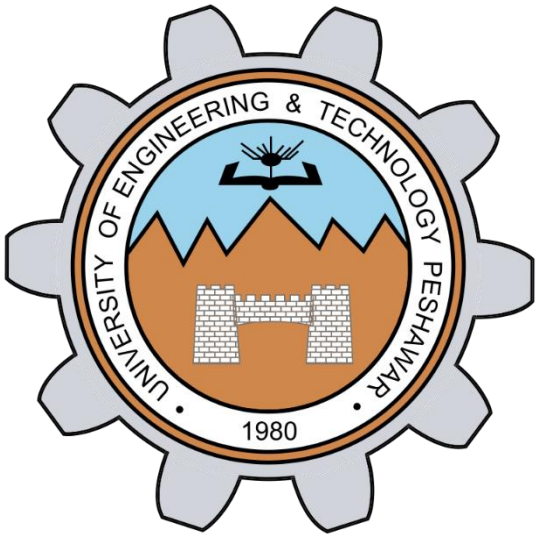
UNIVERSITY OF ENGINEERING AND TECHNOLOGY, PESHAWAR
PAKISTAN

Main Campus



A.I Assingment 03

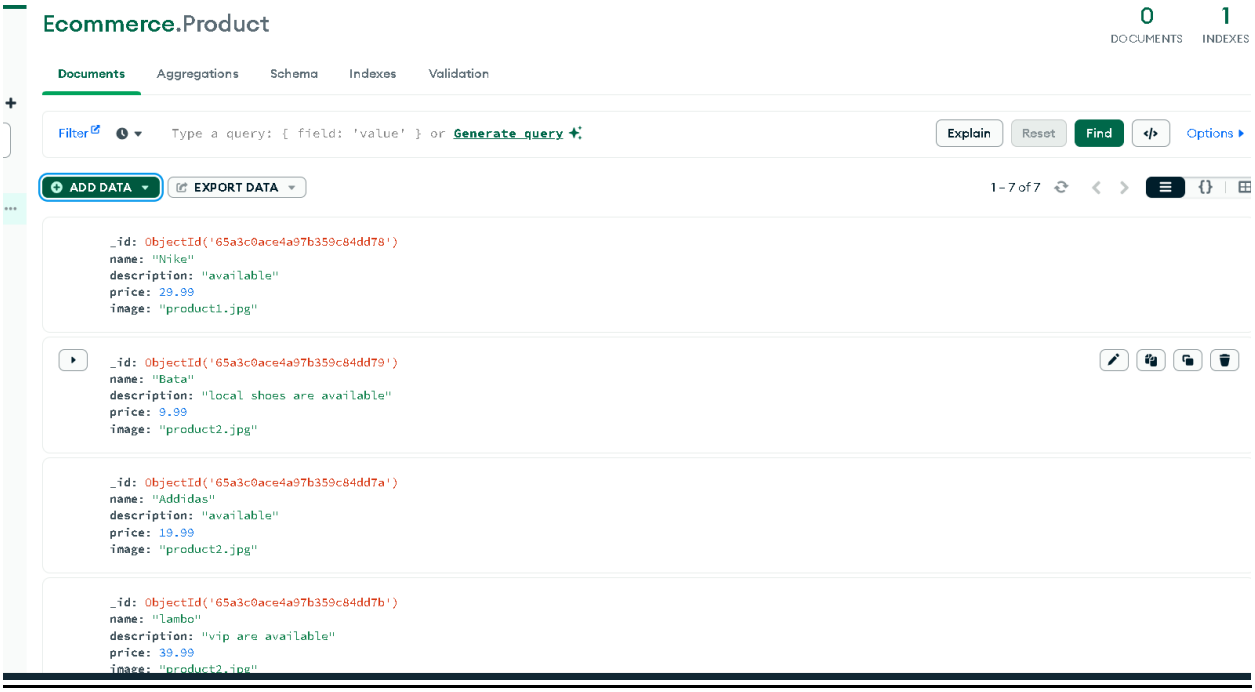
Submitted By

Name: **Umair Abbas**
Registration No. **Reg_No:21pwdsc0018**
Semester: **BS DS 5th**
Section: **Data Science**
Submission Date: **1/14/2024**

