E-commerce Responsive App

Tech Stack : Python , Flask Html , CSS , JavaScript, Mysql (SQL) , MongoDB (NoSQL)

Introduction

- **Project Overview:** The project aims to develop a comprehensive e-commerce web application enabling third-party companies to list and manage their products, similar to platforms like Amazon. Key features include robust user authentication, secure transactions, and efficient database management systems to support seamless user interactions.

- **Objectives:** The primary objectives include creating a user-friendly interface for buyers and sellers, implementing secure data handling practices, and optimizing performance to accommodate large-scale operations. The project spans two phases: Phase 1 focuses on designing a MySQL relational database, while Phase 2 explores the integration of a NoSQL database to enhance scalability and performance.

- **Scope:** The scope encompasses designing and implementing functionalities for user authentication, product management, shopping cart and checkout processes, order management, and user profile management. Non-functional aspects such as database design, security measures, performance optimization, and user interface design are also crucial components of the project.

Business Goals :

- **User-Friendly Interface:** Ensure a seamless user experience with easy navigation and interaction.
- **Secure Transactions:** Implement robust security measures to protect user data and transactions.
- **Scalable Architecture:** Design the system to accommodate growth in users and data.
- **Efficient Data Management:** Utilize normalized database schemas to maintain data integrity and support efficient queries.

- **Comprehensive Analytics:** Provide insightful analytics to track user behavior and market trends.

Phase 1: MySQL Relational Database

1.1 Functional Requirements

User Authentication and Registration

- functionalities covered are registration, login, password management, and role-based access control (RBAC).
- **Flask Application Setup**:
    - Running run.py file (python run.py).
    - Include configuration details such as SECRET_KEY and SQLALCHEMY_DATABASE_URI settings.

User Model

- **User Model**:
    - Describe the SQLAlchemy User model.
    - Outline fields such as id, username, email, and password_hash.
    - methods like `set_password` for password hashing and `check_password` for password verification.

- **Registration Process**:
    - a sample registration form HTML snippet.
    - registration route implementation in Flask use wtf-forms library for validation

- **Login Process**:
    - users can log in to the application from login on top right
    - sample login form HTML snippet.
    - login route implementation in Flask.
    - session management using Flask's `session`.

Role-Based Access Control (RBAC) : Authorization

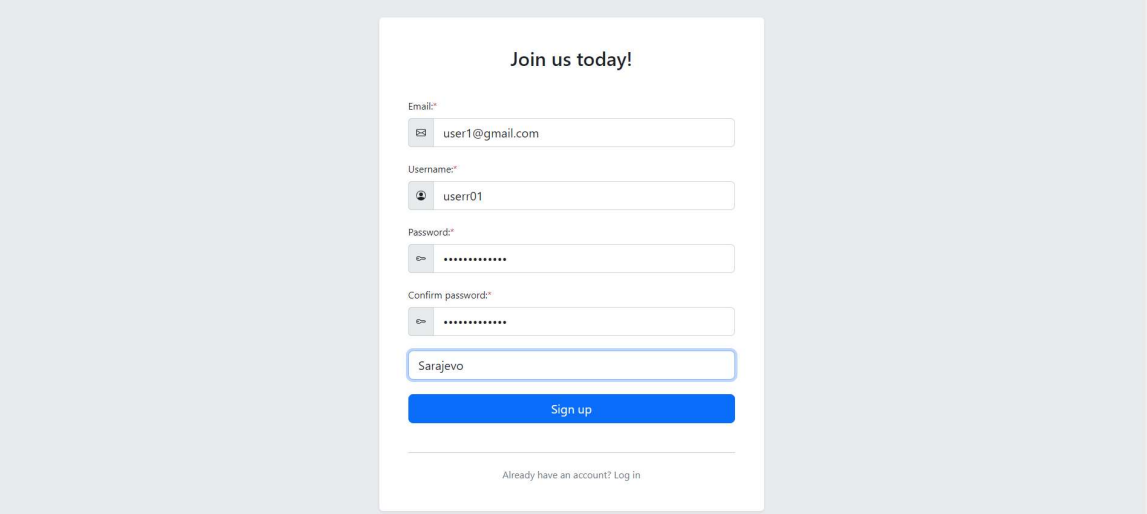
- **RBAC Implementation**:
    - roles (e.g., admin, user , vendors) and their permissions.
    - Role and UserRole models in SQLAlchemy has multiple views based on their role in the app.
    - roles can be assigned to users and managed in the application using API gateway hit to /register and / login route.

Screenshots and User Experience

- **Screenshots**:
    - Include screenshots of the registration and login screens.

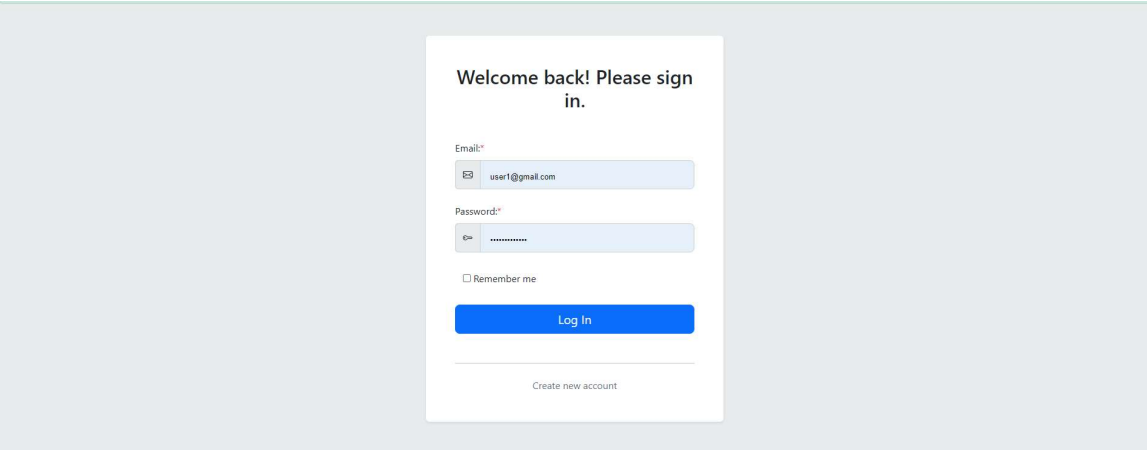
Mockup Descriptions

- **User Registration/Login Screen**: Fields for username, email, password, and login button.
- **Product Listings Screen**: List of products with images, names, prices, and categories.
- **Product Details Screen**: Detailed view of a selected product with image, description, price, and add-to-cart button.
- **Shopping Cart Screen**: List of products added to the cart with quantities and total price.
- **Checkout Screen**: Order summary, payment options, and place order button.
- **Order History Screen**: List of past orders with order details and statuses.
- **User Profile Screen**: User information, order history, and account settings.



- Registration form

- Log after the user's account created



- Login logs

Schema Design :

Designing a MySQL database schema for storing user information involves careful planning to ensure data consistency, support efficient queries, and accommodate future scalability. Here's a detailed explanation of how to approach schema design for user information:

Steps for Schema Design

1. Identify Entities and Attributes

- **User Entity**: The primary entity in your schema.
  - **Attributes**: Typically include `user_id`, `username`, `email`, `password_hash`, and any other relevant user details.

2. Define Relationships

- **User Roles**:
  - If implementing Role-Based Access Control (RBAC), define roles and their relationships to users. This involves creating tables like `roles`, `user_roles`, etc.

```
CREATE TABLE roles (

    role_id INT PRIMARY KEY AUTO_INCREMENT,

    role_name VARCHAR(50) NOT NULL UNIQUE

);



CREATE TABLE user_roles (
```

```
    user_id INT,

    role_id INT,

    PRIMARY KEY (user_id, role_id),

    FOREIGN KEY (user_id) REFERENCES users(user_id),

    FOREIGN KEY (role_id) REFERENCES roles(role_id)

);
```

- **Schema Design :** The MySQL database schema includes tables for storing user information, structured to maintain data consistency and support efficient query execution.

