**Scalable Services Assignment.**

**Microservice based Ecommerce Application**

**Team :**

**2023TM93522 : Kantamsetti Mohan Durga Sathish**

**2023TM93557 : Eoghan Pereira**

**2023TM93625 : Gouri Remesh**

**2023TM93639 : Anmol Kalra**

**Table of Contents**

[**Team Contributions 3**](#_i6jubz32k0x5)

[**Introduction 4**](#_vmn0eiw2fi2q)

[**Architecture 5**](#_iqfv7h63h3y)

[Components of the eCommerce Application 6](#_bm73jacrh9c9)

[**Communication and Integration 9**](#_xueceidnqqjf)

[Asynchronous Messaging 9](#_xhhhvppchgyj)

[API Gateway 11](#_hg7p6soh9w2g)

[Database Schema 13](#_uljkv3vn529d)

[API Endpoints 16](#_3bys6heaho4u)

[Postman Samples 19](#_ljeg1pzhzk22)

[**Deployment 23**](#_evwkz6tnnggt)

[Deployment with Docker/podman Compose 24](#_shwcj6pfj7uh)

[Transaction Flow 25](#_725vzuvi9bou)

# Team Contributions

|  |  |
| --- | --- |
| Team Member | Role/Component of the application |
| 2023TM93522 | Checkout service, Email service, RabbitMQ Service, Mongodb, Kong API Gateway service.  Architecture Design & Deployment Design, Application Documentation, Containerisation and Deployment of the Application with podman compose. |
| 2023TM93557 | User Microservice, Mongo DB, Postman,  Docker, Documentation |
| 2023TM93625 | Blog/Review Microservice, Docker  Containerisation,Document  updates, RabbitMQ |
| 2023TM93639 | Inventory Microservice, RabbitMQ Consumer and Publisher Service, Docker Containerization and documentation updates |

# **Introduction**

An eCommerce application is a digital platform that enables the buying and selling of products and services online. It offers functionalities like product browsing, user account management, order processing, and customer feedback. The goal is to provide a smooth shopping experience with features such as search options, secure payments, and personalized recommendations. To meet the demands of online retail, the application needs to be robust and capable of handling high traffic, especially during peak shopping times.



Microservices architecture is vital for making an eCommerce application agile and scalable. By breaking down the application into smaller, independent services focused on specific tasks, it allows for flexible development and operation. This means each service, like inventory or checkout, can be scaled up independently to handle increased demand during sales, without affecting other parts of the application.

Microservices also encourage innovation and reliability. They allow different teams to work on separate services using the best technologies for each task, speeding up the introduction of new features. Moreover, if one service experiences an issue, it doesn't necessarily disrupt the entire application, making the platform more resilient. Overall, microservices help eCommerce applications stay competitive and responsive to changes in the digital market.

## 

# **Architecture**

The architecture of our eCommerce application is designed to optimize scalability, reliability, and security through a microservices approach. Client devices interact with the application via an API Gateway, which serves as the central access point, managing and routing requests efficiently. Each core service—User, Inventory, Review, and Checkout—functions independently, allowing for seamless development, deployment, and scaling. These services are interconnected through RabbitMQ, facilitating asynchronous communication to ensure high responsiveness and decoupled interactions. Each service maintains its own database, reinforcing data isolation and service autonomy.

## 

In this architecture, the API Gateway plays a crucial role in securing and streamlining client interactions with the backend services. By leveraging microservices, the application benefits from modularity and flexibility, enabling focused improvements and updates to individual components without disrupting the entire system. RabbitMQ acts as a robust message broker that supports asynchronous communication between services, enhancing fault tolerance and system resilience. This design not only supports the current functional requirements of the eCommerce platform but also provides a solid foundation for future scalability and feature expansion. Each service's dedicated database ensures that data management remains efficient and tailored to specific service needs, further enhancing the application’s performance and reliability.

The service architecture reflects domain-driven design principles, where each microservice focuses on specific business capabilities. This design approach ensures that each service is aligned with business processes, improving clarity and efficiency in development.

**Components of the eCommerce Application**

The architecture of this eCommerce application is based on the microservices model, which divides the application into distinct, independent services. Each microservice is responsible for a specific business function, allowing them to be developed, deployed, and scaled independently. This approach enhances flexibility and efficiency, crucial for a dynamic environment like eCommerce.

**Key Components**

**1. User Service**

The User Service is responsible for managing all aspects of user accounts. This includes user registration, authentication, and profile management. It handles login and logout processes, password management, and user information updates. It also supports user roles and permissions, ensuring that users have appropriate access levels to various features within the application.

Operating with its own database, the User Service ensures secure storage of sensitive user data, such as personal information and hashed passwords. This isolation not only enhances security but also allows for independent scaling and updates, minimizing the risk of data breaches and ensuring compliance with data protection regulations.

**Spring Security**

Spring Security is a powerful and flexible security framework for Spring applications. It provides a comprehensive solution for authentication, authorization, and protection against security vulnerabilities.

**Key Features:**

* **Authentication:** Supports various authentication mechanisms like form-based, HTTP Basic, LDAP, OAuth 2.0, and OpenID Connect.
* **Authorization:** Enforces access control rules based on user roles and permissions.
* **CSRF Protection:** Protects against Cross-Site Request Forgery attacks.
* **Session Management:** Manages user sessions securely.

**OAuth**

OAuth is an open standard authorization framework that allows users to grant third-party applications access to their data without sharing their credentials.

**Key Features:**

* **Authorization Code Flow:** A secure and flexible flow for web applications.
* **Implicit Grant Flow:** Simpler but less secure, often used for mobile and JavaScript applications.
* **Resource Owner Password Credentials Grant:** Less secure, but can be used for simple scenarios.
* **Client Credentials Grant:** Used for machine-to-machine authentication.

**JWT (JSON Web Token)**

A JWT is a standard used to create access tokens that assert some number of claims. It is compact and self-contained, and typically signed using a cryptographic algorithm to ensure that the contents have not been tampered with.

**Key Features:**

* **Compact and Self-Contained:** JWTs are compact and can be easily transmitted.
* **Secure:** JWTs can be signed and encrypted to protect against tampering and unauthorized access.
* **Stateless:** JWTs are stateless, meaning the server does not need to store session information.

