GREECE - VOLOS
University of Thessaly
Department of Electrical and
Computer Engineering

Advance Data Management
Programming Project

Development of an application supported by a NoSQL Database

Prepared by:

Giacomo Di Fabrizio

Professor

Michael Vassilakopoulos

Contents

Selection of the topic

The objective of this project is to develop a MERN (MongoDB, Express.js, React.js, Node.js) application tailored for an e-commerce website.

Description and specification of the topic.

- **Description of the topic**
  - The aim of this project is to design and implement a robust e-commerce website using the MERN (MongoDB, Express.js, React.js, Node.js) stack. The website will serve as a platform for users to browse, search, and purchase products online. It will provide an intuitive and seamless shopping experience for customers. (while offering efficient management tools for administrators.)

    The e-commerce website will feature various functionalities including user authentication, product catalog management, filtering of the products based on category, shopping cart management and order tracking. The frontend will be developed using React.js to ensure a responsive and dynamic user interface, while the backend will be powered by Node.js and Express.js to handle server-side logic and API requests. MongoDB will be utilized as the database to store product information, user data, and transaction records.

    To enhance user experience, the website will incorporate modern design principles, intuitive navigation, and responsive layout to ensure compatibility across different devices and screen sizes.

    Additionally, security measures such as encryption for sensitive data, secure authentication mechanisms, and protection against common web vulnerabilities will be implemented to safeguard user information and transactions.

    Overall, the project endeavors to deliver a fully functional e-commerce website that meets the needs of both customers and administrators, pro-

viding a seamless shopping experience and efficient management tools.

- **Specification of the topic**
  - User Authentication:
    - Users should be able to register, log in, and log out securely.
      Passwords should be securely hashed and stored in the database.
  - Product Catalog Management:
    - Products should be categorized and displayed with relevant details such as name, description, price, and images.
      Each product should have a unique identifier.
  - Shopping Cart Management:
    - Users should be able to add products to their shopping cart and adjust quantities.
      The shopping cart should display a summary of selected products with total prices.
      Users should have the option to remove items from the cart or proceed to checkout.
  - Order Tracking:
    - Users should be able to view their order history and track the status of their orders.
  - Responsive Design:
    - The website should be responsive and compatible with various devices and screen sizes.
      Mobile-friendly design elements should be implemented for optimal user experience on smartphones and tablets.
  - Performance Optimization:
    - Efficient querying and indexing of database to ensure fast response times. Minification and bundling of frontend assets to reduce loading times. Implementation of caching mechanisms for frequently accessed data.

Selection of NoSQL DB system suitable for the above topic.

For this project, MongoDB has been selected as the NoSQL database system. MongoDB's flexibility, scalability, and ability to handle unstructured data make it well-suited for the requirements of an e-commerce website. Its document-oriented data model allows for easy storage and retrieval of product information, user data, and transaction records. Additionally, MongoDB's support for sharding and replication ensures high availability and fault tolerance, crucial aspects for an e-commerce platform handling large volumes of data and transactions. Overall, MongoDB is an excellent choice to meet the needs of the project and provide a reliable foundation for data storage and management.

Installation of the NoSQL DB System.

For the installation of the NoSQL DB system, specifically MongoDB, I opted to utilize MongoDB Atlas, a cloud-based database service provided by MongoDB. Here's how I set it up:

- **Sign up for MongoDB Atlas:**
  - I visited the MongoDB Atlas website and signed up for an account. MongoDB Atlas offers a free tier, which is suitable for getting started with small-scale projects.
- **Create a Project:**
  - After signing in to MongoDB Atlas, I created a new project.

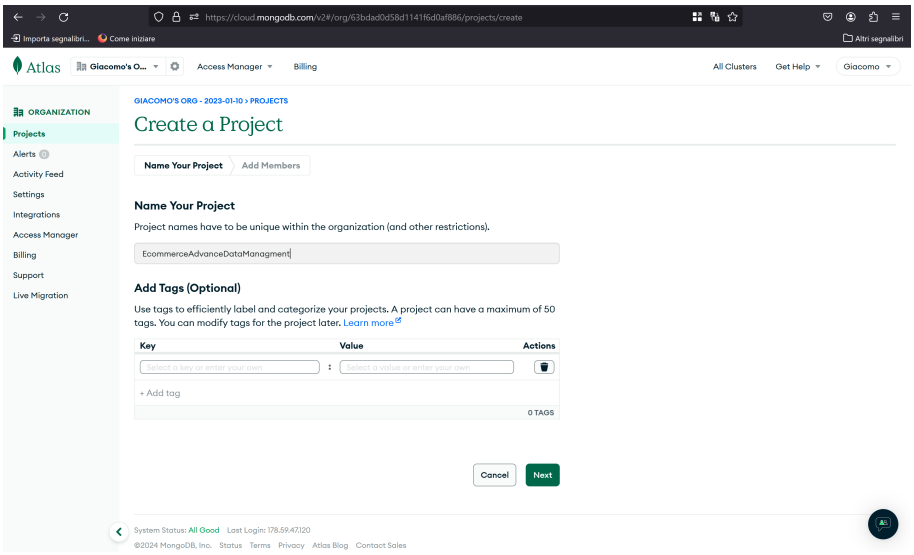

Figure 1: Creation of a new Project called "Ecommerce Advance Data Management"