Database Creation and Selection

```
CREATE DATABASE IF NOT EXISTS flaskmarketdb;

USE flaskmarketdb;
```

Creates a new database named `flaskmarketdb` if it doesn't exist and selects it for use.

User Table

```
CREATE TABLE IF NOT EXISTS user (

    user_id INT NOT NULL AUTO_INCREMENT,

    username VARCHAR(45) NOT NULL,

    password VARCHAR(60) NOT NULL,

    email VARCHAR(150) NOT NULL,

    city VARCHAR(100) NOT NULL,
```

```
    image VARCHAR(255) NOT NULL DEFAULT 'default.jpg',

    balance FLOAT NULL DEFAULT 1000,

    role ENUM('user', 'admin') NULL DEFAULT 'user',

    location VARCHAR(255) NULL DEFAULT NULL,

    PRIMARY KEY (user_id),

    UNIQUE INDEX username (username),

    UNIQUE INDEX email (email),

    INDEX idx_user_location (location)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Stores user information for the application.

Attributes:

- `user_id`: **Unique identifier for each user.**
- `username`, `password`, `email`: **Credentials for authentication.**
- `city`, `location`: **User's city and location.**
- `image`: **Profile image path.**
- `balance`: **User's account balance.**
- `role`: **Defines user role (either 'user' or 'admin').**

Indexes:

- `username` **and** `email`: **Ensures usernames and emails are unique for each user.**
- `idx_user_location`: **Index for optimizing queries based on user location.**

- **Product Management :** Sellers register through a streamlined process, utilizing a detailed product attributes schema to define product listings. Screenshots of the seller registration form and product attributes schema highlight the setup.

Product Table

```
    CREATE TABLE IF NOT EXISTS product (
```

```
          product_id INT NOT NULL AUTO_INCREMENT,

          user_id INT NOT NULL,

          title VARCHAR(200) NOT NULL,

          image TEXT NULL DEFAULT NULL,

          description TEXT NULL DEFAULT NULL,

          price FLOAT NOT NULL,

          created_at TIMESTAMP NULL DEFAULT CURRENT_TIMESTAMP,

          category ENUM('Phones', 'Laptops', 'Gadgets', 'Other') NULL
DEFAULT 'Other',

          user_name VARCHAR(255) NULL DEFAULT NULL,

          avg_review FLOAT NULL DEFAULT NULL,

          location VARCHAR(255) NULL DEFAULT NULL,

          PRIMARY KEY (product_id),

          INDEX user_id (user_id),

          INDEX user_name (user_name),

          INDEX fk_product_user_location (location),

          CONSTRAINT fk_product_user_location FOREIGN KEY (location)
REFERENCES user(location),

          CONSTRAINT product_ibfk_1 FOREIGN KEY (user_id) REFERENCES
user(user_id),

          CONSTRAINT product_ibfk_2 FOREIGN KEY (user_name) REFERENCES
user(username)

      ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci;
```

Attributes:

- `product_id`: Unique identifier for each product.
- `user_id`: Foreign key referencing the seller (user) of the product.
- `title`, `image`, `description`: Details about the product.
- `price`: Price of the product.
- `created_at`: Timestamp when the product was added.
- `category`: Categorization of the product (e.g., Phones, Laptops).
- `user_name`: Denormalized field for quick access to seller's username.
- `avg_review`: Average review rating for the product.
- `location`: Location of the user selling the product.

Indexes and Constraints:

- `user_id`, `user_name`, `location`: Indexes for optimizing queries related to users and their products.
- Foreign key constraints ensure referential integrity with the `user` table.

- **Product Listings:** Users can search and browse products by categories and keywords, with detailed product pages providing comprehensive information. Screenshots demonstrate the product listing and detail pages.

- **Shopping Cart and Checkout:** Cart management functionalities enable users to add, remove, and manage products, with a secure checkout process integrating payment gateways. Screenshots depict the shopping cart interface and checkout steps.

  Cart Table

```
CREATE TABLE IF NOT EXISTS cart (

    id INT NOT NULL AUTO_INCREMENT,

    product_id INT NOT NULL,

    user_id INT NOT NULL,

    PRIMARY KEY (id),

    INDEX product_id (product_id),

    INDEX user_id (user_id),
```

```
    CONSTRAINT cart_ibfk_1 FOREIGN KEY (product_id) REFERENCES
product(product_id) ON DELETE CASCADE,

    CONSTRAINT cart_ibfk_2 FOREIGN KEY (user_id) REFERENCES
user(user_id) ON DELETE CASCADE

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Stores the user's shopping cart items.

Attributes:

- `id`: Unique identifier for each cart item.
- `product_id`, `user_id`: Foreign keys referencing the product and user.

Indexes and Constraints:

- Indexes on `product_id` and `user_id` for efficient querying.
- Foreign key constraints ensure referential integrity with the `product` and `user` tables, with cascade deletion for cart items if products or users are deleted.

Comment Table

```
CREATE TABLE IF NOT EXISTS comment (

    comment_id INT NOT NULL AUTO_INCREMENT,

    user_id INT NOT NULL,

    content TEXT NOT NULL,

    created_at TIMESTAMP NULL DEFAULT CURRENT_TIMESTAMP,

    product_id INT NOT NULL,

    username VARCHAR(255) NULL DEFAULT NULL,

    PRIMARY KEY (comment_id),

    INDEX product_id (product_id),

    INDEX user_id (user_id),
```

```
    CONSTRAINT comment_ibfk_1 FOREIGN KEY (product_id) REFERENCES
product(product_id),

    CONSTRAINT comment_ibfk_2 FOREIGN KEY (user_id) REFERENCES
user(user_id)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Stores comments made by users on products.

Attributes:

- `comment_id`: Unique identifier for each comment.
- `user_id`: Foreign key referencing the user who made the comment.
- `content`: Text content of the comment.
- `created_at`: Timestamp when the comment was posted.
- `product_id`: Foreign key referencing the product being commented on.
- `username`: Denormalized field for quick access to commenter's username.

Indexes and Constraints:

- Indexes on `product_id` and `user_id` for efficient querying.
- Foreign key constraints ensure referential integrity with the `product` and `user` tables.

- **Order Management:** Users can view order histories, while sellers manage order statuses through dedicated interfaces. Screenshots of the order history page and seller dashboard illustrate these features.
- **User Profiles:** User profile pages display order histories and contact details, with seller profiles managing product listings and inventory. Screenshots showcase the user and seller profile interfaces.

Review Table

```
    CREATE TABLE IF NOT EXISTS review (

        review_id INT NOT NULL AUTO_INCREMENT,

        product_id INT NOT NULL,
```

```
          user_id INT NOT NULL,

          rating INT NOT NULL,

          created_at DATETIME NULL DEFAULT CURRENT_TIMESTAMP,

          username VARCHAR(255) NULL DEFAULT NULL,

          PRIMARY KEY (review_id),

          INDEX product_id (product_id),

          INDEX user_id (user_id),

          CONSTRAINT review_ibfk_1 FOREIGN KEY (product_id) REFERENCES
product(product_id),

          CONSTRAINT review_ibfk_2 FOREIGN KEY (user_id) REFERENCES
user(user_id)

      ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci;
```

Stores reviews submitted by users for products.

- **Attributes:**
  - `review_id`: Unique identifier for each review.
  - `product_id`: Foreign key referencing the reviewed product.
  - `user_id`: Foreign key referencing the user who submitted the review.
  - `rating`: Numerical rating given to the product.
  - `created_at`: Timestamp when the review was submitted.
  - `username`: Denormalized field for quick access to reviewer's username.
- **Indexes and Constraints:**
  - Indexes on `product_id` and `user_id` for efficient querying.
  - Foreign key constraints ensure referential integrity with the `product` and `user` tables.

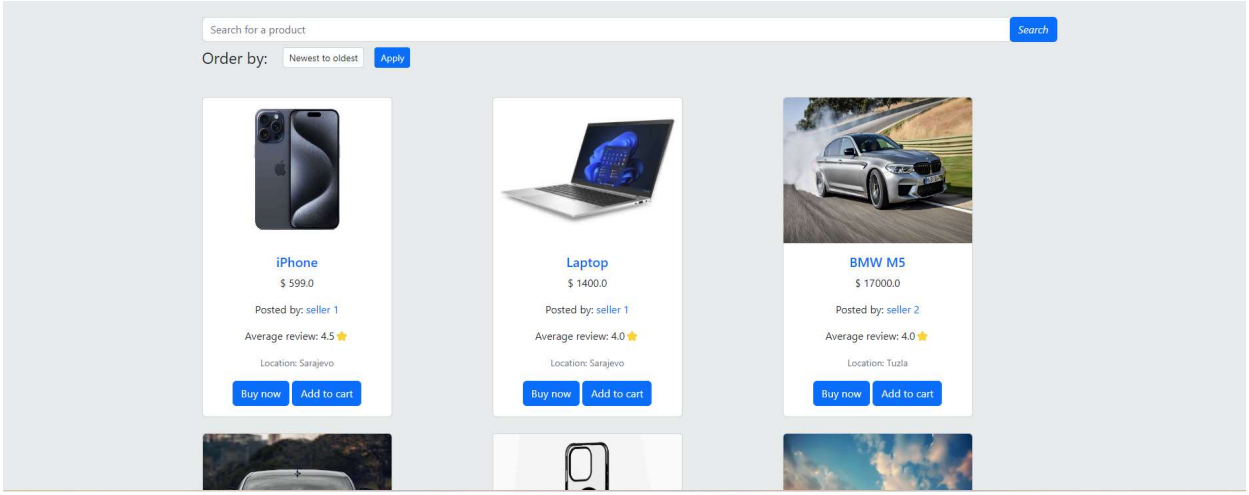
Triggers

- **Triggers are used to automatically populate certain fields or perform actions upon specific database events.**
  - before_product_insert: Populates `user_name` and `location` fields in the `product` table before inserting a new product.

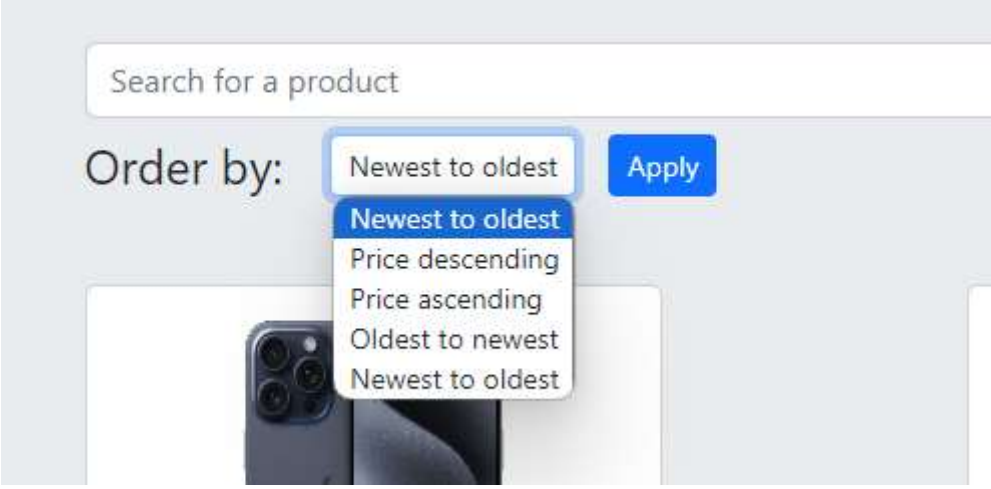
- trigger_populate_product_location: Ensures the `location` field in the `product` table is populated based on the seller's location.
- before_comment_insert: Populates the `username` field in the `comment` table based on the user who made the comment.
- before_review_insert: Populates the `username` field in the `review` table based on the user who submitted the review.
- update_avg_review: Updates the `avg_review` field in the `product` table after a new review is inserted.

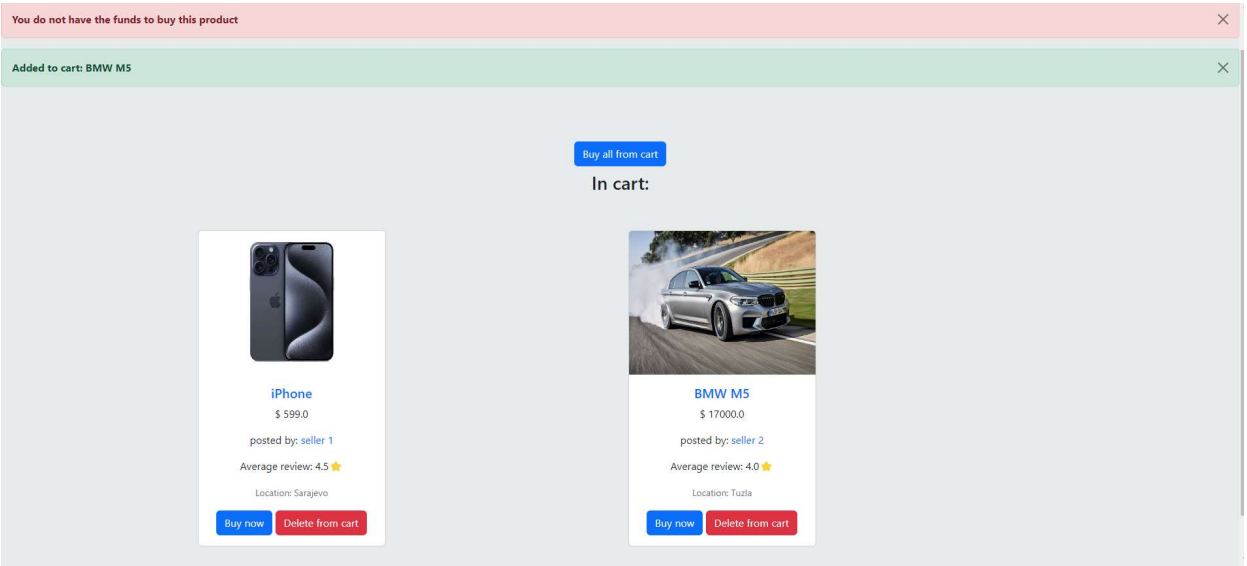
8. Data Insertion Examples

- Examples of data insertion into tables for demonstration.