**Docker Deployment**

* Docker provides a standardized and lightweight approach to packaging applications and their dependencies into containers. These containers encapsulate the runtime environment, ensuring consistency from development through testing to production.
* In our e-commerce microservices application built using .NET 6, Docker plays a pivotal role in facilitating a smooth and efficient deployment workflow.
* Some key advantages of using Docker are,

**2. Inventory Service:**

The inventory service is used for storing the product catalog information like live product listings, updates and availability of products. It handles product additions, updates (price and description) and deletions to make sure that valid product data reaches the user.

This service is backed by an in-house database that stores inventory data, which fetches the inventory state quickly and handles any changes to the inventory with very little latency. As catalogs increase in size, this method is critical to ensure that product availability will always be current. Built with **.NET** and **MongoDB** together, the service is packaged using **Docker** so it can be deployed and scaled in a consistent way. MongoDB is built to scale, so performance remains solid as the catalog grows.

The service implements event-driven architecture to listen for order events via **RabbitMQ** and updates the inventory synchronously in response. This decouples the Inventory Service from the other services and enables putting them out of service without affecting its fault tolerance.

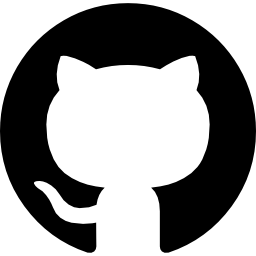
MongoDB offers replication and sharding natively allowing high availability from the service. Scalability and fault tolerance are further ensured by Docker, allowing the service to scale individually as per demand.

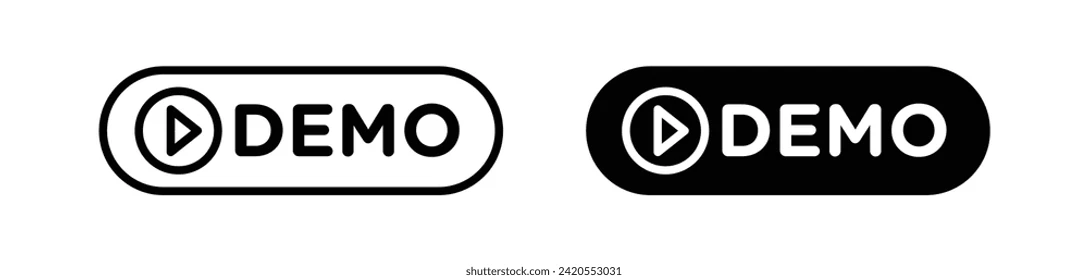
The Inventory Service is designed following microservice principles, with best practices on loose coupling and scaling potential as well as powerful APIs to manage a given source of products. Combining observability, security, and testing to provide compliance confidence in the operation.

GitHub code repository URL: <https://github.com/anmolkalra17/ss-inventory-microservice>  
Demo video drive link: [ss\_inventory\_service\_demo.mov](https://drive.google.com/file/d/1Zmv6s0cpMOmzcKFGVTpNak2lSDs6OGE2/view?usp=sharing)

Demo video YouTube link: <https://youtu.be/vOHM9nm3kyU>

3. **Checkout Service**



Github repo : <https://github.com/MohanKantamsetti/SSAssignment>

**Demo video link(s) :**

[**2023TM93522\_EcommerceApp\_Demo.mp4**](https://wilpbitspilaniacin0-my.sharepoint.com/:v:/g/personal/2023tm93522_wilp_bits-pilani_ac_in/EX4kzFw3NLdEpBQfd6RyvcUBE9owyIrbSFXpHmVy1N6Pfg?e=ZNOMQn) **(sharepoint)** [2023TM93522\_EcommerceApp\_Demo.mp4](https://drive.google.com/file/d/1BRWNQomgSSPpZgknYsbY7dtCY1i2jxNQ/view?usp=sharing) **(GDrive)**

The Checkout Service is a pivotal component of the eCommerce application, orchestrating the entire order processing workflow to ensure a seamless customer experience. This service is implemented using Python Flask, which provides a lightweight and flexible framework for building RESTful APIs. Flask's simplicity and extensibility make it an ideal choice for developing and maintaining the service, allowing for rapid iterations and easy integration with other components.

- Order Management : Service creates a new order in the MongoDB database. MongoDB's flexible schema allows it to accommodate diverse and dynamic order-related information, such as customer details, product specifications, and order status.

The service also allows creation of cart, adding items to cart and managing a cart, which could later be used to place an order.

- Payment Processing : The Checkout Service handles all payment-related functionalities by validating payment details and records transaction data to maintain financial integrity.

- Inter-Service Communication : The service communicates asynchronously with other microservices to update order status and manage inventory:

- Email Notifier Service : Sends order status updates and confirmations to customers. This is achieved by placing messages on an order queue, which the Email Service consumes to trigger notifications.

**Microservice Patterns and Architectural Decisions**

- API and Model Separation : The service follows a clear separation of concerns, with the API layer distinct from the data models. This separation enhances maintainability and allows for independent evolution of the API and underlying data structures.

- Database Independence : MongoDB is used as a separate database, isolated from the application backend. This decoupling allows for flexible scaling and ensures that database operations do not impact the performance of the application logic.

- Decentralized Data Management : Each microservice manages its own data, reducing dependencies and allowing for service-specific optimizations. This pattern supports scalability and resilience, as each service can independently handle its data load.

- Asynchronous Communication : By leveraging RabbitMQ for message brokering, the Checkout Service can asynchronously communicate with the Email and Inventory services. This decoupling allows services to operate independently, improving system resilience and responsiveness.

- Independent Deployment : The Email Service runs independently, allowing it to scale based on demand without affecting the Checkout Service. This autonomy is a core microservice principle, ensuring that services can be developed, deployed, and scaled independently.

Containerized using Podman, the Checkout Service is deployed with Podman Compose. This setup involves two containers running on the same network to provide redundancy and load balancing. This ensures high availability and reliability, allowing the service to handle large volumes of transactions without degradation in performance.

4. **Blogs/Reviews Service**

The Blogs/Reviews Service handles all user-generated content, such as product reviews and blog posts. It enables users to share their experiences and opinions, providing valuable feedback and engagement within the platform. This service supports content moderation, ensuring that all submissions adhere to community guidelines.