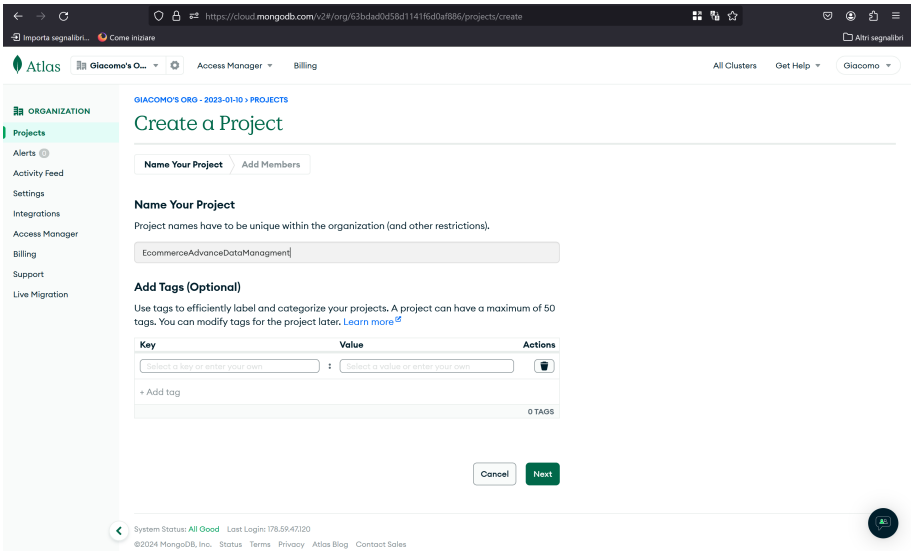

Figure 2: Adding member and set permissions

- **Create a Cluster:**
  - After creating a new project, I created a new cluster.

    A cluster in MongoDB Atlas consists of one or more MongoDB instances running on the cloud.

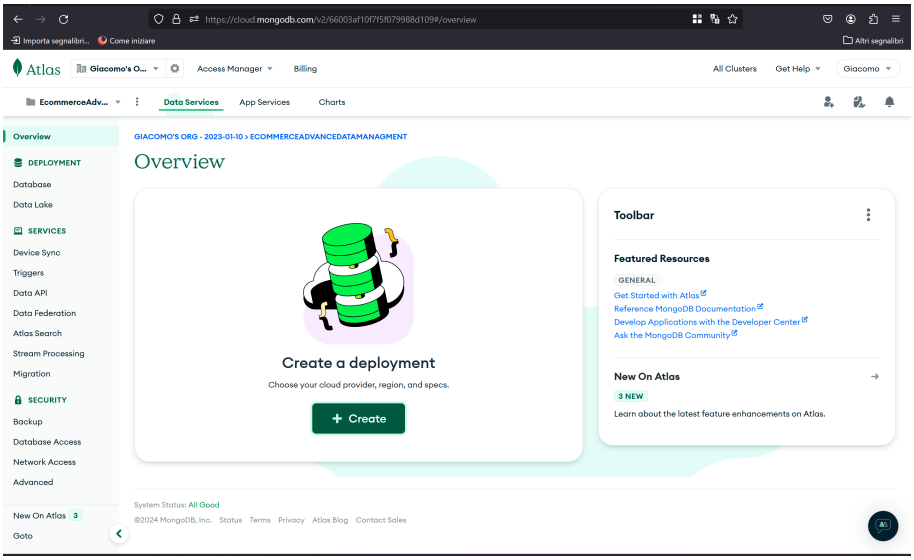

Figure 3: Creation of a new cluster

- **Configure Cluster Settings:**
  - During the cluster creation process, I configured settings such as the cloud service provider (e.g., AWS, Azure, GCP), region, and cluster tier based on my requirements and preferences. MongoDB Atlas provides

options for high availability, automatic scaling, and backups to ensure data reliability and availability.

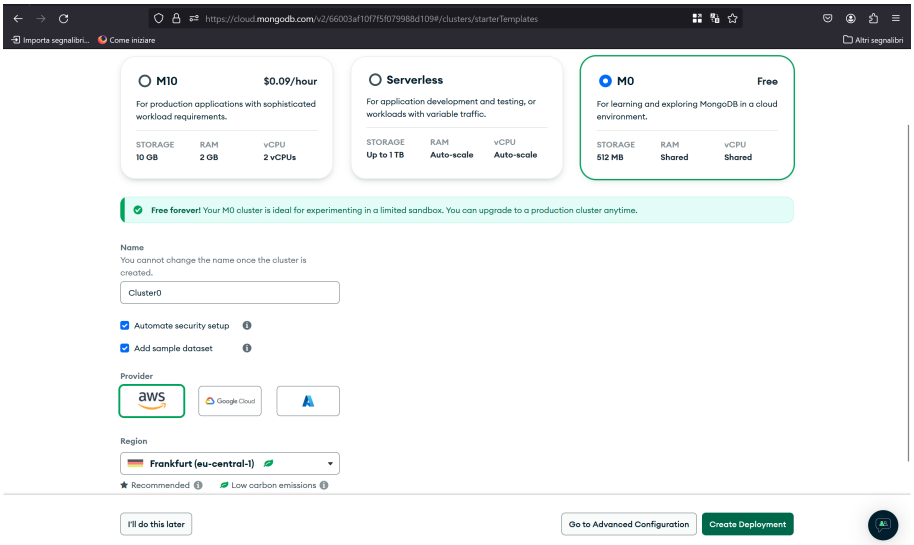

Figure 4: Configuration of the cluster settings

- **Set Up Security:**
  - I configured security settings for the cluster, including setting up IP whitelisting, creating database users with appropriate privileges, and enabling encryption at rest and in transit to protect data security.