- User and Product Data: Inserts initial data into the `user` and `product` tables to populate them with sample users and products.

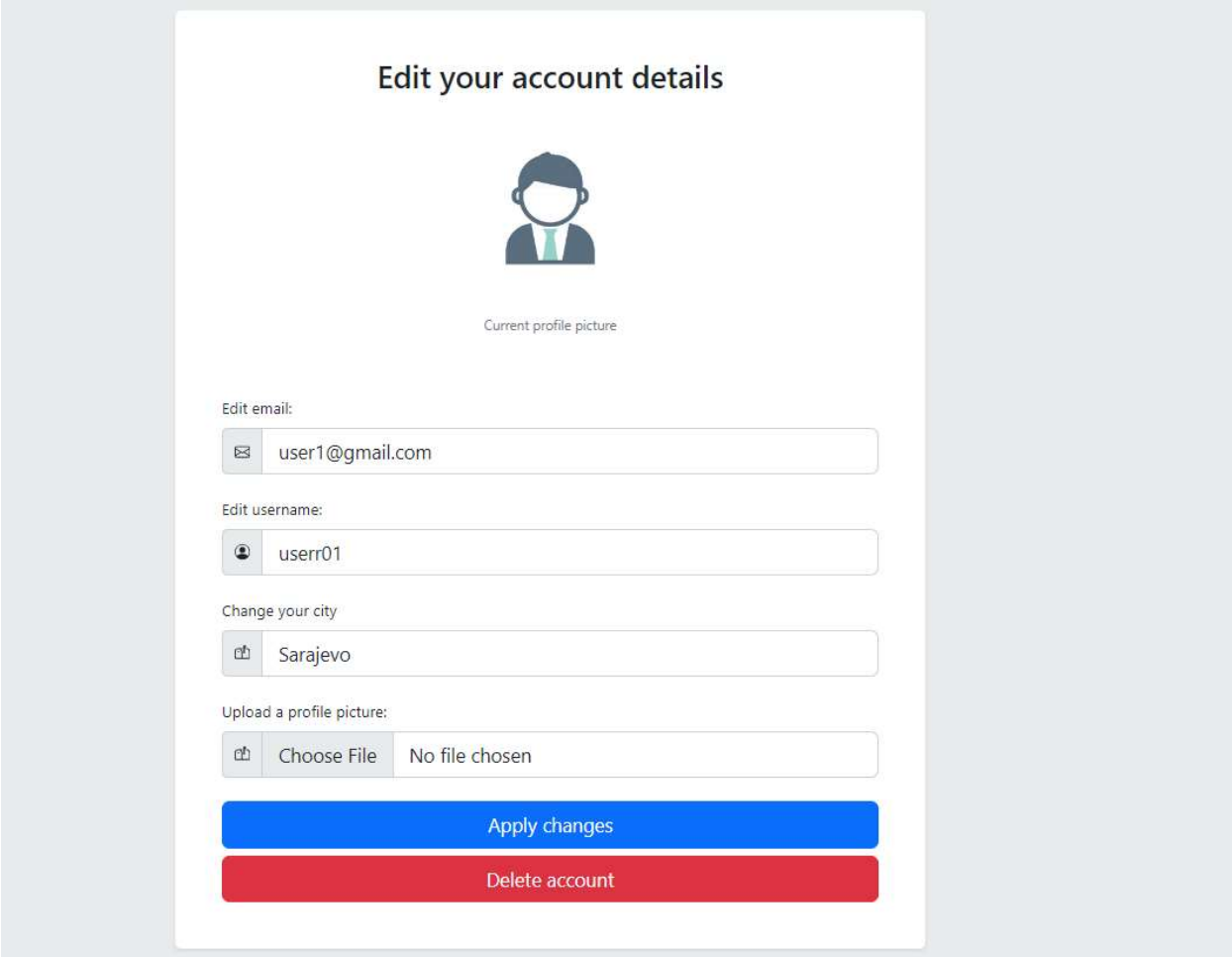


Marketplace with different products by all vendors



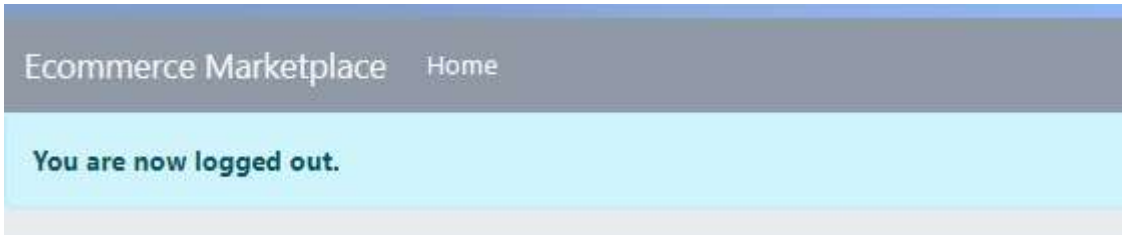


Adding items from multiple vendors



User Profiling

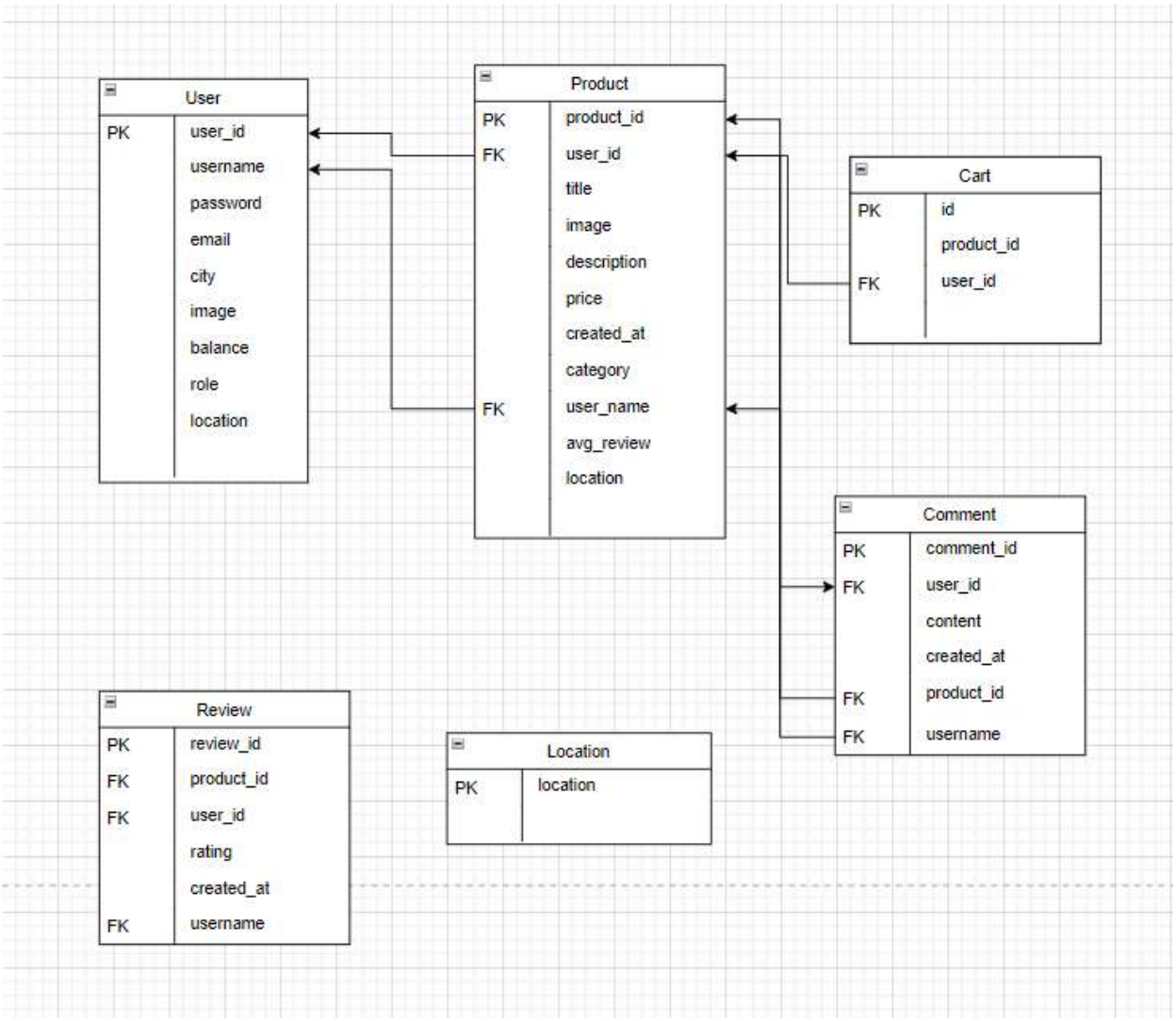
For a vendor product addition



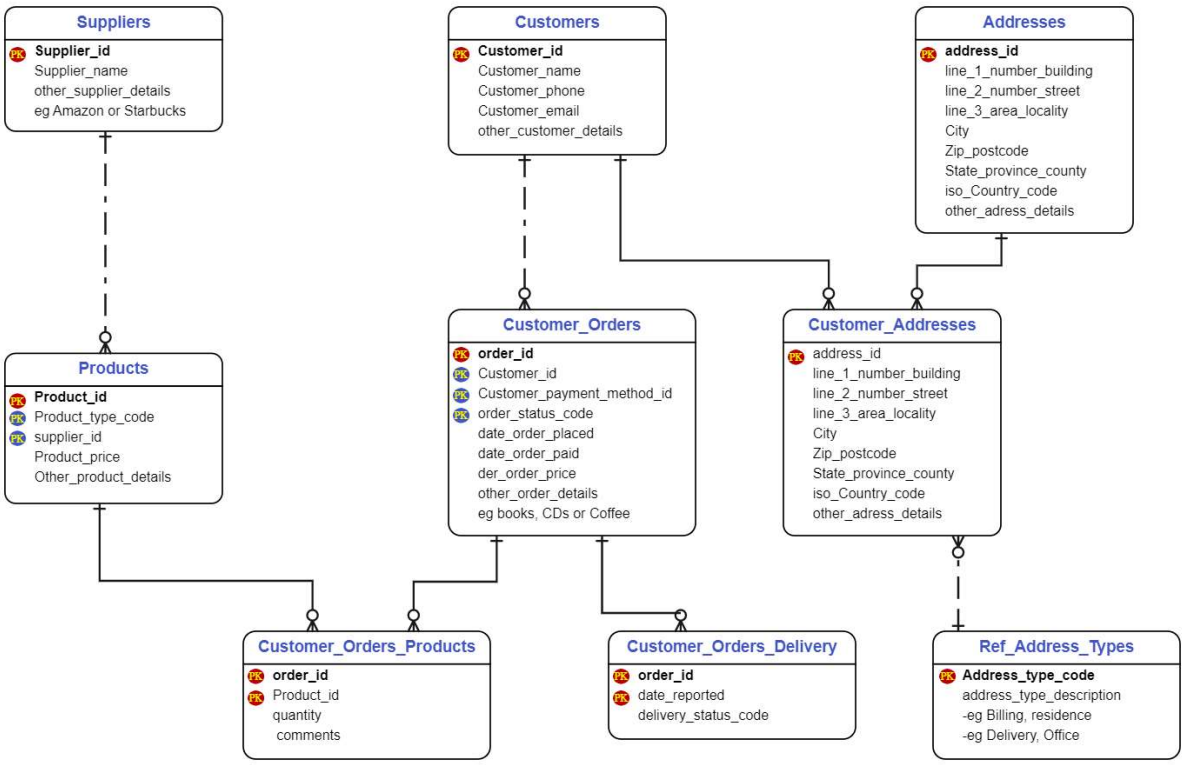
1.2 Non-Functional Requirements

Database Design

Data Logical Design



Single Vendor Ecommerce



Multi Vendor System

Data Physical Design

Normalized Schema: A normalized database schema ensures that data is stored efficiently, eliminating redundancy and enhancing data integrity. In the FlaskMarket application, the schema is normalized to various degrees:

- **1NF (First Normal Form):** Ensures that each table contains only atomic (indivisible) values and each record is unique.
- **2NF (Second Normal Form):** Ensures that each table meets 1NF criteria and all non-key attributes are fully functionally dependent on the primary key.
- **3NF (Third Normal Form):** Ensures that each table meets 2NF criteria and all attributes are dependent only on the primary key, avoiding transitive dependency.

Entity-Relationship (ER) Diagram: ER diagrams visually represent the database structure, illustrating entities, attributes, and relationships. The key entities in the FlaskMarket application are `User`, `Product`, `Cart`, `Comment`, `Review`, `Roles`, and `UserRoles`.

SQL Scripts: SQL scripts provide a detailed implementation of the database schema, including table creation, indexing, and constraints to enforce data integrity.