Submitted To
Mr. Mohammad

DEPARTMENT OF COMPUTER SCIENCE & IT
UNIVERSITY OF ENGINEERING AND TECHNOLOGY, PESHAWAR, PAKISTAN

Ecommerce Backend



This is a mongodb database where I use online connection and create my ecommerce database and make a collection “product” in it.

```
import mongoose from 'mongoose';

const shoestSchema = mongoose.Schema(
  {
    name: {
      type: String,
      required: true,
    },
    description: {
      type: String,
      required: true,
    },
    price: {
      type: Number,
      required: true,
    },
    image: {
      type: String,
      required: true,
    },
  },
  { ...
  }
);

export const Shoes = mongoose.model('Shoes', shoestSchema);
```

This is my shoesmodel.js , in this file I have created a schema for my products that I have add to my database .Here I use to functions mongoose.schema and mongoose.model.

```
// userModel.js
import mongoose from 'mongoose';

const userSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  role: { type: String, enum: ['user', 'admin'], default: 'user' },
});

export const User = mongoose.model('User', userSchema);
export default User;
```

This my usermodel.js file where I have created a user ‘admin’ schema and use two functions

Mongoose.schema and mongoose.model.

```
const isAdmin = (req, res, next) => {
    if (req.user && req.user.role === 'admin') {
      return next();
    }
    res.status(403).send({ message: 'Unauthorized' });
  };

  export { isAdmin };
```

An athuntication middleware for admin to check the user is admin or not. The isAdmin middleware is used to ensure that only users with the 'admin' role can access the '/admin-only-route'. If a user without admin privileges tries to access the route, they will receive a 403 "Unauthorized" response.

```
// authRoute.js
import express from 'express';
import bcrypt from 'bcrypt';
import User from '../models/userModel.js';

const authRouter = express.Router();

// User Signup
authRouter.post('/signup', async (req, res) => {
    try {
      const { username, password } = req.body;

      if (!username || !password) {
        return res.status(400).send({ message: 'Username and password are required' });
      }

      // ... your signup logic
      res.status(201).send({ message: 'User created successfully' });
    } catch (error) {
      console.error(error.message);
      res.status(500).send({ message: error.message });
    }
  });
```

Express Router: authRouter is an instance of Express Router, which is used to create modular, mountable route handlers.

User Signup Route: The route is defined for the HTTP POST method at the path '/signup'. This route is intended for user registration.

Request Handling: The route handler is an asynchronous function that uses async and await for handling promises.

Request Body: The req.body is expected to contain a JSON object with username and password properties.

Validation: It checks if both username and password are present in the request body. If either is missing, it responds with a 400 Bad Request status and a message indicating that both username and password are required.

Signup Logic: The actual signup logic is marked with a comment (// ... your signup logic). This is where you would typically interact with your database or authentication service to create a new user.

Response: If the signup logic is successful, it responds with a 201 Created status and a message indicating that the user was created successfully.

```
// authRoute.js
authRouter.post('/login', async (req, res) => {
    try {
      const { username, password } = req.body;
      const user = await User.findOne({ username });

      if (!user) {
        return res.status(404).send({ message: 'User not found' });
      }

      // Compare plain text passwords (not recommended for production)
      const isPasswordValid = password === user.password;

      if (!isPasswordValid) {
        return res.status(401).send({ message: 'Invalid credentials' });
      }

      res.status(200).send({ message: 'Login successful' });
    } catch (error) {
      console.error(error.message);
      res.status(500).send({ message: error.message });
    }
  });

export default authRouter;
```

User Login Route: This route is for handling user login using the HTTP POST method at the path '/login'.

Request Handling: The route handler is an asynchronous function that uses async and await for handling promises.

Request Body: The req.body is expected to contain a JSON object with username and password properties.

User Retrieval: It attempts to find a user in the database based on the provided username.

Validation and Authentication: If the user is not found, it responds with a 404 status and a message indicating that the user was not found. It compares the provided password with the stored password in the user object. If the password is not valid, it responds with a 401 status and a message indicating invalid credentials.

Successful Login: If both the user is found and the password is valid, it responds with a 200 status and a message indicating a successful login.

Error Handling: If there's an error during the login process, it logs the error message and responds with a 500 Internal Server Error status along with the error message.

```
// Route for Save a new Shoes
router.post('/', async (request, response) => {
  try {
    if (
      !request.body.name ||
      !request.body.description ||
      !request.body.price ||
      !request.body.image
    ) {
      return response.status(400).send({
        message: 'Send all required fields: name, description, price, image',
      });
    }
    const newshoes = {
      name: request.body.name,
      description: request.body.description,
      price: request.body.price,
      image: request.body.image,
    };

    const shoes = await Shoes.create(newshoes);

    return response.status(201).send(shoes);
  } catch (error) {
    console.log(error.message);
    response.status(500).send({ message: error.message });
  }
});
```

Route for Saving a New Pair of Shoes (Create):

Path: /

Method: POST

Description: Creates a new pair of shoes in the database.

