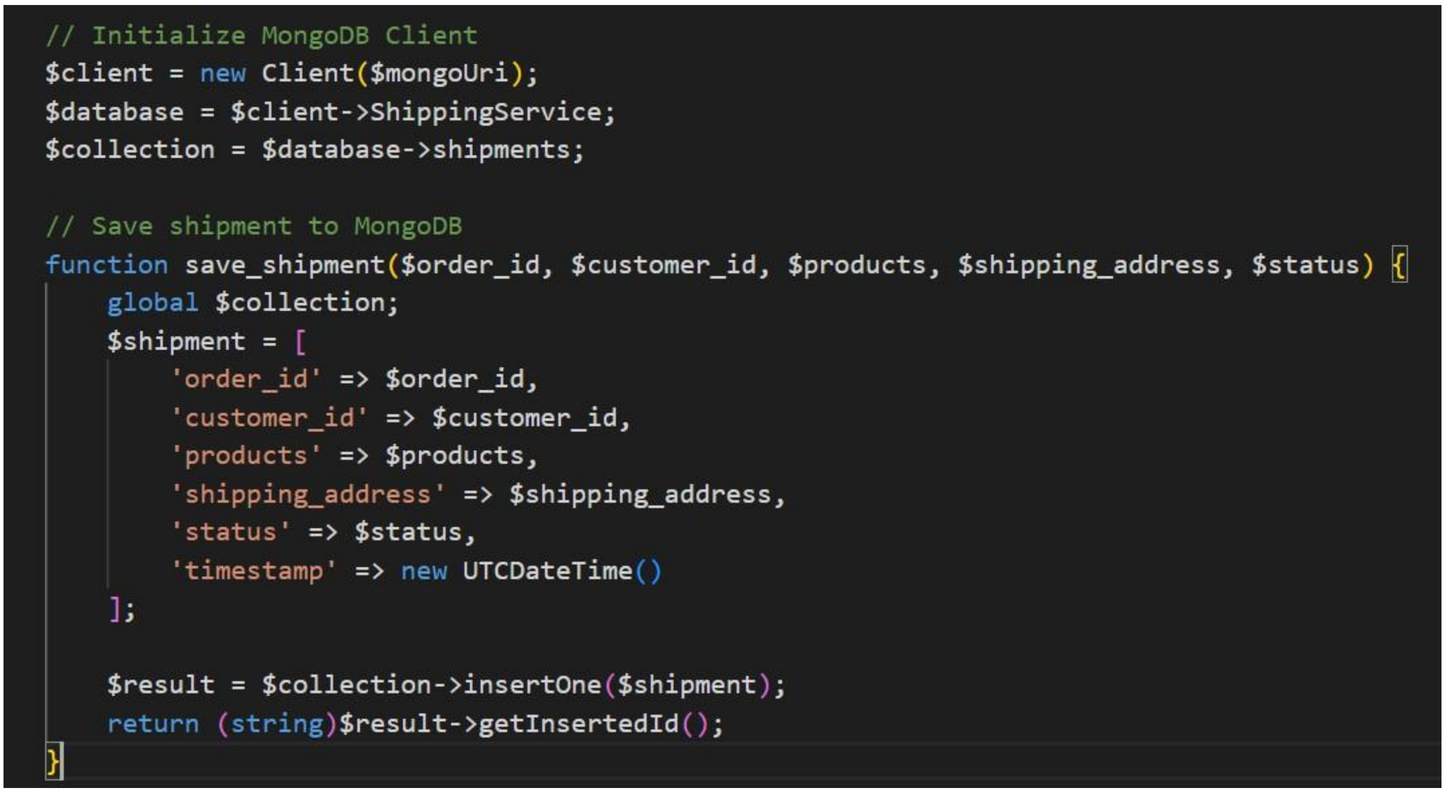
⍰ API Endpoint

POST `/ship`
Used by other services (like Order Service) to initiate a shipment.

Expected JSON Payload:

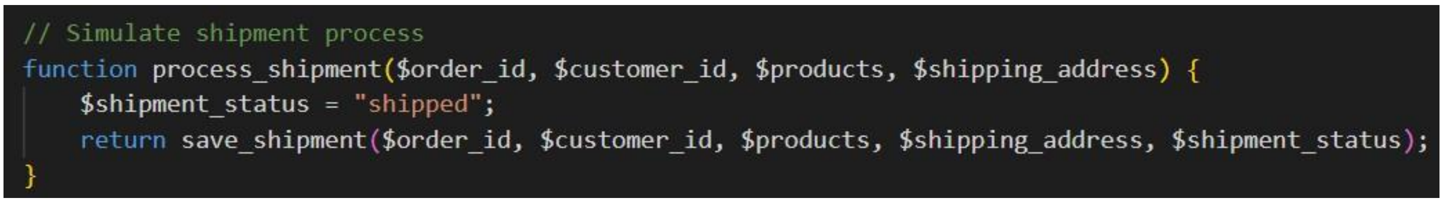
```
{
  "order_id": "ORD123",
  "customer_id": "CUS456",
  "products": [
    { "id": "P1", "name": "Laptop", "quantity": 1, "price": 1000 }
  ],
  "shipping_address": "123 Main Street, Lahore"
}
```

Internal Logic (shippingservice.php)