It utilizes its own database to store content and feedback, allowing for efficient retrieval and analysis of user-generated data. This separation of storage ensures that the service can scale independently as the volume of content grows, and it simplifies content management and moderation processes.

**Github Repository link**: <https://github.com/gouriremesh22/test>

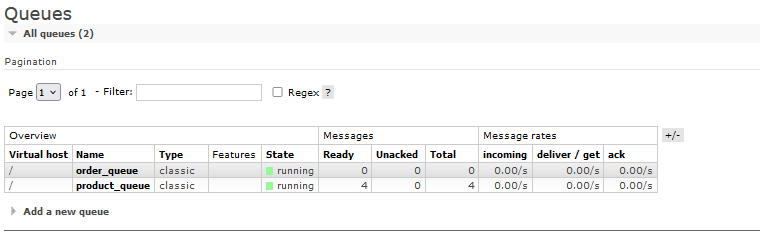
**Demo video link** : [Blogs(review)Service.mov](https://drive.google.com/file/d/1Ehap0wXIGoMLUBG14lU_Gl5slSlETJpY)

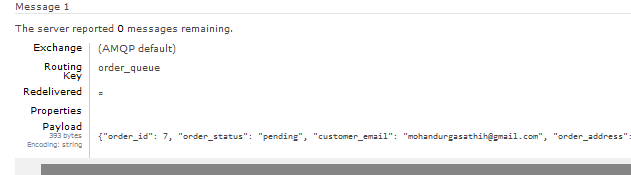
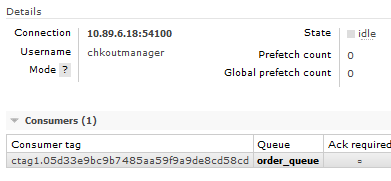
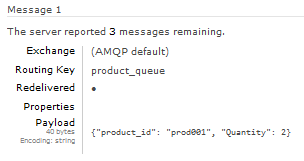
# **Communication and Integration**

## **Asynchronous Messaging**

**- RabbitMQ :** The application uses RabbitMQ as a message broker to facilitate asynchronous communication between microservices. Asynchronous messaging allows services to send and receive messages without requiring both parties to be active at the same time, thereby decoupling service interactions. This design is particularly beneficial in handling high volumes of transactions and events, as it prevents bottlenecks and ensures that services can continue operating independently.

- **Message Queues :** RabbitMQ uses message queues to hold messages until they are processed by the receiving service. This setup is ideal for scenarios where tasks need to be queued and processed asynchronously, such as order fulfillment and inventory updates. By leveraging message queues, the application can achieve greater fault tolerance and reliability, as messages can be retried or rerouted in case of service failure.





**Order Placement and Message Queuing**

When a user places an order via the Checkout Service, this service performs a critical role beyond processing payments and confirming purchases. Upon successful order placement, the Checkout Service writes messages to two distinct queues: order\_queue and product\_queue.

* **Order Queue**: This queue is specifically designated for handling order-related messages. The Checkout Service sends a message to the order\_queue, encapsulating details about the order status, customer information, and other pertinent data. This message acts as a trigger for subsequent actions in the workflow.
* **Product Queue**: Simultaneously, the Checkout Service places a message onto the product\_queue. This message contains essential details about the products involved in the order, including product IDs and quantities. Such information is crucial for maintaining accurate inventory levels across the platform.

**Email Service and Order Status Notifications**

The Email Service is configured to listen to the order\_queue. As new messages arrive, it reads the order details and automatically generates email notifications to inform customers about their order status. These notifications provide essential updates, such as order confirmation and estimated delivery times, thereby enhancing the customer experience and maintaining engagement.

**Inventory Service and Stock Management**

Similarly, the Inventory Service monitors the product\_queue for updates on product quantities. Upon receiving a message, it reads the product IDs and associated quantities, promptly adjusting the inventory records to reflect the most current stock levels. This ensures that the inventory is always up-to-date, preventing issues such as overselling or stockouts.

**Benefits of Asynchronous Communication**

By leveraging RabbitMQ for asynchronous communication, the application benefits from increased fault tolerance and decoupled service interactions. This approach allows services like Email and Inventory to operate independently, processing messages at their own pace without waiting for synchronous responses. As a result, the overall system remains responsive and efficient, even under varying loads or during service interruptions. This architecture not only supports seamless operation and scalability but also enhances the reliability and robustness of the eCommerce platform.

## **API Gateway**

**Single Entry Point**

- Centralized Access : The API Gateway functions as the sole entry point for all client interactions with the backend services. This consolidation simplifies the client-side architecture by providing a single URL or endpoint for accessing the application's functionality, regardless of the number or complexity of underlying microservices.

- Unified Interface : By presenting a unified interface, the API Gateway abstracts the complexities of service interactions from the client. Clients do not need to know the details of service locations or communication protocols, which can vary across different microservices. This abstraction layer facilitates easier client-side development and maintenance.

**Routing and Traffic Management**

- Dynamic Routing : Kong's API Gateway dynamically routes incoming requests to the appropriate microservice based on predefined rules that consider factors such as the URL path, HTTP method, and request headers. This flexibility allows for complex routing scenarios, including versioning and canary deployments.

- Load Balancing : The Gateway can distribute incoming requests evenly across multiple instances of a service, aiding in load balancing and ensuring high availability. This feature helps to optimize resource usage and improve response times, especially under high traffic conditions.

- Traffic Shaping : By managing API traffic, the Gateway can implement traffic shaping strategies, such as prioritizing certain types of traffic or throttling requests to specific services to prevent overload.

**Security and Policy Enforcement**

- Authentication and Authorization : Kong enables various authentication methods, including API keys, JWT, and OAuth2, to ensure secure access to services. By centralizing authentication, the Gateway reduces the burden on individual services to implement and maintain security protocols.

- Rate Limiting and Throttling : The API Gateway can enforce rate limits to prevent abuse and ensure fair usage among consumers. Throttling can be applied based on user identity, IP address, or other criteria, protecting backend services from denial-of-service attacks.