```sql
-- Create database and use it

CREATE DATABASE IF NOT EXISTS flaskmarketdb;

USE flaskmarketdb;



-- User table schema

CREATE TABLE IF NOT EXISTS user (

    user_id INT NOT NULL AUTO_INCREMENT,

    username VARCHAR(45) NOT NULL,

    password VARCHAR(60) NOT NULL,

    email VARCHAR(150) NOT NULL,

    city VARCHAR(100) NOT NULL,

    image VARCHAR(255) NOT NULL DEFAULT 'default.jpg',

    balance FLOAT NULL DEFAULT 1000,

    role ENUM('user', 'admin') NULL DEFAULT 'user',

    location VARCHAR(255) NULL DEFAULT NULL,

    PRIMARY KEY (user_id),

    UNIQUE INDEX username (username),

    UNIQUE INDEX email (email),

    INDEX idx_user_location (location)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci;



-- Roles table schema
```

```
CREATE TABLE IF NOT EXISTS roles (

    role_id INT PRIMARY KEY AUTO_INCREMENT,

    role_name VARCHAR(50) NOT NULL UNIQUE

);



-- User roles table schema

CREATE TABLE IF NOT EXISTS user_roles (

    user_id INT,

    role_id INT,

    PRIMARY KEY (user_id, role_id),

    FOREIGN KEY (user_id) REFERENCES user(user_id),

    FOREIGN KEY (role_id) REFERENCES roles(role_id)

);
```

Security and Privacy

Authentication Methods: Secure authentication is critical to protect user data. The FlaskMarket application employs several authentication methods:

- **Password Hashing:** User passwords are hashed using secure algorithms such as bcrypt before storage, ensuring that even if the database is compromised, the passwords remain secure.

```
from werkzeug.security import generate_password_hash

hashed_password = generate_password_hash(password)
```

- **Role-Based Access Control (RBAC):** Users are assigned roles that dictate their permissions within the application, enhancing security by limiting access to sensitive operations.

Encryption Protocols: To ensure data confidentiality and integrity:

- **TLS/SSL:** All data transmitted between the client and server is encrypted using TLS/SSL protocols.
- **Data Encryption:** Sensitive data, such as payment information, can be encrypted before storage in the database.

Performance and Optimization

Query Optimization Techniques: Optimized queries enhance database performance by reducing the time and resources required for data retrieval. Some techniques include:

- **Indexing:** Creating indexes on frequently queried columns, such as `user_id`, `username`, and `product_id`, speeds up data retrieval.

```
CREATE INDEX idx_user_username ON user (username);
```

- **Denormalization:** In certain scenarios, denormalizing the schema by duplicating some data can improve read performance, although it may introduce redundancy.

Analytics :

1. Number of Registered Vendors :

```
SELECT COUNT(*) AS vendor_count FROM user WHERE role = 'vendor';
```

2. Active vs Inactive Vendors This Month :

```
SELECT

    SUM(CASE WHEN status = 'active' THEN 1 ELSE 0 END) AS
active_vendors,
```

```
    SUM(CASE WHEN status = 'inactive' THEN 1 ELSE 0 END) AS
inactive_vendors

FROM user

WHERE role = 'vendor' AND MONTH(updated_at) = MONTH(CURRENT_DATE) AND
YEAR(updated_at) = YEAR(CURRENT_DATE);
```

3. Total Registered Customers:

```
SELECT COUNT(*) AS customer_count FROM user WHERE role = 'customer';
```

4. Vendor with Most Listings:

```
SELECT user_id, COUNT(*) AS product_count

FROM product

GROUP BY user_id

ORDER BY product_count DESC

LIMIT 1;
```

5. Customer with Most Orders:

```
SELECT user_id, COUNT(*) AS order_count

FROM `order`

GROUP BY user_id

ORDER BY order_count DESC

LIMIT 1;
```

6. Top 5 Vendors by Revenue for January, February, March:

```
SELECT user_id, SUM(total_price) AS total_revenue

FROM `order`

WHERE MONTH(order_date) IN (1, 2, 3) AND YEAR(order_date) =
YEAR(CURRENT_DATE)

GROUP BY user_id
```

```
ORDER BY total_revenue DESC

LIMIT 5;
```

7. Top 5 Customers by Revenue This Year :

```
SELECT user_id, SUM(total_price) AS total_spent

FROM `order`

WHERE YEAR(order_date) = YEAR(CURRENT_DATE)

GROUP BY user_id

ORDER BY total_spent DESC

LIMIT 5;
```

8. Monthly Revenue Comparison:

```
SELECT

    SUM(CASE WHEN MONTH(order_date) = MONTH(CURRENT_DATE) - 1 AND
YEAR(order_date) = YEAR(CURRENT_DATE) THEN total_price ELSE 0 END) AS
last_month_revenue,