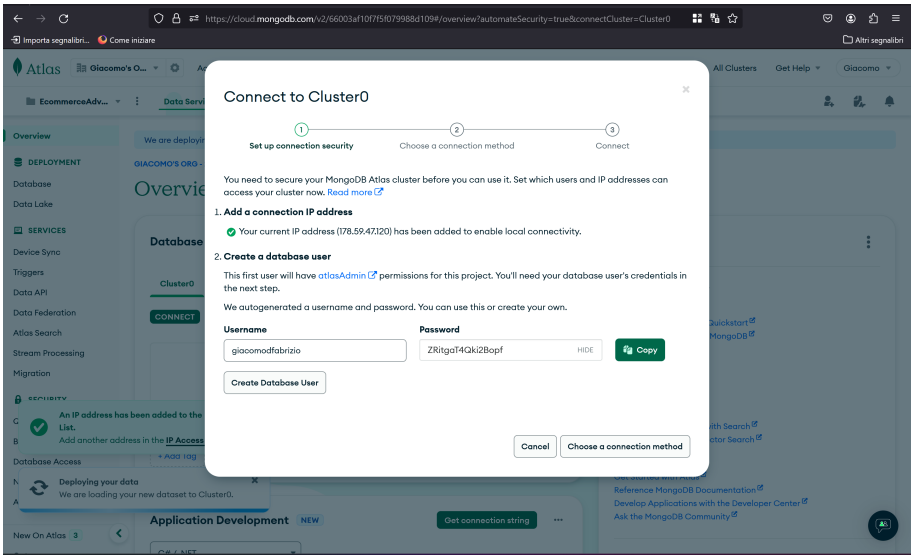

Figure 5: Adding a connection IP address

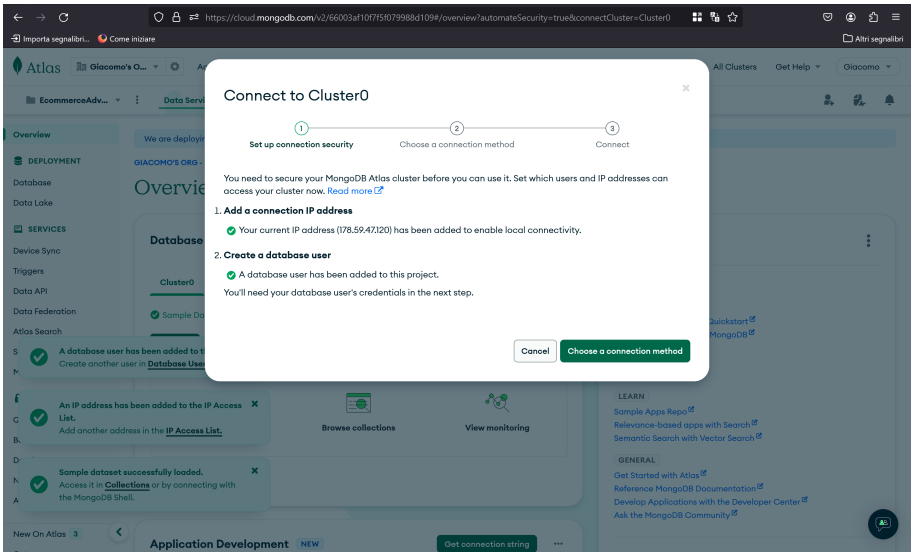

Figure 6: Creation of a database user

- **Connect to the Cluster:**
  - MongoDB Atlas provides a connection string that I used to connect to the cluster from my application code. This connection string includes authentication credentials and other necessary information to establish a connection securely.