- Access Controls : Kong allows for granular access control policies, ensuring that only authorized users can access specific services or endpoints. Policies can be defined and managed centrally, streamlining security administration and compliance.

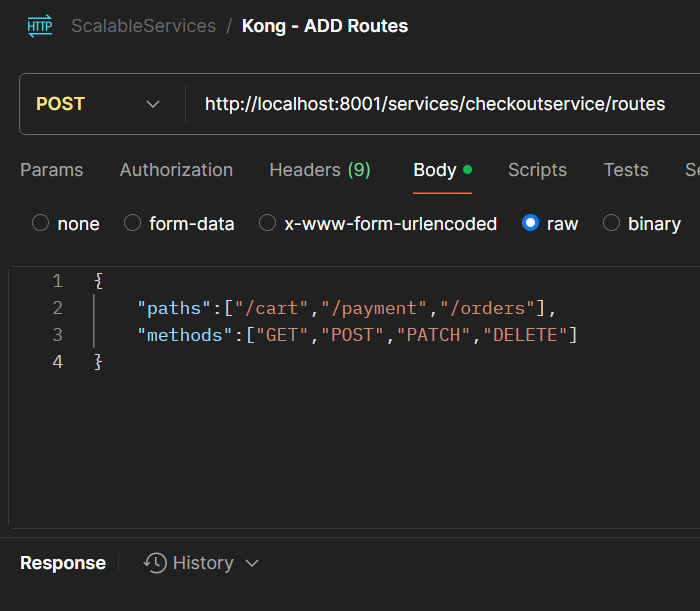
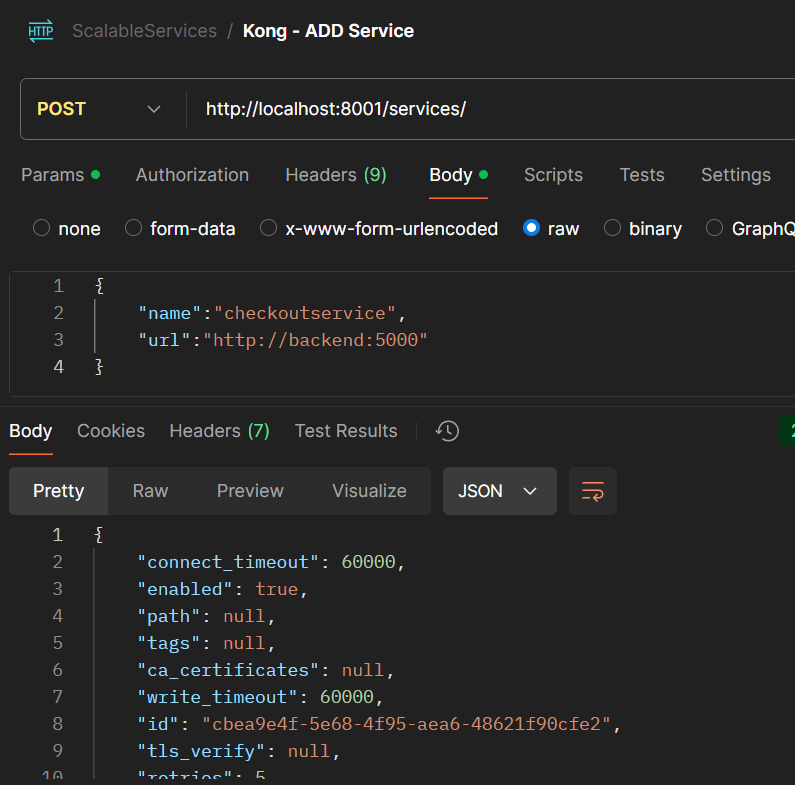
- Logging and Monitoring : Security is bolstered by comprehensive logging and monitoring capabilities. Kong can log all incoming and outgoing requests, providing valuable insights into usage patterns, potential security threats, and operational issues.

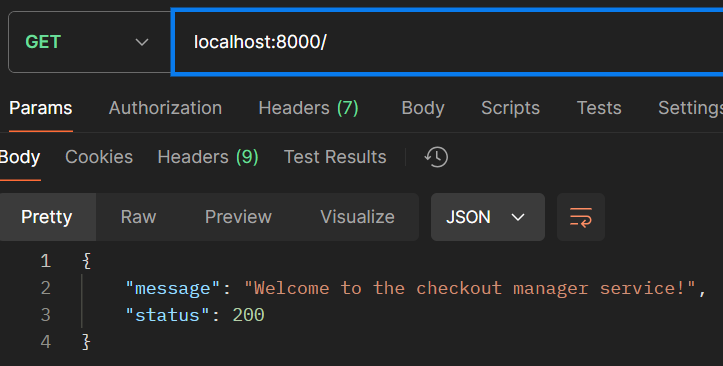
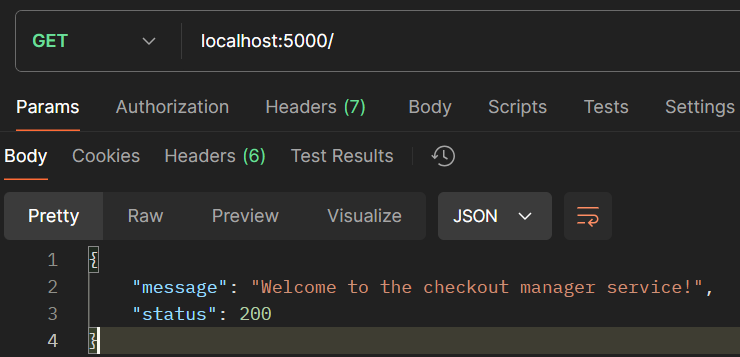
**Implementation with Kong**

By using Kong as the API Gateway, we leverage its robust plugin architecture to easily extend functionality with minimal changes to service code. Kong's ecosystem supports a wide range of plugins for security, traffic control, and analytics, allowing you to tailor the gateway to meet specific business and technical requirements. Additionally, Kong's scalability and performance make it well-suited for handling large volumes of traffic, crucial for scaling modern microservices architectures effectively.

Through these capabilities, Kong's API Gateway not only simplifies the management of microservices but also enhances the overall security, scalability, and resilience of the application infrastructure. This makes it an integral component in modern application architectures, where seamless and secure service interactions are paramount.