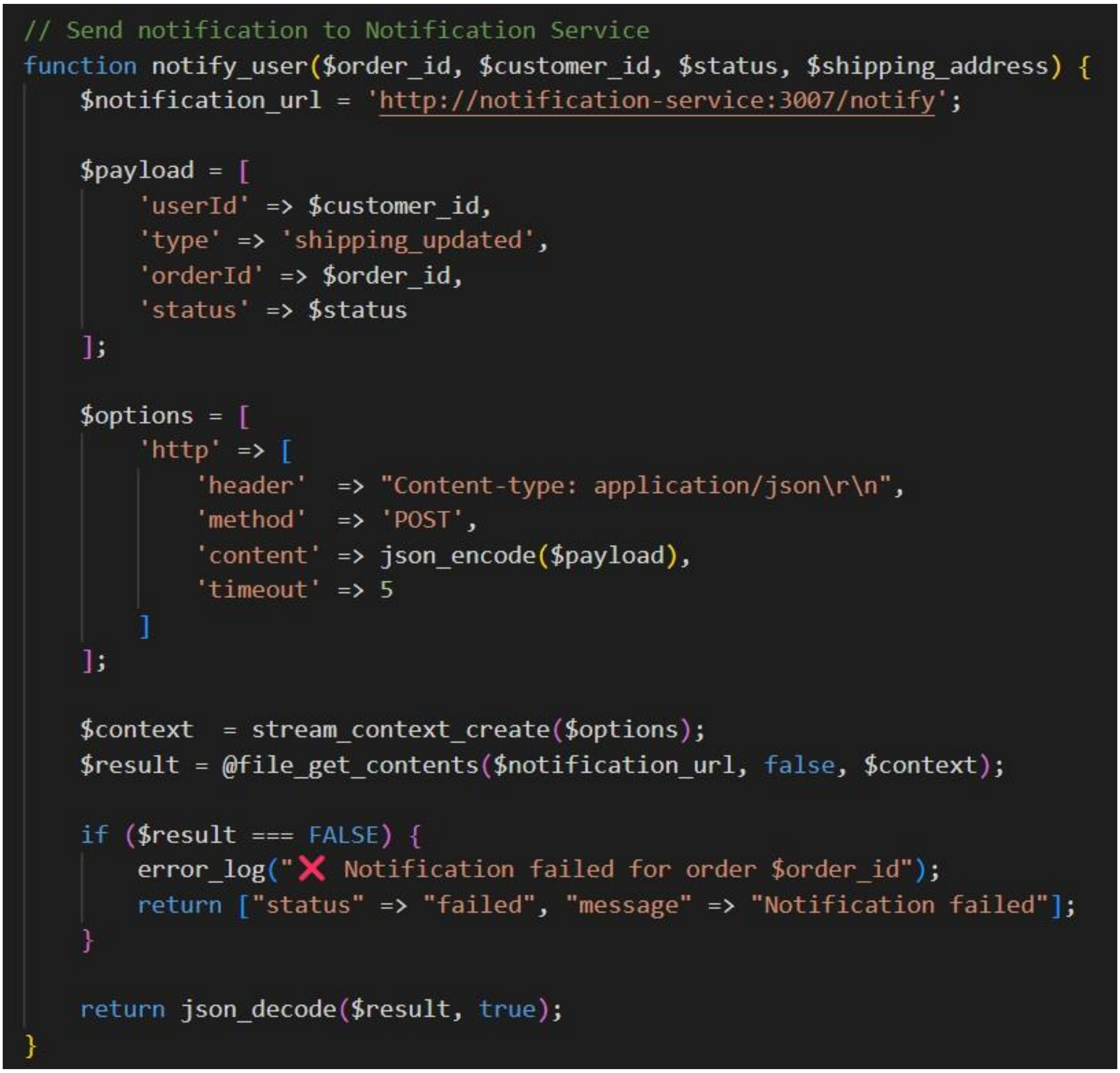
```
// Initialize MongoDB Client
$client = new Client($mongoUri);
$database = $client->ShippingService;
$collection = $database->shipments;

// Save shipment to MongoDB
function save_shipment($order_id, $customer_id, $products, $shipping_address, $status) {
    global $collection;
    $shipment = [
        'order_id' => $order_id,
        'customer_id' => $customer_id,
        'products' => $products,
        'shipping_address' => $shipping_address,
        'status' => $status,
        'timestamp' => new UTCDateTime()
    ];

    $result = $collection->insertOne($shipment);
    return (string)$result->getInsertedId();
}
```