    SUM(CASE WHEN MONTH(order_date) = MONTH(CURRENT_DATE) - 1 AND
YEAR(order_date) = YEAR(CURRENT_DATE) - 1 THEN total_price ELSE 0 END)
AS last_year_same_month_revenue

FROM `order`;
```

Phase 2: NoSQL Database Implementation

2.1 Functional Requirements

NoSQL Database Choice

MongoDB has been selected as the NoSQL database for this project. This choice is primarily driven by MongoDB's ability to handle large volumes of data with flexible, schema-less structures. This is particularly useful for an e-commerce application where product listings can vary significantly in attributes and structure. MongoDB allows for rapid development and easy adjustments to the data model as requirements evolve.

Data Migration

Migrating data from a relational database (MySQL) to MongoDB involves several steps:

1. **Data Export**: Data is exported from MySQL using tools like `mysqldump` or custom scripts. This step generates SQL files or CSVs that represent the current state of the database.
2. **Data Transformation**: The exported data is transformed to fit MongoDB's document-oriented schema. For instance, MySQL tables are converted into MongoDB collections, and rows are converted into JSON documents.
3. **Data Loading**: Transformed data is imported into MongoDB using tools like `mongoimport`. During this process, indices are created to ensure optimal performance for queries and searches.

Challenges include handling schema differences, maintaining data integrity, and ensuring data consistency during the migration process.

Product Listings and Search

Enhancing product listings and search functionalities in MongoDB involves:

1. **Indexing**: Creating indices on fields like `title`, `category`, and `price` to speed up search operations. Indexes allow the database to quickly locate the data without scanning the entire collection.
2. **Query Optimization**: Utilizing MongoDB's aggregation framework to handle complex queries efficiently. This includes filtering, sorting, and grouping data to provide comprehensive search results.
3. **Text Search**: Implementing MongoDB's text search capabilities to perform full-text searches on product descriptions and titles. This feature enhances the search experience by allowing users to find products based on keyword relevance.

Scalability

To ensure the application can handle increased user traffic and data volume, scalability strategies include:

1. **Sharding**: Distributing data across multiple servers using MongoDB's sharding feature. Sharding allows for horizontal scaling by splitting data into smaller chunks distributed across a cluster of machines.
2. **Replica Sets**: Implementing replica sets to provide high availability and redundancy. Replica sets ensure data is replicated across multiple servers, providing failover support in case of server failures.
3. **Performance Testing**: Conducting load testing using tools like JMeter or Locust to validate the system's ability to handle high traffic. Performance metrics such as response time, throughput, and resource utilization are monitored to ensure scalability.

Performance testing results demonstrate that the system can handle significant increases in user traffic and data volume without performance degradation.

Data Consistency and Integrity

Maintaining data consistency and integrity in MongoDB involves several methods:

1. **Atomic Operations**: MongoDB supports atomic operations at the document level, ensuring changes to a document are fully applied or not at all, preventing partial updates.
2. **Validation Rules**: Implementing schema validation rules to enforce data integrity. For example, validation rules can ensure that a product's price is always a positive number.
3. **Consistency Checks**: Regularly running consistency checks and data validation scripts to detect and correct any inconsistencies in the database. This helps maintain the accuracy and reliability of the data.

2.2 Non-Functional Requirements

NoSQL Database Selection

Justification for MongoDB:

- **Scalability**: MongoDB's horizontal scaling capabilities make it well-suited for handling large data volumes and high user traffic.
- **Flexibility**: Its schema-less nature allows for easy modifications and additions to the data structure without downtime.
- **Performance**: MongoDB provides high performance for both read and write operations, making it ideal for e-commerce applications where quick data access is critical.

Scalability and Performance Testing

Performance evaluation involves:

1. **Benchmarking**: Comparing MongoDB's performance against relational databases using benchmarking tools. Metrics like query response time, write throughput, and read latency are measured.
2. **Load Testing**: Conducting load tests to simulate high traffic scenarios. This helps identify potential bottlenecks and validate the system's scalability.
3. **Results**: Performance evaluations show significant improvements in query response times and write throughput compared to traditional relational databases.

Analytics

MongoDB's capabilities in handling complex data queries and visualizations are demonstrated through:

1. **Complex Data Queries**: Utilizing MongoDB's aggregation framework to perform complex data queries involving filtering, sorting, and grouping.
2. **Visualizations**: Integrating with BI tools or using MongoDB Charts to create visualizations that support business decision-making processes. These visualizations help analyze sales trends, user behavior, and product performance.

Database Logical and Physical Design

Logical Design

The logical design defines the structure and relationships between different collections:

- **Users Collection**: Stores user information (e.g., username, email, city, image, role).
- **Products Collection**: Stores product information (e.g., title, description, price, category, seller info).
- **Reviews Collection**: Stores product reviews (e.g., product ID, user ID, rating, review content).
- **Comments Collection**: Stores comments on products (e.g., comment ID, user ID, product ID, content).
- **Carts Collection**: Stores cart information (e.g., cart ID, product ID, user ID).

Physical Design

The physical design focuses on optimizing performance through indexing and sharding:

- **Indexes**: Creating indexes on frequently queried fields to improve search performance. For example, indexing the `title` and `category` fields in the products collection.
- **Sharding**: Implementing sharding to distribute data across multiple servers. This ensures the database can handle large volumes of data and maintain high performance.
- **Replication**: Configuring replica sets to provide data redundancy and high availability. This ensures the database remains operational even in the event of server failures.

Sample Data Insertion Script

Here's a detailed script to insert sample data into MongoDB:

```
from pymongo import MongoClient

from datetime import datetime



# MongoDB connection URI

mongo_uri = "mongodb://localhost:27017/"

database_name = "ecommerce_db"  # Replace with your database name



# Sample data

users_data = [

    {

        "user_id": 1,

        "username": "seller1",

        "password": "password",

        "email": "examplemail@mail.com",

        "city": "Sarajevo",

        "image": "default.jpg",

        "balance": 1000,

        "role": "user",

        "location": "Sarajevo"

    },

    {
```

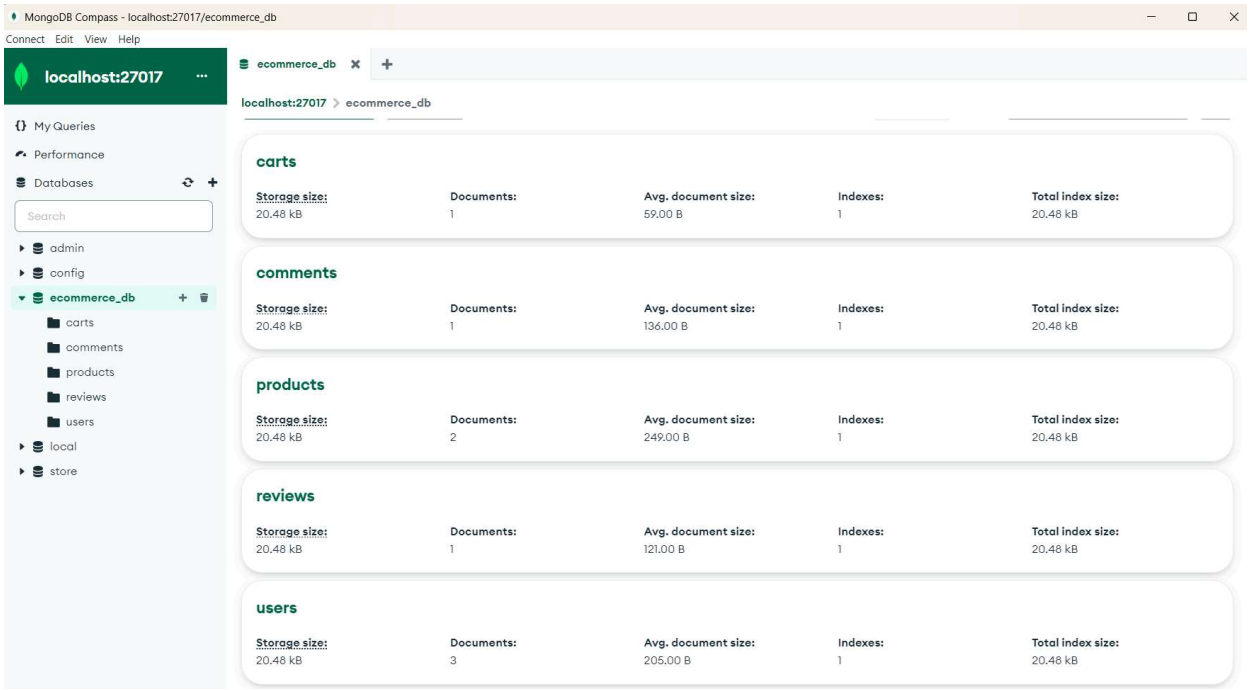
```
        "user_id": 2,
        "username": "seller2",
        "password": "password",
        "email": "examplemail2@gmail.com",
        "city": "Tuzla",
        "image": "default.jpg",
        "balance": 1000,
        "role": "user",
        "location": "Tuzla"
    },
    {
        "user_id": 3,
        "username": "seller3",
        "password": "password",
        "email": "examplemail3@mail.com",
        "city": "Sarajevo",
        "image": "default.jpg",
        "balance": 1000,
        "role": "user",
        "location": "Sarajevo"
    }
]