By integrating these components, the eCommerce application achieves a robust and scalable communication framework that supports the independent operation of microservices while maintaining security and reliability.



****

Each service is designed to run independently and can be scaled based on demand. This means that during high-traffic periods, such as sales or promotions, specific services like inventory or checkout can be scaled without impacting the rest of the application. This modular design not only supports scalability but also simplifies maintenance and updates.

## **Database Schema**

The database schema for this eCommerce application is designed using MongoDB, a NoSQL database known for its flexibility and scalability. This schema encompasses collections that represent key components of the application: carts, orders, and payments. Each collection is structured to store relevant data in a JSON-like format, which aligns with MongoDB's document-oriented approach.

**Advantages of Using This Schema**

1. **Flexibility**: MongoDB's schema-less nature allows for easy modifications and expansions of the data model without requiring complex migrations. This is particularly advantageous in eCommerce applications where data requirements can frequently change.
2. **Scalability**: The ability to horizontally scale by adding more servers makes MongoDB ideal for handling large volumes of transactions and user data as the application grows.
3. **Embedded Documents**: By using embedded documents (such as cart\_items and order\_items), the schema reduces the need for complex joins, enhancing read performance and allowing for more intuitive data modeling.
4. **Rich Query Capabilities**: MongoDB supports powerful query operations and indexing, which can efficiently handle diverse data retrieval needs, such as searching for orders by date, status, or customer information.
5. **JSON-like Structure**: The JSON-like structure of MongoDB documents aligns closely with the data format used in web applications (JSON), simplifying data interchange between the database and application layers.

By using MongoDB for this schema, the application benefits from a robust, scalable, and flexible data management system that aligns well with modern web application needs, particularly those of dynamic and rapidly evolving eCommerce platforms.

1. Checkout Service Schema :

|  |  |  |
| --- | --- | --- |
| carts | Orders | Payments |
| {  "\_id": {"$oid": "673b57c5840b8e4cc11419d6"},  "cart\_id": 1,  "cart\_items": [  {  "item\_id": 5678,  "name": "Wireless Mouse",  "quantity": 2,  "price\_per\_unit": 25.99  },  {  "item\_id": 6789,  "name": "Keyboard",  "quantity": 1,  "price\_per\_unit": 49.99  }  ],  "cart\_total": 101.97,  "cart\_date": "2024-11-15T12:34:56Z" } | {  "\_id": {"$oid": "672cb4abfc2124556d60492f"},  "order\_id": 1,  "customer\_name": "John Doe",  "customer\_email": "john.doe@example.com",  "customer\_phone": "+1234567890",  "order\_items": [  {  "product\_id": "prod001",  "product\_name": "Product 1",  "quantity": 2,  "price": 50  },  {  "product\_id": "prod002",  "product\_name": "Product 2",  "quantity": 1,  "price": 30  }  ],  "order\_total": 130,  "order\_date": {"$date": "2024-11-07T18:08:03.600Z"},  "order\_status": "pending",  "order\_address": {  "street": "\*\*\*\*\*\*\*\*\*\*\*",  "city": "Anytown",  "state": "Anystate",  "zipcode": "12345",  "country": "USA"  },  "order\_payment": {  "method": "credit\_card",  "transaction\_id": "txn\_123456"  },  "order\_shipment": {  "carrier": "UPS",  "shipment\_id": "ship\_123456"  },  "order\_tracking": "tracking\_123456",  "order\_notes": "Please deliver between 9 AM and 5 PM" } | {  "\_id": {"$oid": "673b5ec092ca117137d42d5b"},  "payment\_id": 1,  "payment\_amount": 150,  "payment\_date": {"$date": "2024-11-18T21:05:27.995Z"},  "payment\_status": "Completed",  "payment\_method": "Credit Card",  "payment\_notes": "Payment processed successfully for order #1234" } |

2. Inventory Service Schema:

|  |  |  |  |
| --- | --- | --- | --- |
| **FIELD** | **TYPE** | **REQUIRED** | **DESCRIPTION** |
| id | String | Yes | Unique identifier for the product |
| name | String | Yes | Name of the product |
| quantity | Number | Yes | Number of items in stock |
| price | Number | Yes | Price of the product |
| description | String | Yes | Description of the product |
| available | Boolean | Yes | Whether the product is available |
| createdAt | Date | No | Automatically added timestamp when the document is created |
| updatedAt | Date | No | Automatically added timestamp when the document is updated |

3. **Blogs/ Review Service DB Schema**

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Type** | **Required** | **Description** |
| id | Long | Yes | Unique identifier for the blog |
| title | String | Yes | Name of the product (e.g., "Electric Kettle") |
| content | String | Yes | Review about the product |
| author | String | Yes | Author of the review |
| productId | Long | Yes | Id of the product which is reviewed |

## **API Endpoints**

1. Checkout Service

|  |  |  |  |
| --- | --- | --- | --- |
| Http Verb | Endpoint | Statu code | Description |
| GET | / | 200 | Shows the status/health of the checkout service. |
| GET | /order/<orderid> | 200  404 | Get order details by orderid if order exists. |
| POST | /orders | 200  400 | Create order entry and return orderid, takes order obj as input. |
| PATCH | /order/<orderid> | 200  404 | Update order entry with orderid, takes in updated order obj as input. |
| DELETE | /order/<orderid> | 200  404 | Delete an order by orderid. |
| GET | /orders | 200  400 | Get all orders in the db. |
| POST | /cart | 200  400 | Add items to cart, response body has the cart obj(cart id, cart items, total) |
| GET | /cart/<cartid> | 200  404 | Fetch items in a cart with cartid. |
| PATCH | /cart/<cartid> | 200  404 | Update cart entry with cartid, modifies the items in the cart. |
| DELETE | /cart/<cartid> | 200  404 | Delete cart/all items in the cart. |
| POST | /payment | 200  400 | Add payment details of an order. |
| GET | /payment/<id> | 200  404 | Get status of a payment. |
| POST | /services/ | 200 | Modify/add services of api gateway (kong) |
| POST | /routes/ | 200 | Modify/add routes for api gateway (Kong) |