1. Saves the shipment to **MongoDB** (`ShippingService`)

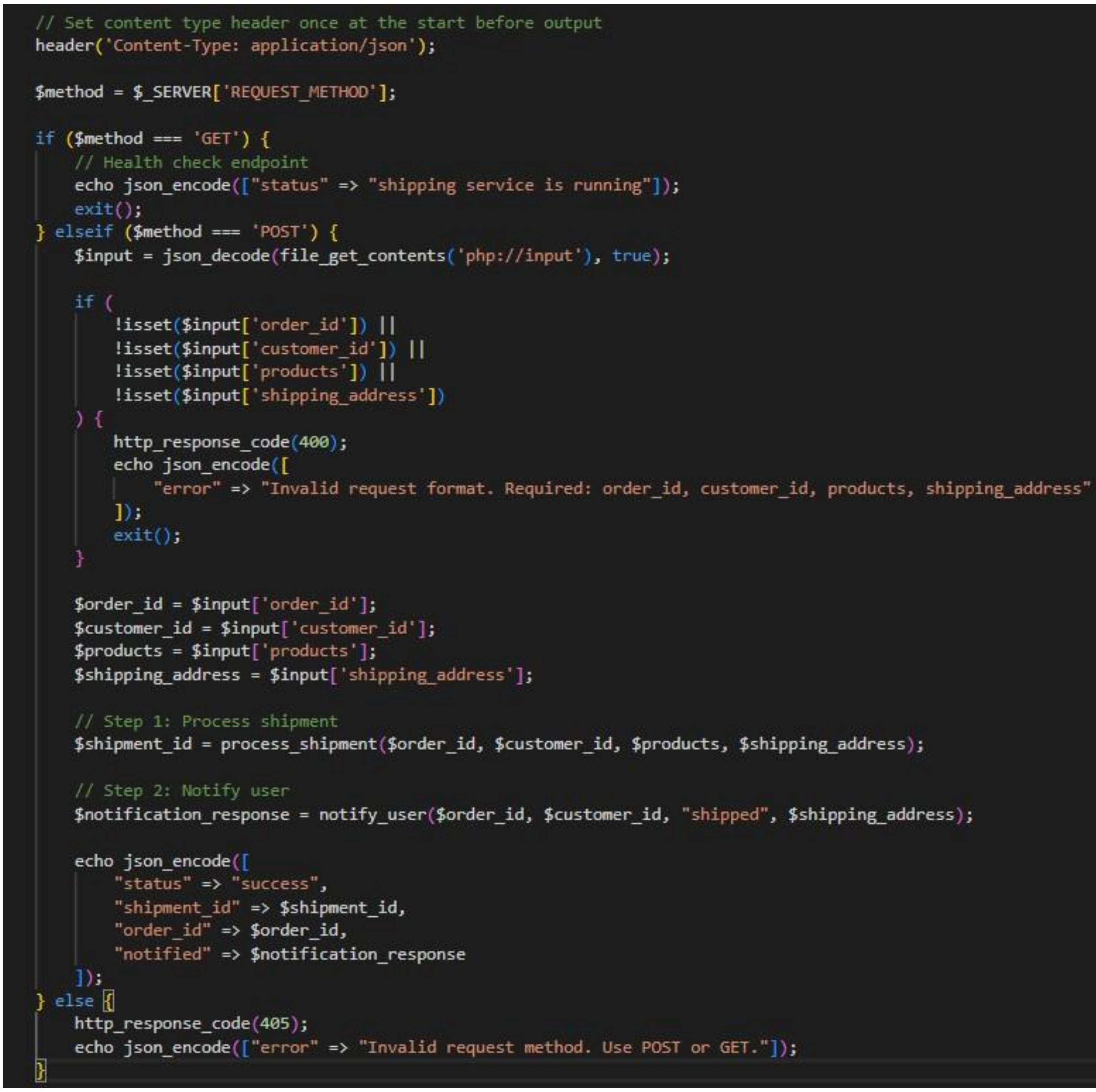


```
// Simulate shipment process
function process_shipment($order_id, $customer_id, $products, $shipping_address) {
    $shipment_status = "shipped";
    return save_shipment($order_id, $customer_id, $products, $shipping_address, $shipment_status);
}
```

2. Processes shipment


```php
// Send notification to Notification Service
function notify_user($order_id, $customer_id, $status, $shipping_address) {
    $notification_url = 'http://notification-service:3007/notify';

    $payload = [
        'userId' => $customer_id,
        'type' => 'shipping_updated',
        'orderId' => $order_id,
        'status' => $status
    ];

    $options = [
        'http' => [
            'header'  => "Content-type: application/json\r\n",
            'method'  => 'POST',
            'content' => json_encode($payload),
            'timeout' => 5
        ]
    ];

    $context  = stream_context_create($options);
    $result = @file_get_contents($notification_url, false, $context);

    if ($result === FALSE) {
        error_log("❌ Notification failed for order $order_id");
        return ["status" => "failed", "message" => "Notification failed"];
    }

    return json_decode($result, true);
}
```

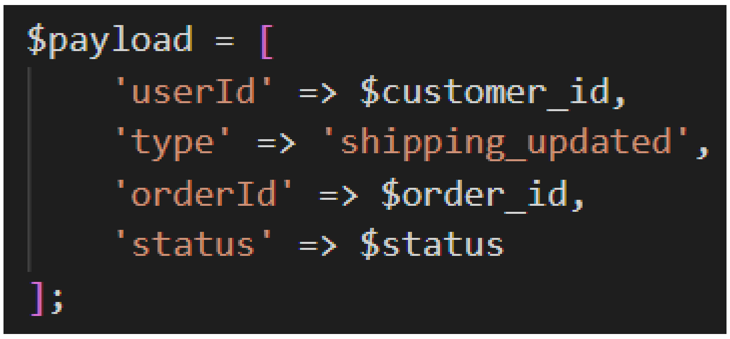
3. Sends notification via `POST` to:
   `http://notification-service:3007/notify`



```php
// Set content type header once at the start before output
header('Content-Type: application/json');

$method = $_SERVER['REQUEST_METHOD'];

if ($method === 'GET') {
    // Health check endpoint
    echo json_encode(["status" => "shipping service is running"]);
    exit();
} elseif ($method === 'POST') {
    $input = json_decode(file_get_contents('php://input'), true);

    if (
        !isset($input['order_id']) ||
        !isset($input['customer_id']) ||
        !isset($input['products']) ||
        !isset($input['shipping_address'])
    ) {
        http_response_code(400);
        echo json_encode([
            "error" => "Invalid request format. Required: order_id, customer_id, products, shipping_address"
        ]);
        exit();
    }

    $order_id = $input['order_id'];
    $customer_id = $input['customer_id'];
    $products = $input['products'];
    $shipping_address = $input['shipping_address'];

    // Step 1: Process shipment
    $shipment_id = process_shipment($order_id, $customer_id, $products, $shipping_address);

    // Step 2: Notify user
    $notification_response = notify_user($order_id, $customer_id, "shipped", $shipping_address);

    echo json_encode([
        "status" => "success",
        "shipment_id" => $shipment_id,
        "order_id" => $order_id,
        "notified" => $notification_response
    ]);
} else {
    http_response_code(405);
    echo json_encode(["error" => "Invalid request method. Use POST or GET."]);
}
```

```
4. main
```