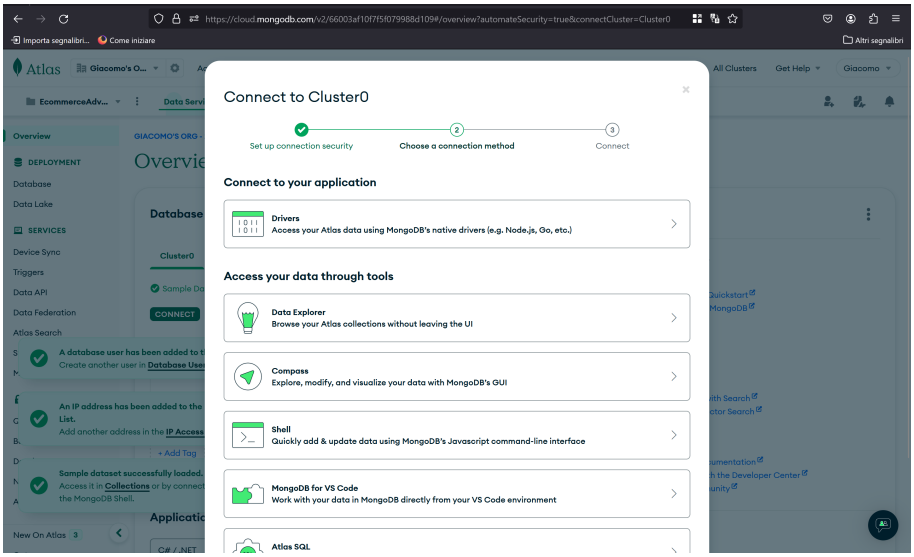

Figure 7: Different ways to connect a cluster

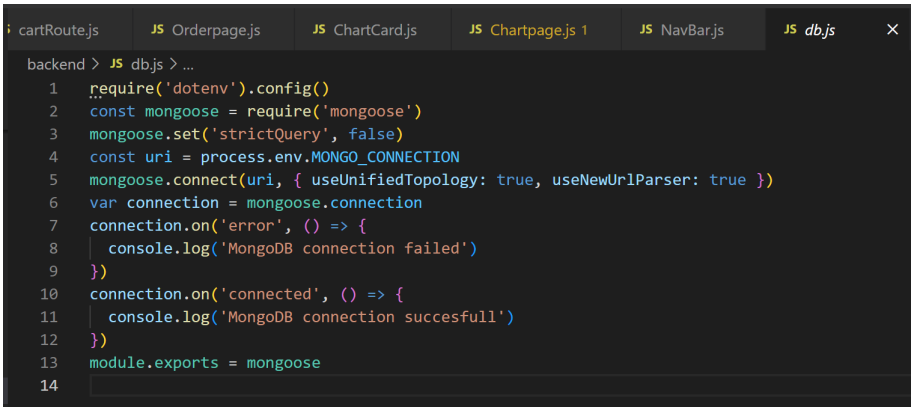

Figure 8: Connection of the cluster by drivers using Javascript

- **Verify Connection:**
  - After configuring the connection, I tested the connection from my application to MongoDB Atlas to ensure that it was working correctly. This involved connecting to the cluster using the provided connection string and performing basic database operations to verify functionality.

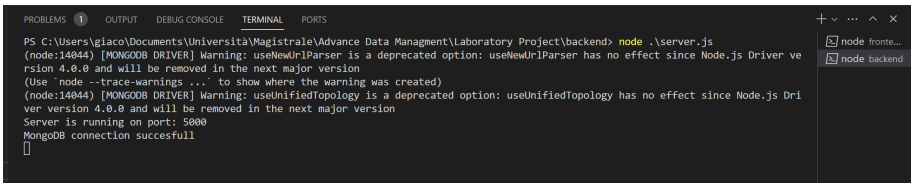

Figure 9: Verify of the connection

By using MongoDB Atlas, I benefited from the advantages of a cloud-based database service, including scalability, high availability, automated backups, and managed infrastructure, without the need for manual installation and maintenance of database servers. Additionally, MongoDB Atlas provides a user-friendly interface for managing clusters and monitoring database performance, making it a convenient choice for hosting MongoDB databases in the cloud.

Identification of useful queries for the application.

Here are some useful queries for the e-commerce application:

- **User Authentication and Authorization:**
  - Verify user credentials during login: Find a user by email and compare hashed passwords.
  - Register a new user: Check if the user already exists and then save the user details with a hashed password.
  - Verify user role: Decode the JWT token and extract the isAdmin field to determine user role.
- **User Profile:**
  - Retrieve user details: Decode the JWT token to extract user information for profile display.
- **Product Management:**
  - Get specific product details: Find a product by its unique identifier.
  - Filter products by category: Retrieve products belonging to a specific category.
  - Get all products: Retrieve all products available in the catalog.
- **Order Management:**
  - Add a new order: Create a new order by retrieving items from the user's cart, updating product quantities, and calculating the total cost.

  - Get user's order history: Find orders associated with a specific user and include product details for each order.

- **Cart Management:**
  - Retrieve user's cart: Find the user's cart and retrieve details of products added to it.
  - Add object to cart: Update the user's cart by adding a new object or increasing the quantity of an existing object.
  - Delete object from cart: Remove an object from the user's cart based on the product ID.
  - Delete user's entire cart: Delete all objects from the user's cart.

- **Catalog Management:**
  - Add object to catalog: Add a new product to the catalog or update the quantity of an existing product.
  - Get quantity available for a product: Retrieve the quantity available for a specific product in the catalog.

Creating data using data generator or finding ready-made data.

For the creation of data in the e-commerce application, I utilized original product data obtained from an external source. This data consists of various products with details such as title, price, description, category, image, and ratings. Here is a snippet of the original data:


```
[
  {
    "id": 1,
    "title": "Fjallraven-Foldsack No. 1 Backpack, Fits 15 Laptops",
    "price": 109.95,
    "description": "Your perfect pack for everyday use",
    "category": "men's clothing",
    "image": "https://fakestoreapi.com/img/81fPKd-AC-SL1500-.jpg",
    "rating": { "rate": 3.9, "count": 120 }
  },
  {
    "id": 2,
    "title": "Mens Casual Premium Slim Fit T-Shirts",
    "price": 22.3,
    "description": "Slim-fitting style, contrast raglan sleeve",
    "category": "men's clothing",
    "image": "https://fakestoreapi.com/img/71-3HjGNDUL.jpg",
    "rating": { "rate": 4.1, "count": 259 }
```

```
  },
  // More products...
]
```