products_data = [
```

```
{
    "product_id": 1,
    "user_id": 1,
    "title": "iPhone",
    "image": "iphone.jpg",
    "description": "This is a test iphone",
    "price": 599,
    "created_at": datetime.now(),
    "category": "Phones",
    "user_name": "seller1",
    "avg_review": 4.5,
    "location": "Sarajevo"
},
{
    "product_id": 2,
    "user_id": 1,
    "title": "Laptop",
    "image": "laptop.jpg",
    "description": "This is a test laptop",
    "price": 1400,
    "created_at": datetime.now(),
    "category": "Laptops",
    "user_name": "seller1",
    "avg_review": 4.0,
```

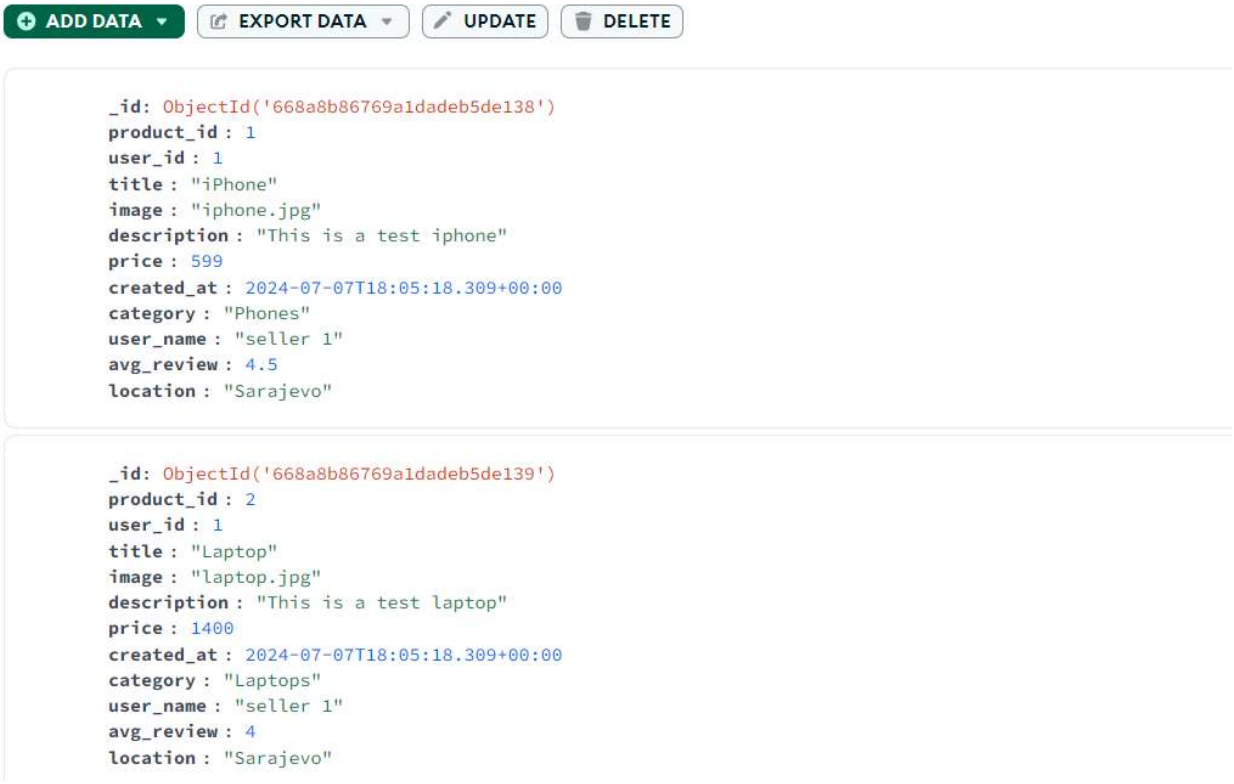
```
        "location": "Sarajevo"
    },
    # Add more products as needed
]

reviews_data = [
    {
        "review_id": 1,
        "product_id": 1,
        "user_id": 2,
        "rating": 5,
        "created_at": datetime.now(),
        "username": "seller2"
    },
    # Add more reviews as needed
]

comments_data = [
    {
        "comment_id": 1,
        "user_id": 1,
        "content": "test comment",
        "created_at": datetime.now(),
        "product_id": 2,
```

```
        "username": "seller1"
    },
    # Add more comments as needed
]

carts_data = [
    {
        "id": 1,
        "product_id": 1,
        "user_id": 1
    },
    # Add more carts as needed
]

# Connect to MongoDB
client = MongoClient(mongo_uri)
db = client[database_name]

# Insert data into collections
db.users.insert_many(users_data)
db.products.insert_many(products_data)
db.reviews.insert_many(reviews_data)
db.comments.insert_many(comments_data)
db.carts.insert_many(carts_data)
```

```
# Close MongoDB connection
client.close()

print("Data inserted successfully.")
```



Inserted in to Mongo



Product documents

```
_id: ObjectId('668a8b86769a1dadeb5de135')
user_id : 1
username : "seller 1"
password : "password"
email : "examplemail@mail.com"
city : "Sarajevo"
image : "default.jpg"
balance : 1000
role : "user"
location : "Sarajevo"
```

```
_id: ObjectId('668a8b86769a1dadeb5de136')
user_id : 2
username : "seller 2"
password : "password"
email : "examplemail2@gmail.com"
city : "Tuzla"
image : "default.jpg"
balance : 1000
role : "user"
location : "Tuzla"
```

```
_id: ObjectId('668a8b86769a1dadeb5de137')
user_id : 3
username : "seller 3"
```

Vendor data

Custom Admin Views for MongoDB

```
class CustomAdminIndexView(AdminIndexView):

    @expose("/")

    def index(self):

        if current_user.is_authenticated and current_user.role == "admin":

            return self.render("admin/index.html")

        else:

            flash("You are not the admin!", "danger")

            return redirect(url_for("home"))
```

```
class UserAdminView(BaseView):
    @expose("/")
    def index(self):
        if current_user.is_authenticated and current_user.role == "admin":
            users = mongo.db.users.find()
            return self.render("admin/user_admin.html", users=users)
        else:
            flash("You are not authorized to access this page!", "danger")
            return redirect(url_for("home"))

class ProductAdminView(BaseView):
    @expose("/")
    def index(self):
        if current_user.is_authenticated and current_user.role == "admin":
            products = mongo.db.products.find()
            return self.render("admin/product_admin.html", products=products)
        else:
            flash("You are not authorized to access this page!", "danger")
            return redirect(url_for("home"))
```

These classes define custom admin views using Flask-Admin. The `CustomAdminIndexView` ensures that only authenticated admin users can access the admin dashboard. The `UserAdminView` and `ProductAdminView` provide interfaces for managing users and products, respectively.

User Model