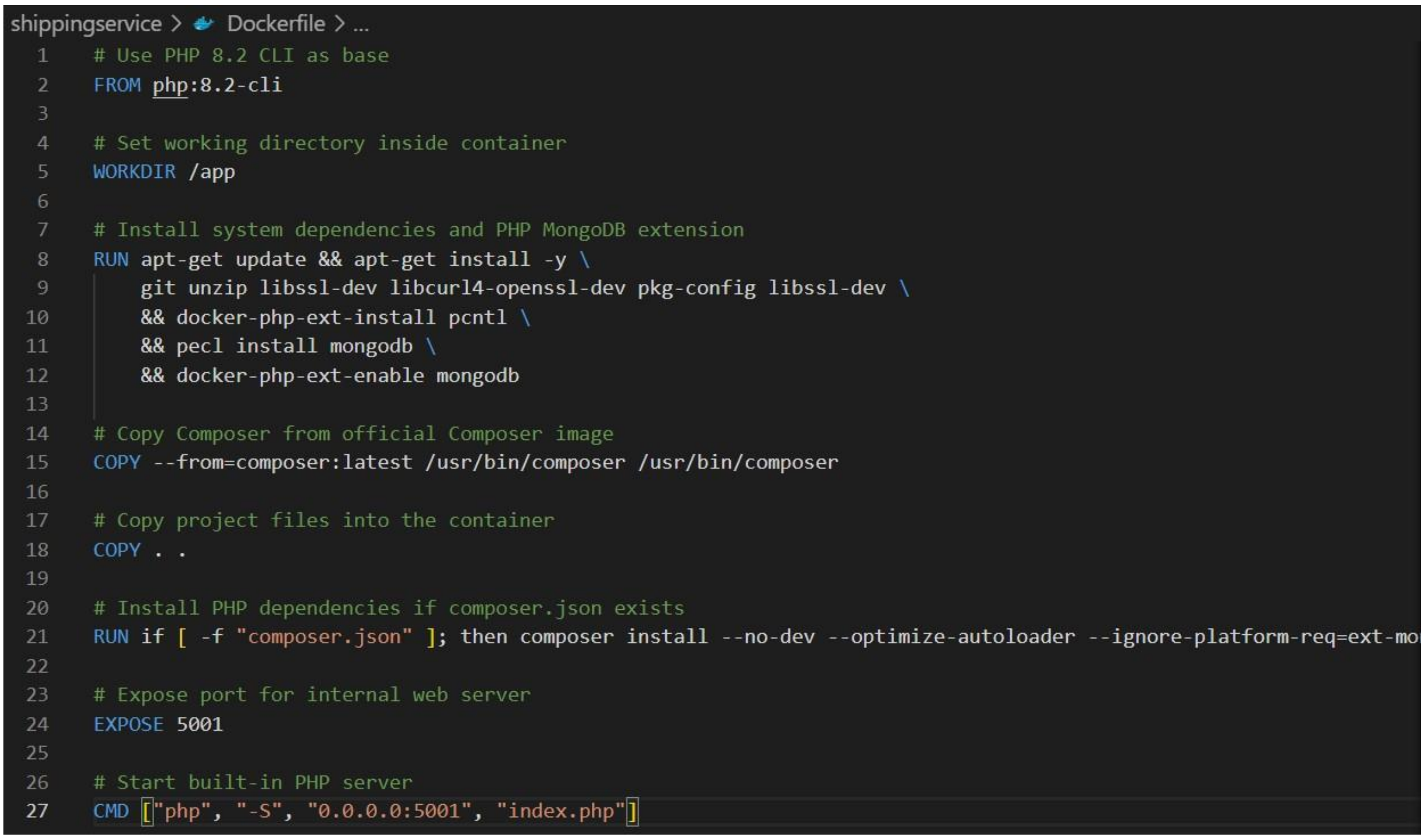
Notification Format Sent



```php
$payload = [
    'userId' => $customer_id,
    'type' => 'shipping_updated',
    'orderId' => $order_id,
    'status' => $status
];
```

```
{
  "userId": "CUS456",
  "type": "shipping_updated",
  "orderId": "ORD123",
  "status": "shipped"
}
```

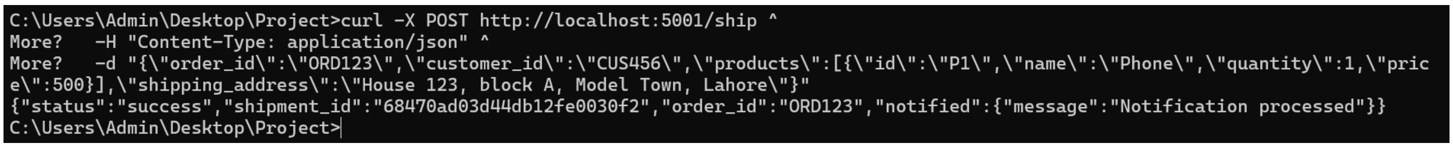
⍰ Dockerfile (Summary)



```
shippingservice > Dockerfile > ...
# Use PHP 8.2 CLI as base
FROM php:8.2-cli

# Set working directory inside container
WORKDIR /app

# Install system dependencies and PHP MongoDB extension
RUN apt-get update && apt-get install -y \
    git unzip libssl-dev libcurl4-openssl-dev pkg-config libssl-dev \
    && docker-php-ext-install pcntl \
    && pecl install mongodb \
    && docker-php-ext-enable mongodb

# Copy Composer from official Composer image
COPY --from=composer:latest /usr/bin/composer /usr/bin/composer

# Copy project files into the container
COPY . .

# Install PHP dependencies if composer.json exists
RUN if [ -f "composer.json" ]; then composer install --no-dev --optimize-autoloader --ignore-platform-req=ext-mo

# Expose port for internal web server
EXPOSE 5001

# Start built-in PHP server
CMD ["php", "-S", "0.0.0.0:5001", "index.php"]
```

- Based on `php:8.1-cli`
- Installs `mongodb` extension
- Installs dependencies via Composer
- Runs `shippingservice.php` on port `5001`