2. Inventory Service

|  |  |  |  |
| --- | --- | --- | --- |
| **HTTP VERB** | **ENDPOINT** | **STATUS CODE** | **DESCRIPTION** |
| GET | /api/inventory/products | 200: Ok  404: Not Found | Returns back all the products in the inventory database |
| POST | /api/inventory/products | 201: Created  400: Bad Request | Add a new product in the inventory database |
| GET | /api/inventory/products/{id} | 200: Ok  404: Not Found | Returns a specific product that matches the supplied id from the inventory database |
| PUT | /api/inventory/products/{id} | 200: Ok  400: Bad Request | Update a specific product that matched with the supplied id in the inventory database using the JSON Body |
| DELETE | /api/inventory/products/{id} | 200: Ok  404: Not Found | Deletes a specific product that matches with the supplied id from the inventory database |

**API Endpoints for Blog service:**

|  |  |  |  |
| --- | --- | --- | --- |
| HTTP VERB | ENDPOINT | STATUS CODE | DESCRIPTION |
| GET | api/blogs | 200: Ok  404: Not Found | Returns all the blogs |
| POST | api/blogs | 200: Ok  400: Bad Request | Adds a new blog in the H2 DB |
| GET | api/blogs/{id} | 200: Ok  404: Not Found | Returns the blog with the specified id |
| PUT | api/blogs/{id} | 200: Ok  400: Bad Request | Updates the blog with the specified id |
| DELETE | api/blogs/{id} | 200: Ok  404: Not Found | Deletes the blog with the specified id |

## **API Endpoints for User Service**

*User Account Endpoints*

This group of endpoints provides functionalities to manage user accounts, including creation, authentication, retrieval, updating, and deletion. The "Verify User" endpoint allows the verification of a user's account.

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Endpoint** | **Method** | **Description** |
| **1** | **/api/user** | POST | Creates a new user account. |
| **2** | **/api/user** | GET | Retrieves a list of all user accounts. |
| **3** | **/api/user/login** | POST | Authenticates a user based on provided credentials. |
| **4** | **/api/user/{id}** | GET | Retrieves detailed information about a user by ID. |
| **5** | **/api/user/{id}** | PUT | Updates user details by ID. |

* User Registration
  + Name
  + Password
  + Email
  + Mobile Number
  + Role
  + Verification
* User Verification
  + Email
  + Token

A diagram of a computer program

Description automatically generated

## 

## 

## Postman Samples

**1. Checkout Service : Sample Request & Responses**

|  |  |  |
| --- | --- | --- |
| Endpoint | Body | Response |
| GET / | N/A | { "message": "Welcome to the checkout manager service!",  "status": 200 } |
| GET /cart/  <cartid> | N/A | {  "message": "Cart not found!",  "status": 404  } |
| PATCH  /cart/1 | {  "cart\_items": [  {  "item\_id": 5678,  "name": "Wireless Mouse",  "quantity": 3,  "price\_per\_unit": 25.99  }],  "cart\_total": 101.97,  "cart\_date": "2024-11-15T12:34:56Z"  } | {  "message": "Cart updated!",  "status": 200  } |
| POST /orders | {  "customer\_name": "John Doe",  "customer\_email": "sathishkantamsetti01@gmail.com",  "customer\_phone": "+1234567890",  "order\_items": [  {  "product\_id": "Prd01",  "product\_name": "Wireless Mouse",  "quantity": 2,  "price": 50  },  {  "product\_id": "Prod02",  "product\_name": "Keyboard",  "quantity": 1,  "price": 30  }  ],  "order\_total": 130,  "order\_status": "Success",  "order\_address": {  "street": "123 Main St",  "city": "Anytown",  "state": "Anystate",  "zipcode": "54312",  "country": "India"  },  "order\_payment": {  "method": "credit\_card",  "transaction\_id": "txn\_1325456"  }  } | {  "message": "Order placed successfully!",  "order\_id": "673f0315748964c3eeed4a63",  "status": 200  } |
| DELETE  /order/1 | N/A | {  "message": "Order deleted!",  "status": 200  } |
| POST  /payment | {  "payment\_amount": 150.00,  "payment\_date": "2024-11-15T14:23:00Z",  "payment\_status": "Completed",  "payment\_method": "Credit Card",  "payment\_notes": "Payment processed successfully for order #1234"  } | {  "message": "Payment Added successfully!",  "payment\_id": "673f04ee11c6488ac7636129"  } |