To prepare this data for insertion into MongoDB, I filtered and transformed it into a format suitable for database insertion. The filtered data includes essential product details such as name, description, price, category, and image. Here is an example of the filtered data:

```
[
  {
    "name": "Fjallraven-Foldsack No. 1 Backpack, Fits 15 Laptops",
    "description": "Your perfect pack for everyday use",
    "price": 109.95,
    "category": "men's clothing",
    "image": "https://fakestoreapi.com/img/81fPKd-AC-SL1500-.jpg"
  },
  {
    "name": "Mens Casual Premium Slim Fit T-Shirts",
    "description": "Slim-fitting style, contrast raglan sleeve",
    "price": 22.3,
    "category": "men's clothing",
    "image": "https://fakestoreapi.com/img/71-3HjGNDUL.jpg"
  },
  // More filtered products...
]
```

To accomplish this transformation, I utilized Python scripting. The Python script filtered the original data and generated the filtered data in the desired format, which was then ready for insertion into the MongoDB database. This process ensured that the data inserted into the database met the requirements of the e-commerce application and was structured appropriately for efficient retrieval and usage.

```
import json
```

```
with open("product.json","r",encoding="utf-8") as file:
    contenuto = json.load(file)

# Creation of a new elenc with only the request keys
filtered_data = []
for item in contenuto:
    filtered_item = {
        "name": item["title"],
        "description": item["description"],
        "price": item["price"],
        "category": item["category"],
        "image": item["image"]
    }
    filtered_data.append(filtered_item)

# Writing of the filtered data in a new JSON file
with open('filtered_data.json', 'w') as f:
    json.dump(filtered_data, f, indent=4)
```

Designing the DB model and implementing it in the NoSQL DB system.

For designing the database model and implementing it in the NoSQL DB system, I structured the data using Mongoose schemas in MongoDB. Below are the models I've utilized:

- **Cart Model:**
  - Stores the items added to the user's cart.
  - Schema includes user ID and an array of items with product ID and quantity.

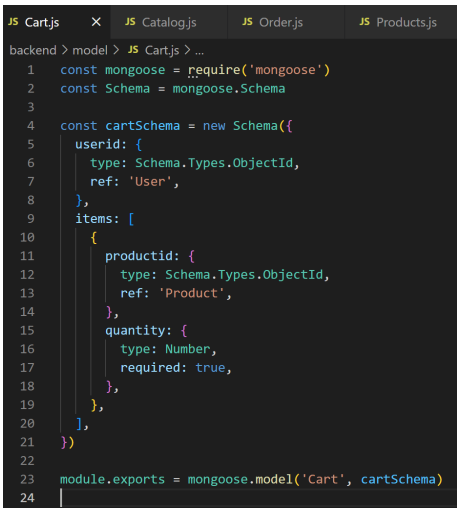

Figure 10: Cart Model

- **Catalog Model:**
  - Represents the catalog of products with their available quantities.

– Schema includes product ID and quantity.

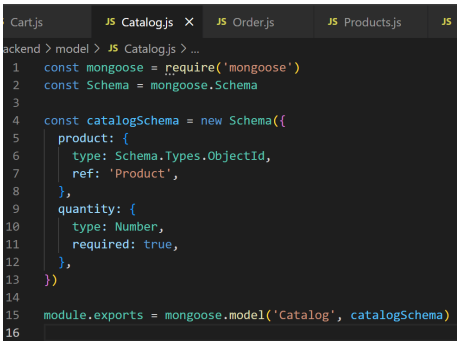

Figure 11: Catalog Model

- **Order Model:**
  - Stores user orders with details of products, quantities, total cost, and order status.
  - Schema includes user ID, array of items with product ID and quantity, total cost, and status.

Figure 12: Ordel Model

- **Product Model:**
  - Represents the products available in the e-commerce system.

– Schema includes details such as name, description, price, category, and image.

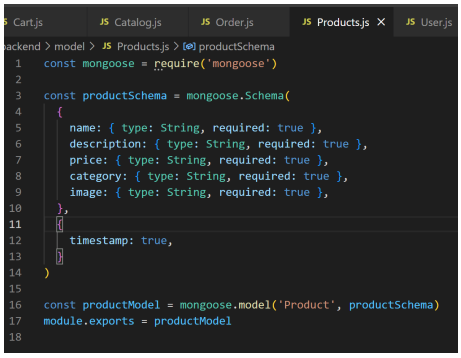

Figure 13: Product Model

- **User Model:**
  - Represents user accounts in the e-commerce system.
  - Schema includes user details such as firstname, lastname, email, password, street, money (balance), and isAdmin (admin privilege).

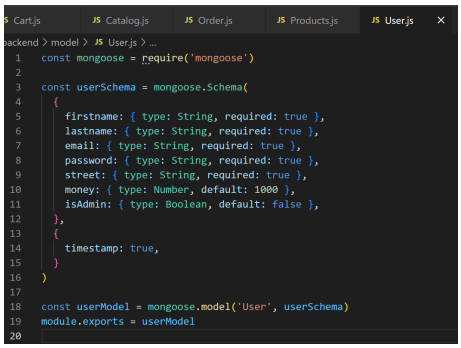

Figure 14: User Model

Loading the data into the DB.

To load the data into the NoSQL database, I utilized a JSON file containing the product information and implemented a route in the backend to insert this data into the database.
Here's how the process was carried out:

- **Data Source:**
  - The product data was stored in a JSON file named "filtered_data.json". This file contains an array of objects, with each object representing a product and its details such as name, description, price, category, and image.
- **Reading JSON File:**
  - Using the 'fs' module in Node.js, the backend server reads the JSON file and parses its contents into a JavaScript object.
  - This is achieved with the following code snippet:

```
3   const fs = require('fs')
4
5   const Product = require('../model/Products')
6
7   const jsonData = JSON.parse(
8     fs.readFileSync('../backend/filtered_data.json', 'utf-8')
9   )
10
```

Figure 15: Reading of the JSON file

- **Loading Data Route:**
  - A route named "/addallproduct" was created to handle the insertion of all products from the JSON file into the database.