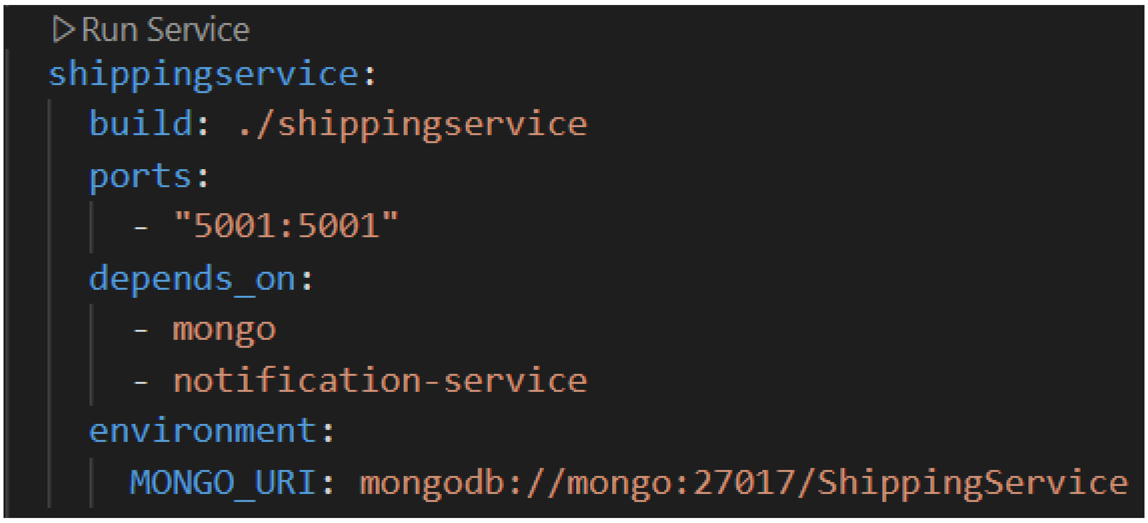
Example curl Command (Test on Localhost)



```
C:\Users\Admin\Desktop\Project>curl -X POST http://localhost:5001/ship ^
More?   -H "Content-Type: application/json" ^
More?   -d "{\"order_id\":\"ORD123\",\"customer_id\":\"CUS456\",\"products\":[{\"id\":\"P1\",\"name\":\"Phone\",\"quantity\":1,\"pric
e\":500}],\"shipping_address\":\"House 123, block A, Model Town, Lahore\"}"
{"status":"success","shipment_id":"68470ad03d44db12fe0030f2","order_id":"ORD123","notified":{"message":"Notification processed"}}
C:\Users\Admin\Desktop\Project>
```

```
curl -X POST http://localhost:5001/ship \
  -H "Content-Type: application/json" \
  -d '{
        "order_id": "ORD123",
        "customer_id": "CUS456",
        "products": [{"id": "P1", "name": "Phone", "quantity": 1, "price": 500}],
        "shipping_address": "House 123, block A, Model Town, Lahore"
      }'
```

docker-compose.yml Integration

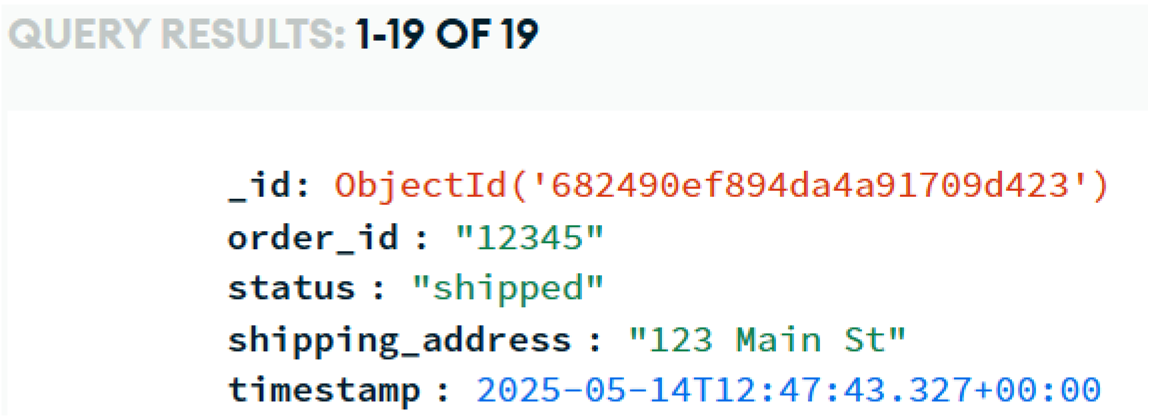


```yaml
shippingservice:
  build: ./shippingservice
  ports:
    - "5001:5001"
  depends_on:
    - mongo
    - notification-service
  environment:
    MONGO_URI: mongodb://mongo:27017/ShippingService
```

`docker-compose.yml` includes:

```
shippingservice:
  build: ./shippingservice
  ports:
    - "5001:5001"
```

▯ MongoDB Collection



```
_id: ObjectId('682490ef894da4a91709d423')
order_id : "12345"
status : "shipped"
shipping_address : "123 Main St"
timestamp : 2025-05-14T12:47:43.327+00:00
```

Database: `ShippingService`
Collection: `shipments`

Each record includes:

- `order_id`
- `customer_id`
- `products`
- `shipping_address`
- `status`
- `timestamp`

Troubleshooting

- 400 Error? → Check if required fields are missing in payload
- Notification not working? → Verify notification-service URL & port

Team-G- Lead: **Hashim Abdullah**
Microservice: **Notification Service**

#	Members
1	Misbah Ur rehman
2	Imran Ullah
3	Ubaid Malik
4	**Hammad Hassan**
5	**Aqsam Qureshi**
6	

Notification Service Report

Overview

The Notification Service is a critical microservice in the E-commerce Order Processing System, responsible for sending real-time notifications to users regarding order confirmations and shipping updates. Built using Node.js with Express, it integrates MongoDB for persistent notification logging and RabbitMQ for event-driven communication. The service leverages WebSocket (via Socket.io) to push notifications to the frontend in real-time, ensuring users receive instant updates. Deployed in a Docker container, it interacts with other services through REST APIs and a message queue system.