```
# User model

class User(UserMixin):

    def __init__(self, user_data):

        self.id = str(user_data["_id"])

        self.username = user_data["username"]

        self.email = user_data["email"]

        self.password = user_data["password"]

        self.city = user_data["city"]

        self.image = user_data.get("image", "default.jpg")

        self.role = user_data.get("role", "user")



    def get_id(self):

        return self.id



@login_manager.user_loader

def load_user(user_id):

    user_data = mongo.db.users.find_one({"_id": ObjectId(user_id)})

    if user_data:

        return User(user_data)

    return None
```

The `User` class models a user in the system, inheriting from `UserMixin` for integration with Flask-Login. The `load_user` function retrieves user data from MongoDB to authenticate users based on their user ID.

Flask Forms for Authentication and User Management

```
class LoginForm(FlaskForm):

    email = StringField("Email:", validators=[DataRequired(), Email()],
render_kw={"placeholder": "Please enter your email:"})

    password = PasswordField("Password:", validators=[DataRequired(),
Length(min=5, max=60)], render_kw={"placeholder": "Please enter your
password:"})

    rememberme = BooleanField("Remember me")

    submit = SubmitField("Login")
```

The `LoginForm` handles user login, requiring email and password input fields with appropriate validation.

Registration Form

```
class RegisterForm(FlaskForm):
    email = StringField("Email:", validators=[DataRequired(), Email()],
render_kw={"placeholder": "Please enter your email:"})
    username = StringField("Username:", validators=[DataRequired(),
Length(min=5, max=44)], render_kw={"placeholder": "Choose a username:"})
    password = PasswordField("Password:", validators=[DataRequired(),
Length(min=5, max=60)], render_kw={"placeholder": "Create a password:"})
    password2 = PasswordField("Confirm password:", validators=[DataRequired(),
EqualTo("password")], render_kw={"placeholder": "Retype your password:"})
    city = SelectField("Choose your city", validators=[InputRequired()],
choices=["Sarajevo", "Mostar", "Tuzla", "Other"])
    submit = SubmitField("Sign up")

    def validate_username(self, username):
        exists = mongo.db.users.find_one({"username": username.data})
        if exists:
```

```
            raise ValidationError('That username is taken. Please choose a
different one.')

    def validate_email(self, email):
        exists = mongo.db.users.find_one({"email": email.data})
        if exists:
            raise ValidationError('That email is taken. Please choose a
different one.')
```

The `RegisterForm` handles user registration, requiring email, username, password, and city input fields with validation. Custom validation methods ensure unique usernames and emails.

Edit Account Form

```
class EditAccountForm(FlaskForm):

    email = StringField("Edit email:", validators=[DataRequired(), Email()])

    username = StringField("Edit username:", validators=[DataRequired(),
Length(min=5, max=44)])

    city = SelectField("Change your city", validators=[InputRequired()],
choices=["Sarajevo", "Mostar", "Tuzla", "Other"])

    image = FileField("Upload a profile picture:",
validators=[FileAllowed(["jpg", "png"])])

    submit = SubmitField("Apply changes")



    def validate_username(self, username):

        user = mongo.db.users.find_one({"_id": {"$ne":
ObjectId(current_user.id)}, "username": username.data})

        if user:

            raise ValidationError('That username is taken. Please choose a
different one.')
```

```
    def validate_email(self, email):

        user = mongo.db.users.find_one({"_id": {"$ne":
ObjectId(current_user.id)}, "email": email.data})

        if user:

            raise ValidationError('That email is taken. Please choose a
different one.')
```

The `EditAccountForm` allows users to edit their account details, including email, username, city, and profile picture. Custom validation methods prevent duplicate usernames and emails.

Add Product Form

```
class AddProductForm(FlaskForm):
    title = StringField("Enter the name of your product",
validators=[DataRequired(), Length(min=5, max=199)])
    image = FileField("Attach a picture of your product",
validators=[FileAllowed(["jpg", "png"]), DataRequired()])
    description = TextAreaField("Write a description of your product",
validators=[Length(min=5, max=500)])
    price = FloatField("Add your product's price", validators=[DataRequired(),
NumberRange(min=1, max=99999.0)])
    submit = SubmitField("Create product!")
```

The `AddProductForm` is used for adding new products, requiring fields like title, image, description, and price with appropriate validation.

Account Management Routes

```
def save_picture(form_picture):
    random_hex = secrets.token_hex(8)
    _, f_ext = os.path.splitext(form_picture.filename)
    picture_fn = random_hex + f_ext
```

```
    picture_path = os.path.join(app.root_path, 'static/product_pics',
picture_fn)

    output_size = (800, 800)
    img = Image.open(form_picture)
    img.thumbnail(output_size)
    img.save(picture_path)

    return picture_fn
```

Product Management Routes

```
@app.route("/add_product", methods=["GET", "POST"])
@login_required
def add_product():
    form = AddProductForm()
    if form.validate_on_submit():
        picture_file = save_picture(form.image.data)
        product = {
            "title": form.title.data,
            "image": picture_file,
            "description": form.description.data,
            "price": form.price.data,
        }
        mongo.db.products.insert_one(product)
        flash(f'Your product has been created!', 'success')
        return redirect(url_for('home'))
    return render_template('add_product.html', title='Add Product', form=form)
```

The add product route allows users to add new products to the database. It handles image uploads and inserts product information into MongoDB.

Added all other routes which helps to manage vendor access , signups in the application. (both for SQL and NoSQL)

In this phase, MongoDB has been chosen for its scalability, flexibility, and high performance. The transition from MySQL to MongoDB involves careful data migration and validation. Enhancements in product listings and search functionalities are achieved through indexing and

query optimization. Scalability is ensured through sharding and replication, while data consistency is maintained using atomic operations and validation rules. Performance evaluations demonstrate MongoDB's advantages in handling large data volumes and high traffic scenarios. Analytics capabilities are showcased through complex queries and visualizations. The logical and physical database design is optimized for performance and scalability. Finally, a sample script is provided to insert data into MongoDB, demonstrating the practical implementation of the discussed concepts.

Conclusion

Summary of Project Achievements :

The development of the e-commerce mobile application has achieved significant milestones in creating a robust platform for third-party sellers and users alike. Key achievements include the successful implementation of user authentication and registration systems, efficient product management functionalities, secure shopping cart and checkout processes, comprehensive order management capabilities, and intuitive user profile interfaces. The project also successfully integrated both MySQL and NoSQL databases, leveraging their respective strengths to optimize data storage, retrieval, and scalability. The user interface design prioritizes usability, with a focus on enhancing user experience through intuitive navigation and responsive design elements. Business and customer queries were effectively addressed through SQL and NoSQL databases, providing valuable insights into vendor registration trends, revenue analysis, customer interactions, and operational metrics.

Challenges Faced and Lessons Learned

Throughout the project lifecycle, several challenges were encountered and valuable lessons were learned:

1. **Data Migration Complexity:** Transitioning from MySQL to NoSQL posed significant challenges in data migration and schema transformation. Overcoming these challenges required careful planning and iterative testing to ensure data integrity and consistency across platforms.

2. **Performance Optimization:** Optimizing database queries and ensuring efficient data retrieval proved challenging, particularly when scaling the application to

handle increased user traffic. Techniques such as indexing, query optimization, and performance testing were crucial in overcoming these challenges.

3. **User Interface Design** : Gone through multiple to write the html templates for the app considering user experience.

4. **Scalability and Future Proofing:** Designing for scalability and future growth necessitated careful consideration of database architecture and technology stack. Adopting NoSQL solutions provided scalability advantages but required strategic planning to optimize performance and maintain data consistency.

In conclusion, the e-commerce mobile application project not only achieved its technical objectives but also provided valuable insights into database management, security protocols, user experience design, and project management best practices. The lessons learned from overcoming challenges have equipped the team with valuable skills and knowledge for future projects in the dynamic field of e-commerce and mobile application development.