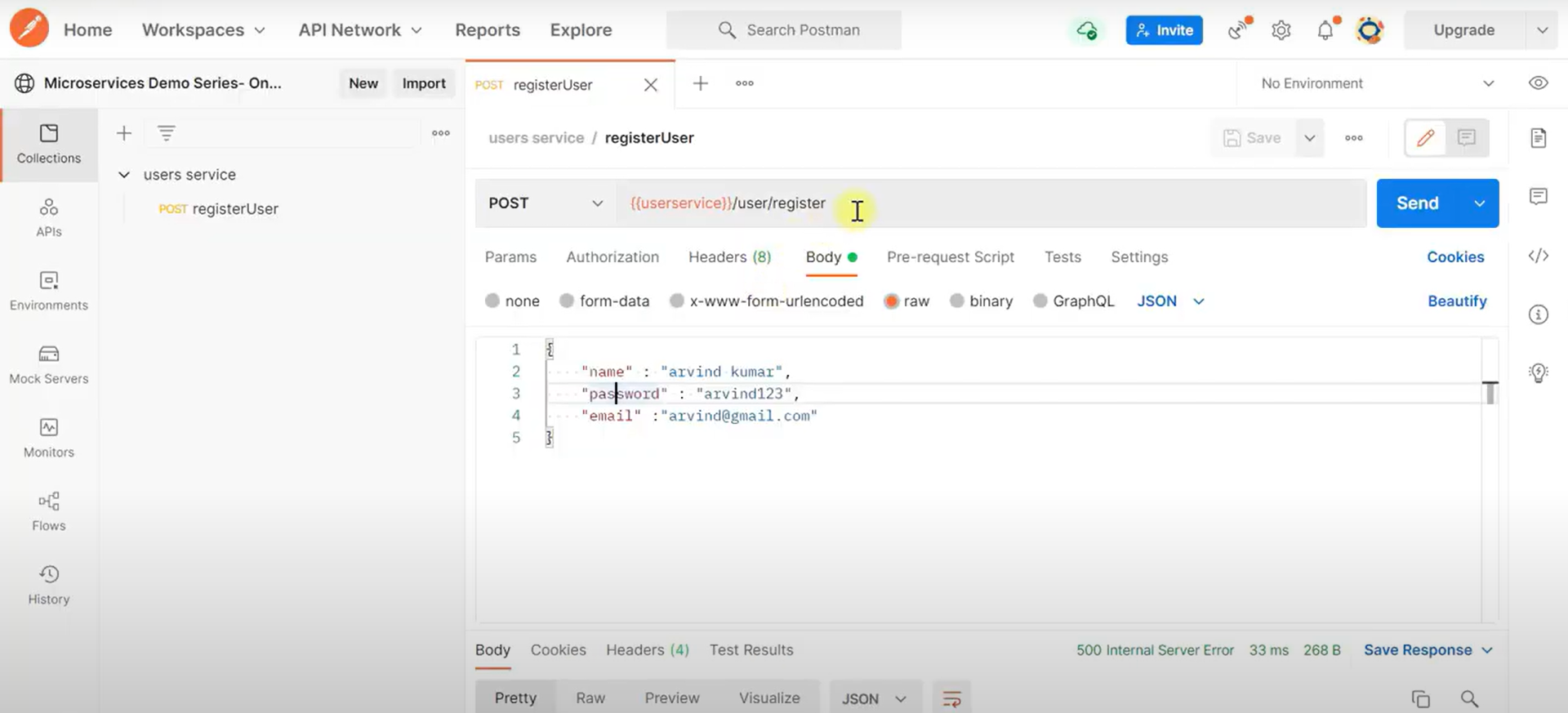
2. Inventory Service: Sample Responses

|  |  |  |
| --- | --- | --- |
| ENDPOINT | HTTP BODY | RESPONSE |
| GET /api/inventory/products | N/A | [ {  "id": "6721d5534e7be3e5f0c1b1a2",  "name": "Electric Kettle",  "quantity": 9,  "price": 39.99,  "description": "Fast boiling electric kettle with automatic shut-off.",  "available": true  },  {  "id": "6721d5584e7be3e5f0c1b1a4",  "name": "Yoga Mat",  "quantity": 50,  "price": 25.99,  "description": "Non-slip yoga mat for comfort and stability during practice.",  "available": true  }, ] |
| POST /api/inventory/products | {  "name": "Canvas Shoes",  "quantity": 10,  "description": "Sturdy shoes for your everday use.",  "price": 29.99,  "available": true  } | {  "id": "6738c4f5522b9841d6c8b88f",  "name": "Canvas Shoes",  "quantity": 10,  "price": 29.99,  "description": "Sturdy shoes for your everday use.",  "available": true  } |
| GET /api/inventory/products/6738c4f5522b9841d6c8b88f | N/A | {  "id": "6738c4f5522b9841d6c8b88f",  "name": "Canvas Shoes",  "quantity": 10,  "price": 29.99,  "description": "Sturdy shoes for your everday use.",  "available": true  } |
| PUT /api/inventory/products/6721d5534e7be3e5f0c1b1a2 | {  "id": "6721d5534e7be3e5f0c1b1a2",  "name": "Electric Kettle",  "quantity": 20,  "price": 39.99,  "description": "Fast boiling electric kettle with automatic shut-off.",  "available": true  } | {  "message": "Item updated successfully"  } |
| DELETE /api/inventory/products/6738c4f5522b9841d6c8b88f | N/A | {  "message": "Item deleted successfully"  } |

3. **Blog service: Sample responses**

|  |  |  |
| --- | --- | --- |
| ENDPOINT | HTTP BODY | RESPONSE |
| GET /api/blogs |  | {  "id": 1,  "title": "Product name 1",  "content": "Review for the product 1",  "author": "Gouri Remesh",  "productId": 1001  },  {  "id": 2,  "title": "Product name 4",  "content": "Sample Review",  "author": "Gouri Remesh",  "productId": 1002  }] |
| POST /api/blogs | {  "title": "Product name 8",  "content": "This is my review for another product",  "author": "Gouri Remesh",  "productId": 6000  } | {  "id": 6,  "title": "Product name 8",  "content": "This is my review for another product",  "author": "Gouri Remesh",  "productId": 6000  } |
| GET /api/blogs/1 |  | {  "id": 1,  "title": "Product name 3",  "content": "This is my review for another product",  "author": "Gouri Remesh",  "productId": null  } |
| PUT /api/blogs/2 | {  "id": 2,  "title": "Product name 4",  "content": "Updated review",  "author": "Gouri Remesh",  "productId": 4000  } | {  "id": 2,  "title": "Product name 4",  "content": "Updated review",  "author": "Gouri Remesh",  "productId": 4000  } |
| DELETE  /api/blogs/2 |  | Blog deleted successfully |

4. **User service: Sample Responses**



A screenshot of a computer

Description automatically generated

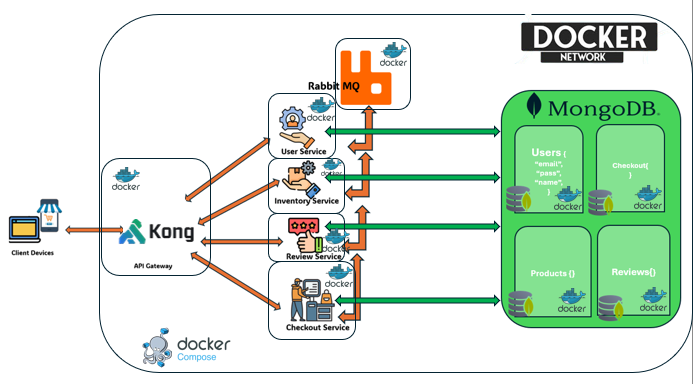
# Deployment

The deployment of the eCommerce application is orchestrated using Docker Compose, which simplifies the management of multiple services running in Docker containers. This setup ensures a consistent and isolated environment for each component, enhancing both scalability and maintainability.