Technologies Used

Node.js (Express): A lightweight framework for building the REST API.
MongoDB: A NoSQL database for storing notification logs.
Mongoose: MongoDB object modeling tool for Node.js.
RabbitMQ: Message broker for handling asynchronous events.
Socket.io: Enables real-time bidirectional communication with the frontend.
Docker: Containerization for deployment.

Implementation Details

MongoDB Setup

The service connects to a MongoDB instance to log notifications. The connection is established using Mongoose with the URL configured as an environment variable or defaulting to `mongodb://localhost:27017/notifications`.

RabbitMQ Integration

The service listens to the `notification_queue` in RabbitMQ, processing events such as `order_confirmed` and `shipping_updated`. Upon receiving a message, it parses the event, generates an appropriate notification message, logs it to MongoDB, and broadcasts it to the frontend via WebSocket.

Notification Processing

The `processNotification` function handles different event types:

`order_confirmed`: Generates a message like "Order {orderId} confirmed for user {userId}".
`shipping_updated`: Generates a message like "Order {orderId} shipping status: {status}".
Unknown events are logged and ignored.

WebSocket Broadcasting

Using Socket.io, the service emits the notification message to all connected frontend clients under the "notification" event, enabling real-time updates.

REST API Endpoint

The `/notify` POST endpoint allows manual notification triggering, accepting `userId`, `type`, `orderId`, and `status` in the request body. It validates required fields and processes the notification accordingly.

Running the Microservice

The service runs on port 3008 by default:

- Start the application with `node index.js`.
- Ensure MongoDB and RabbitMQ are running (e.g., via Docker containers).
- Test with tools like Postman by sending POST requests to `http://localhost:3008/notify`.

Security Considerations

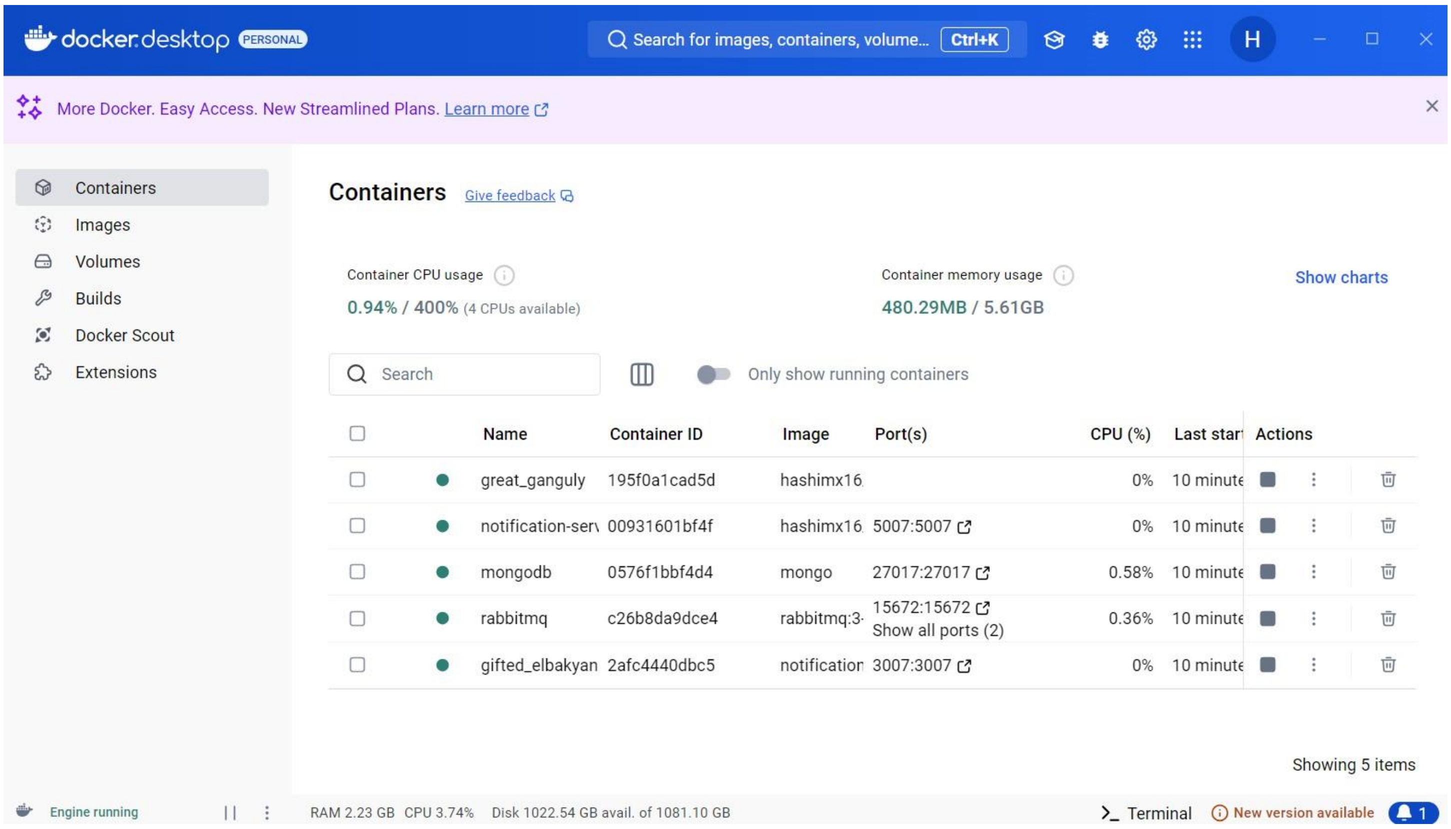
- Input validation is implemented for the `/notify` endpoint.
- WebSocket connections use CORS with wildcard origin for flexibility (to be restricted in production).
- Notification data is persisted securely in MongoDB.