- Inside this route, the ‘insertMany()‘ method provided by Mongoose is used to insert multiple documents into the database in a single operation.
- The route is implemented as follows:

```
47  //Route for add all product from json file inside the server
48  router.get('/addallproduct', async (req, res) => {
49    try {
50      const result = await Product.insertMany(jsonData)
51      res.send(result)
52    } catch (error) {
53      return res.status(404).json(error)
54    }
55  })
56
```

Figure 16: Route for adding all the products in the mongoDB database

- **Execution:**
  - When a GET request is made to the ”/addallproduct” endpoint, the server reads the JSON data from the file and inserts it into the database using the ‘insertMany()‘ method.
  - Upon successful insertion, a response containing the result of the operation is sent back to the client.
- **Error Handling:**
  - Error handling is implemented to catch any potential errors during the insertion process. If an error occurs, a 404 status code along with the error message is returned to the client.

By utilizing this approach, the product data stored in the JSON file is effectively loaded into the NoSQL database, making it accessible for use within the e-commerce application. This ensures that the application has access to the necessary data to provide users with a seamless shopping experience.

Creating code for CRUD (Create, read, update and delete) data operations.

To implement CRUD (Create, Read, Update, Delete) operations for the data in the MongoDB database, several routes have been created in the backend server using Express.js.
These routes handle various operations on different types of data, including products, carts, orders, and user authentication.
Let's examine each route and its corresponding CRUD operation:

- **Products Routes ('products.js'):**
  - Read Product by ID:
    - Route: 'GET /product/:id'
    - Retrieves a product by its ID from the database.
  - Read Products by Category
    - Route: 'GET /product/category/:category'
    - Retrieves all products belonging to a specific category.
  - Read All Products:
    - Route: 'GET /getallproduct'
    - Retrieves all products from the database.
  - Create Products from JSON File:
    - Route: 'GET /addallproduct'
    - Inserts all products from a JSON file into the database.

- **Cart Routes (‘cart.js‘):**
  - Read Cart Items:
    - Route: ‘POST /getcart‘
    - Retrieves the items in the user’s cart.
  - Delete Cart:
    - Route: ‘POST /deletecart‘
    - Deletes the entire cart of the user.
  - Delete Item from Cart:
    - Route: ‘POST /deleteobjectcart‘
    - Deletes a specific item from the user’s cart.
  - Add Item to Cart:
    - Route: ‘POST /addobjectcart‘
    - Adds a new item (and its quantity) to the user’s cart.
- **Catalog Routes (‘catalog.js‘):**
  - Add Product to Catalog:
    - Route: ‘POST /addtocatalog‘
    - Adds a product to the catalog with its quantity.
  - Read Product Quantity:
    - Route: ‘GET /quantity/:id‘
    - Retrieves the available quantity of a product.
- **Order Routes (‘orders.js‘):**
  - Add Order:
    - Route: ‘POST /addorder‘
    - Creates a new order with the user’s cart items, deducts the total cost from the user’s balance, and updates the catalog quantities.
  - Read User Orders:
    - Route: ‘POST /getorders‘
    - Retrieves all orders made by the user along with product details.

- **User Authentication Routes (‘auth.js‘):**
  - User Login:
    - Route: ‘POST /login‘
    - Authenticates the user with email and password, generates a JWT token.
  - User Registration:
    - Route: ‘POST /register‘
    - Registers a new user with hashed password and generates a JWT token.
  - Verify User Role:
    - Route: ‘POST /verifyrole‘
    - Verifies if the user is an admin based on the JWT token.
  - Get User Info:
    - Route: ‘POST /user‘
    - Retrieves user information from the JWT token.
  - Get User Money Balance:
    - Route: ‘POST /getmoneyaviable‘
    - Retrieves the available money balance of the user from the JWT token.

These routes collectively provide functionality for managing the CRUD operations on the data stored in the MongoDB database, ensuring smooth interaction with the e-commerce application.

Grafical User Interface (GUI).

Provided below are some pictures showing the graphical user interface (GUI) of the website.