**Microservices Architecture**: The application is divided into several microservices, each encapsulated within its own Docker container. These include the User, Inventory, Review, and Checkout services. This architecture promotes modular development, allowing each service to be developed, deployed, and scaled independently according to its specific needs.

**Kong API Gateway**: Kong serves as the entry point for all client requests. Operating as a reverse proxy, it manages and secures access to the underlying services. Kong handles routing based on predefined rules, forwarding requests from clients to the appropriate microservice while enforcing security policies such as authentication and rate limiting.

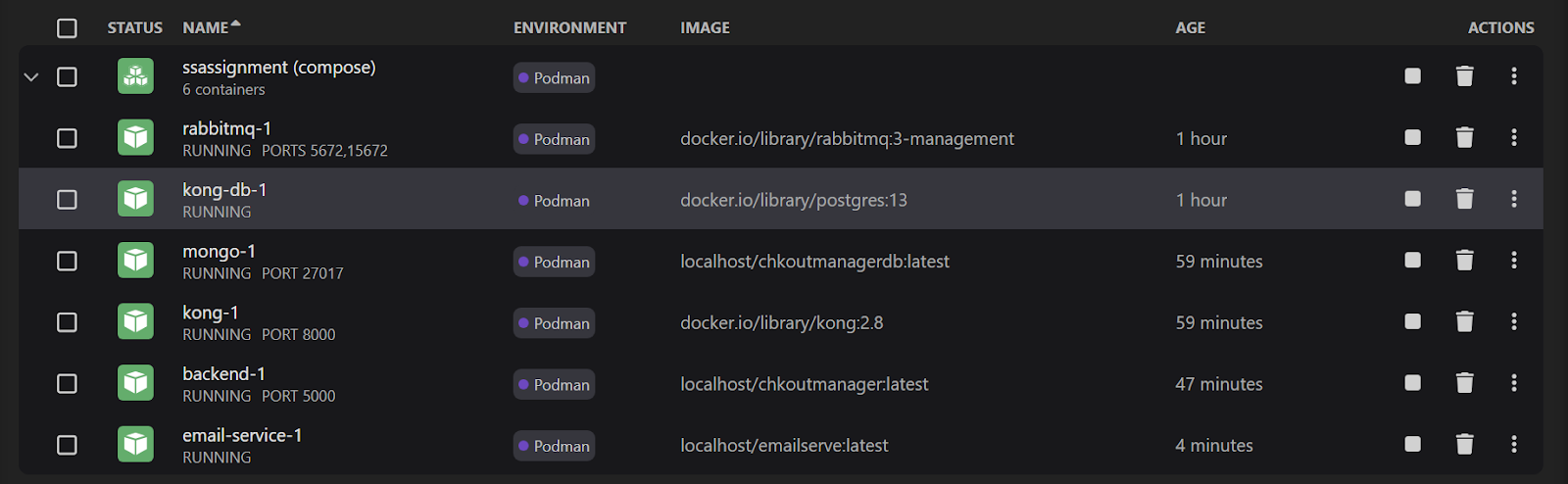
**RabbitMQ for Communication**: RabbitMQ is deployed as a message broker to facilitate asynchronous communication between microservices. This setup decouples service interactions, allowing them to operate independently and enhancing the system's overall resilience. For instance, when an order is placed, the Checkout service sends a message to RabbitMQ, which is then consumed by the Email service to trigger order confirmation emails.



**Dedicated MongoDB Instances**: Each microservice has its own MongoDB database instance running in a separate Docker container. This approach ensures data isolation, allowing each service to manage its data schema independently. This separation of data storage enhances security and simplifies data management, as each service can scale its database according to its demands without affecting others.

## **Deployment with Docker/podman Compose**

* **Container Orchestration**: Docker Compose is used to define and run the application stack. A single docker-compose.yml file specifies all the services, networks, and volumes, streamlining the deployment process. This configuration allows for easy scaling and management of services, ensuring that all components are correctly networked and configured to communicate with each other.
* **Networking**: All containers are part of a shared network, allowing seamless communication between services. This network configuration ensures that each service can reach others by their service names, facilitating smooth interactions and data exchange.
* **Scalability and Portability**: By containerizing each component, the application can be easily scaled horizontally. New instances of a service can be spun up as needed to handle increased loads. Additionally, the Docker-based deployment ensures portability, enabling the application to be easily moved and deployed across different environments without compatibility issues.



## 

## Transaction Flow

**Step 1: User Service**

The interaction begins with the User Service, where user authentication is performed, and user details are retrieved. A login request is initiated using Postman, with user credentials being provided.

**Step 2: Product Listing and Details**

The process continues with the Inventory Service, where available products are explored. A request is sent via Postman to retrieve product listings, filtered by category or other parameters. A comprehensive list of products is provided in response, including details such as name, price, and availability. A specific product is then selected, and another request is sent to obtain detailed information about it. This allows products of interest to be browsed and selected, with all necessary details provided for making an informed purchasing decision.

**Step 3: Product Reviews (Blogs service)**

With product details in hand, the Review Service is accessed to gather customer feedback. A request is made using Postman for reviews of the selected product. The response includes user-generated reviews and ratings, offering insights into product performance and satisfaction from previous buyers. This helps informed decisions to be made by considering the experiences of other customers, enhancing transparency and reliability in the shopping process.

**Step 4: Cart Management, Order Placement, and Payment**

The transaction is completed using the Checkout Service. Initially, the selected product is added to the cart by sending a POST request with product details via Postman. Once added, an order is placed by sending another request, which involves confirming the items in the cart and finalizing delivery details. After placing the order, the payment process is initiated by sending a payment request, capturing payment details such as method and amount. This step concludes the transaction, transitioning from interest and selection to purchase and payment, thereby completing the eCommerce transaction cycle.

**Step 5: Email Confirmation and Inventory Update**

Following the order placement, the Email Service is triggered to send an email confirmation to the customer. This is facilitated by the order message placed in the queue, which the Email Service consumes to generate a confirmation email. Concurrently, the Inventory Service updates inventory levels based on the order details. The product information is read from the queue, allowing the Inventory Service to adjust stock quantities accordingly. This step ensures that both the customer is notified of their order status and inventory records are accurately maintained.