Future Improvements

- Implement authentication (e.g., JWT) to secure the `/notify` endpoint.
- Add email/SMS notification channels alongside WebSocket.
- Enhance error handling for RabbitMQ connection failures.
- Implement rate limiting to prevent abuse of the notification system.
- Add user-specific WebSocket rooms for targeted notifications.

Conclusion

The Notification Service effectively handles real-time user notifications, integrating seamlessly with the E-commerce system’s microservices architecture. With MongoDB for logging and RabbitMQ for event handling, it ensures reliability and scalability. Implementing the suggested improvements will enhance security and functionality, aligning with the project’s goals.



Containers

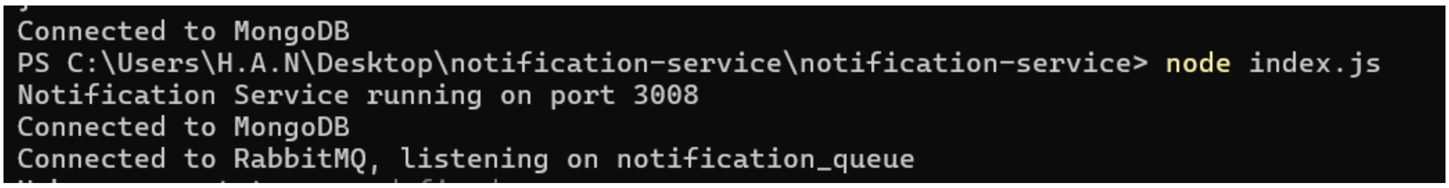
Container CPU usage
0.94% / 400% (4 CPUs available)

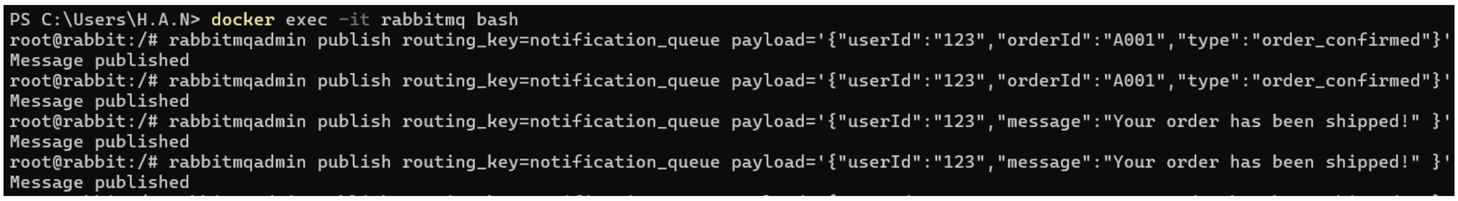
Container memory usage
480.29MB / 5.61GB

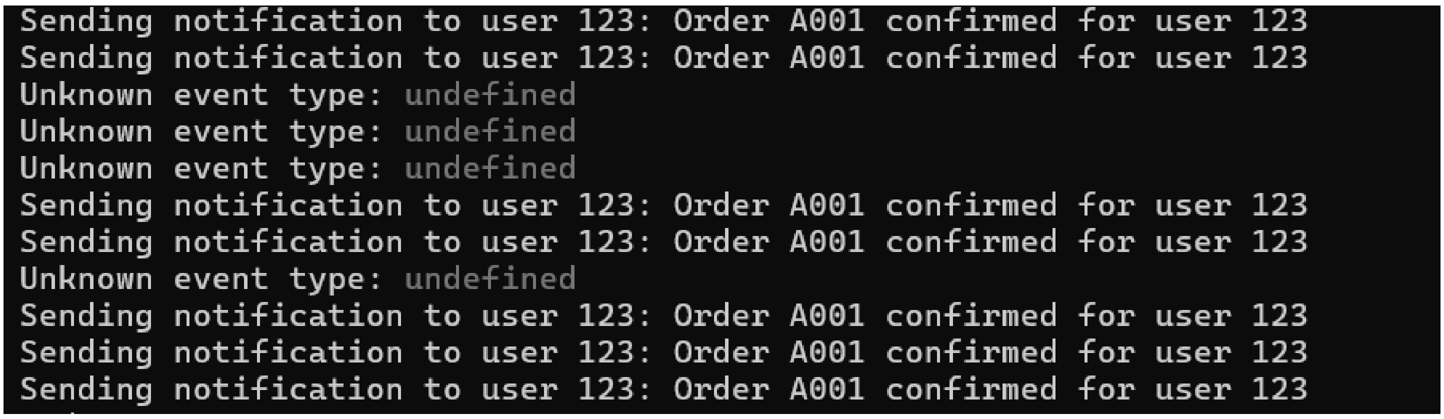
		Name	Container ID	Image	Port(s)	CPU (%)	Last start	Actions
☐	●	great_ganguly	195f0a1cad5d	hashimx16		0%	10 minute	
☐	●	notification-serv	00931601bf4f	hashimx16	5007:5007	0%	10 minute	
☐	●	mongodb	0576f1bbf4d4	mongo	27017:27017	0.58%	10 minute	
☐	●	rabbitmq	c26b8da9dce4	rabbitmq:3-	15672:15672 Show all ports (2)	0.36%	10 minute	
☐	●	gifted_elbakyan	2afc4440dbc5	notification	3007:3007	0%	10 minute	