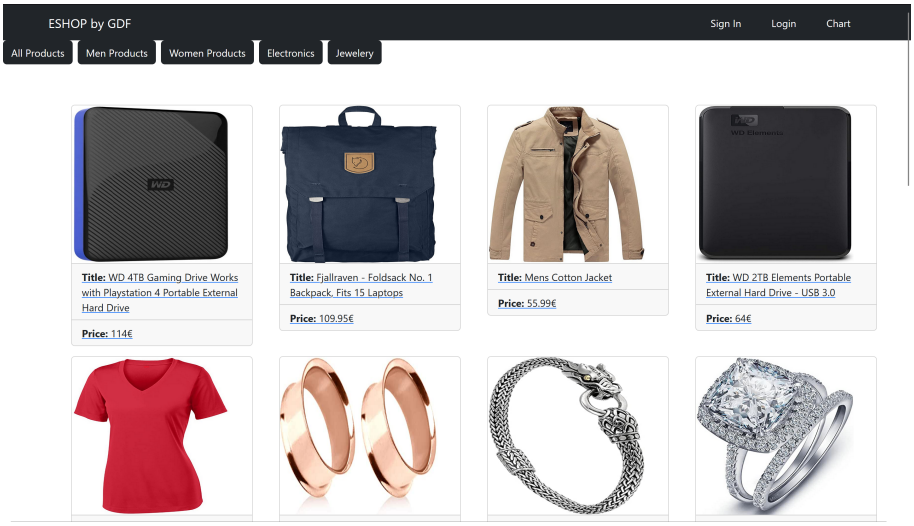

Figure 17: Home page

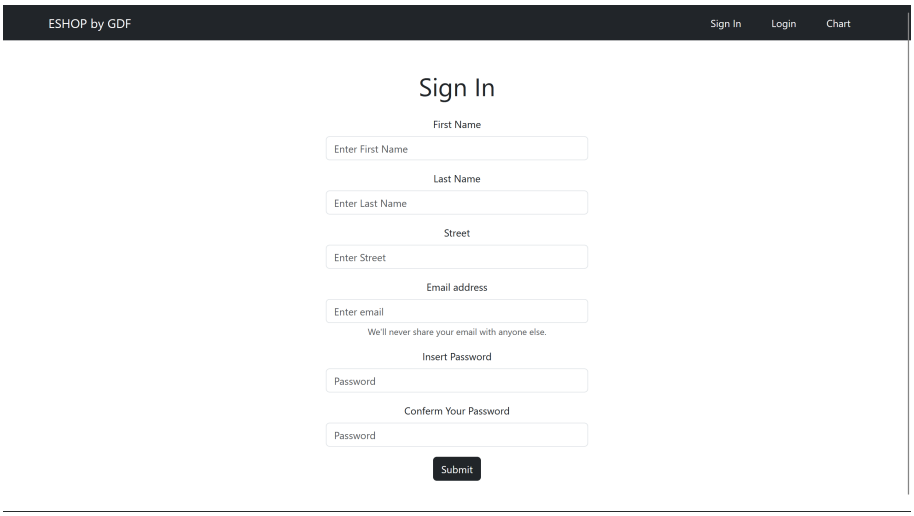

Figure 18: Sign In page

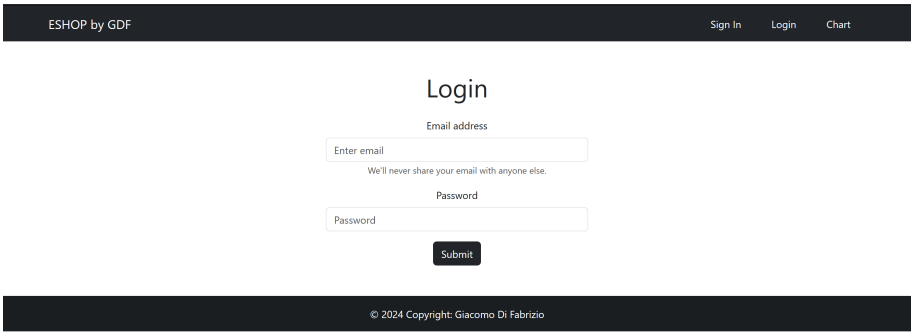

Figure 19: Log In page

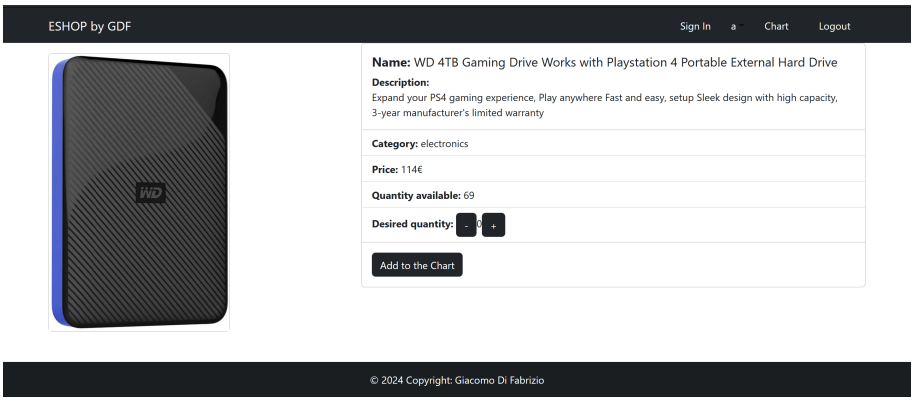

Figure 20: Product page

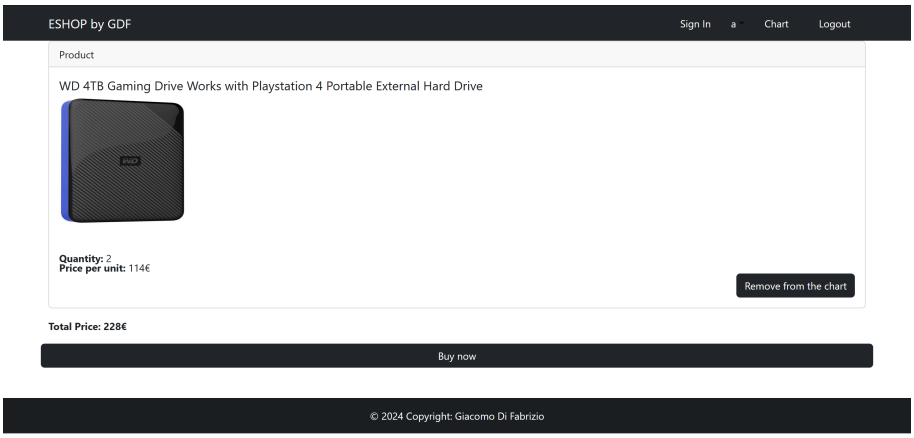

Figure 21: Cart Page

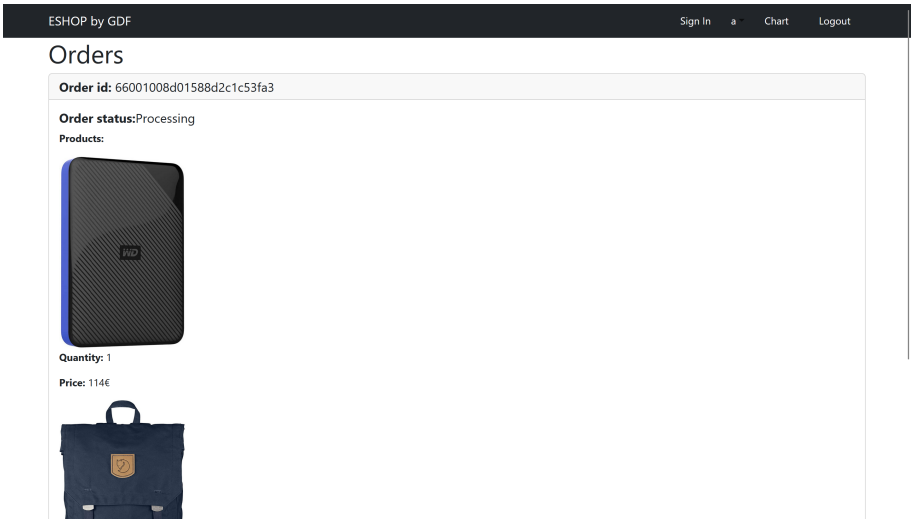

Figure 22: Orders Page

Figure 23: Profile Page

Implementation of the queries in the DB.

Provided below are some pictures showing the queries done inside the routes.