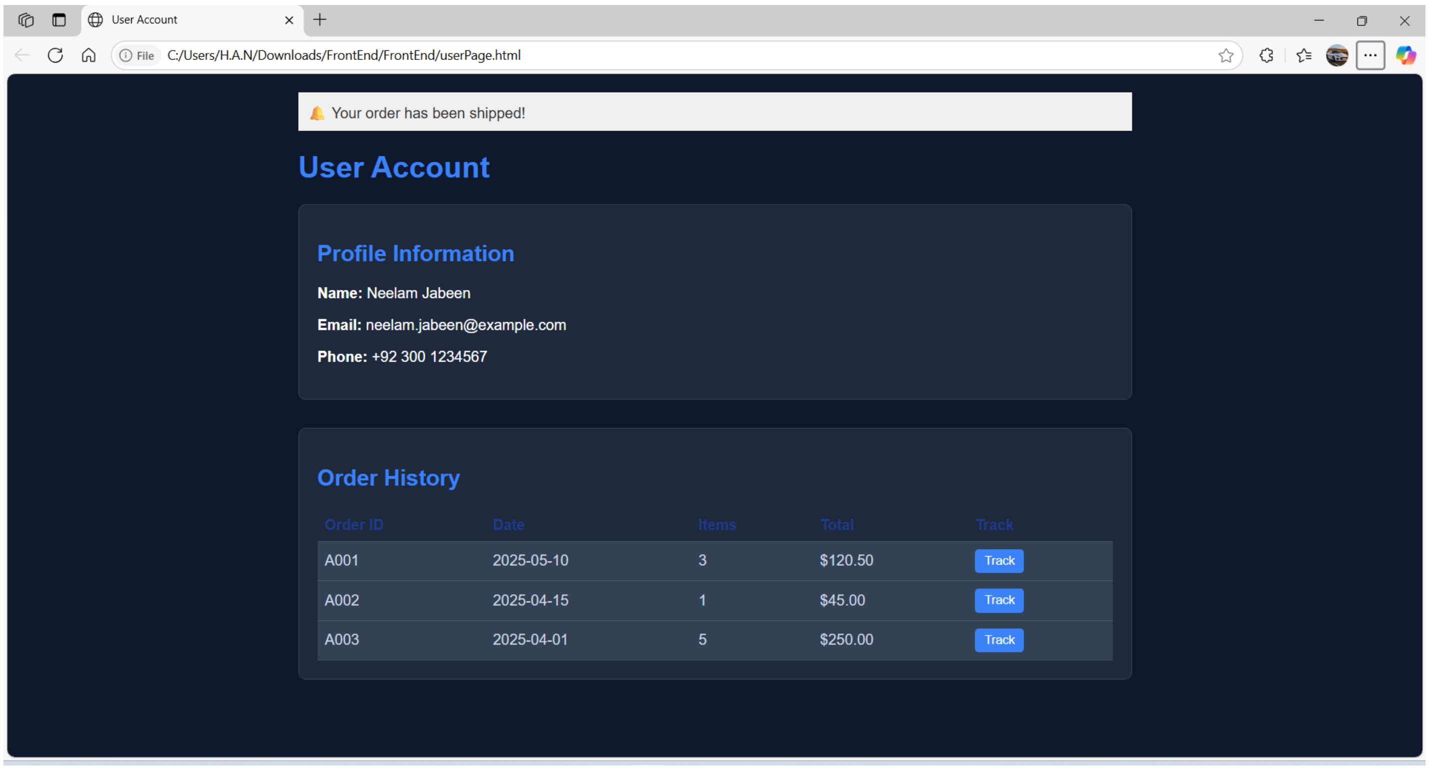
Showing 5 items

RAM 2.23 GB CPU 3.74% Disk 1022.54 GB avail. of 1081.10 GB


```
Connected to MongoDB
PS C:\Users\H.A.N\Desktop\notification-service\notification-service> node index.js
Notification Service running on port 3008
Connected to MongoDB
Connected to RabbitMQ, listening on notification_queue
```


```
PS C:\Users\H.A.N> docker exec -it rabbitmq bash
root@rabbit:/# rabbitmqadmin publish routing_key=notification_queue payload='{"userId":"123","orderId":"A001","type":"order_confirmed"}'
Message published
root@rabbit:/# rabbitmqadmin publish routing_key=notification_queue payload='{"userId":"123","orderId":"A001","type":"order_confirmed"}'
Message published
root@rabbit:/# rabbitmqadmin publish routing_key=notification_queue payload='{"userId":"123","message":"Your order has been shipped!" }'
Message published
root@rabbit:/# rabbitmqadmin publish routing_key=notification_queue payload='{"userId":"123","message":"Your order has been shipped!" }'
Message published
```


```
Sending notification to user 123: Order A001 confirmed for user 123
Sending notification to user 123: Order A001 confirmed for user 123
Unknown event type: undefined
Unknown event type: undefined
Unknown event type: undefined
Sending notification to user 123: Order A001 confirmed for user 123
Sending notification to user 123: Order A001 confirmed for user 123
Unknown event type: undefined
Sending notification to user 123: Order A001 confirmed for user 123
Sending notification to user 123: Order A001 confirmed for user 123
Sending notification to user 123: Order A001 confirmed for user 123
```


User Account
File C:/Users/H.A.N/Downloads/FrontEnd/FrontEnd/userPage.html

🔔 Your order has been shipped!

User Account

Profile Information

Name: Neelam Jabeen

Email: neelam.jabeen@example.com

Phone: +92 300 1234567

Order History

Order ID	Date	Items	Total	Track
A001	2025-05-10	3	$120.50	Track
A002	2025-04-15	1	$45.00	Track
A003	2025-04-01	5	$250.00	Track