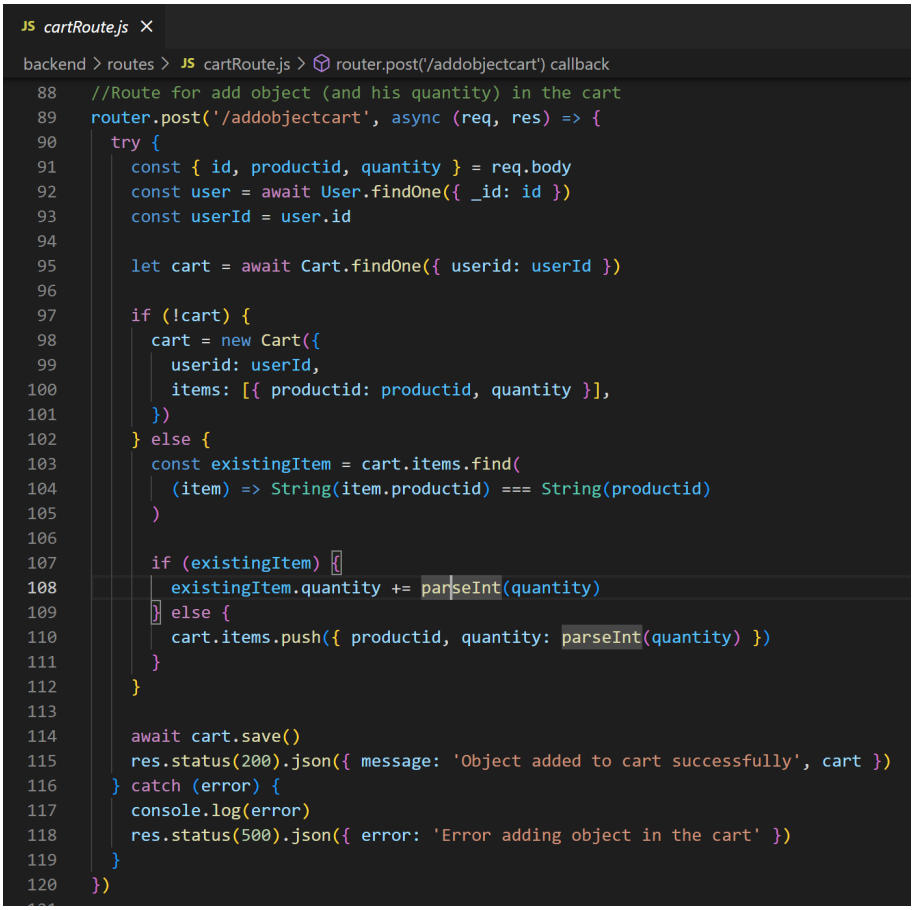

Figure 24: Query for add an object to the cart

Figure 25: Query for add an order

This is another example of the check the correctness of the query for the registration of an user.

```
_id: ObjectId('66000fc3d01588d2c1c53f35')
firstname: "a"
lastname: "a"
email: "a"
password: "$2b$10$Spx3ja2d2jl97Ur9pyGXV.DpKlYuDElAwlI976Fje7ExbnrGIeLuy"
street: "a"
money: 666.1
isAdmin: false
__v: 0
```


Figure 26: Result of the query for the user registration