Validation: Checks if all required fields (name, description, price, image) are present in the request **body.**

Response:

If successful, responds with a 201 status and the created shoes.

If there are missing fields, responds with a 400 status and a message indicating the required fields.

```
// Route for Get One Shoes from database by id
router.get('/:id', async (request, response) => {
  try {
    const { id } = request.params;

    const shoes = await Shoes.findById(id);

    return response.status(200).json(shoes);
  } catch (error) {
    console.log(error.message);
    response.status(500).send({ message: error.message });
  }
});

// Route for Update a Shoes
router.put('/:id', async (request, response) => {
  try {
    if (
      !request.body.name ||
      !request.body.description ||
      !request.body.price ||
      !request.body.image
    ) {
      return response.status(400).send({
        message: 'Send all required fields: name, description, price, image',
      });
    }

    const { id } = request.params;

    const result = await Shoes.findByIdAndUpdate(id, request.body);

    if (!result) {
      return response.status(404).json({ message: 'shirt not found' });
    }

    return response.status(200).send({ message: 'shirt updated successfully' });
  } catch (error) {
    console.log(error.message);
    response.status(500).send({ message: error.message });
  }
});
```

Route for Getting All Shoes from the Database (Read):

Path: /

Method: GET

Description: Retrieves all shoes from the database.

Response:

If successful, responds with a 200 status, the count of shoes, and an array of shoes.

If there's an error, responds with a 500 status and an error message.

Route for Getting One Pair of Shoes by ID from the Database (Read):

Path: /:id

Method: GET

Description: Retrieves a specific pair of shoes by its ID from the database.

Response:

If successful, responds with a 200 status and the details of the requested shoes.

If the shoes are not found, responds with a 404 status and a message indicating that the shoes were not found.

If there's an error, responds with a 500 status and an error message.

```
// Route for Delete a shoes
router.delete('/:id', async (request, response) => {
  try {
    const { id } = request.params;

    const result = await Shoes.findByIdAndDelete(id);

    if (!result) {
      return response.status(404).json({ message: 'shoes not found' });
    }

    return response.status(200).send({ message: 'shoes removed successfully' });
  } catch (error) {
    console.log(error.message);
    response.status(500).send({ message: error.message });
  }
});

export default router;
```

Route for Deleting a Pair of Shoes (Delete):

Path: /:id

Method: DELETE

Description: Deletes a pair of shoes by its ID from the database.

Response:

If successful, responds with a 200 status and a message indicating that the shoes were removed successfully.

If the shoes are not found, responds with a 404 status and a message indicating that the shoes were not found.

If there's an error, responds with a 500 status and an error message.

```
// addProducts.js
import mongoose from 'mongoose';
import { mongoDBURL } from './config.js';
import { Shoes } from './models/shoesModel.js';

mongoose
  .connect(mongoDBURL, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(async () => {
    console.log('Connected to the database');

    // Add new products
    const newProducts = [
      {
        name: 'Nike',
        description: 'available',
        price: 29.99,
        image: 'product1.jpg',
      },
      {
        name: 'Bata ',
        description: 'local shoes are available',
        price: 9.99,
        image: 'product2.jpg',
      },
      {
        name: 'Addidas',
        description: 'available',
        price: 19.99,
        image: 'product2.jpg',
      },

      {
        name: 'lambo  ',
        description: 'vip are available',
        price: 39.99,
        image: 'product2.jpg',
      },

      {
        name: 'gucci',
        description: 'Man shoes are available',
        price: 29.99,
        image: 'product2.jpg',
      },

      {
        name: 'lambo  ',
        description: 'vip are available',
        price: 39.99,
        image: 'product2.jpg',
      },

      {
        name: 'Liverpool',
        description: 'shoes are available',
        price: 21.99,
        image: 'product2.jpg',
      },

      {
        name: 'locdsaa ',
        description: 'available',
        price: 4.99,
        image: 'product2.jpg',
      },
      ];

    const insertedProducts = await Shoes.insertMany(newProducts);
    console.log('New products added:', insertedProducts);

    // Close the database connection
    mongoose.connection.close();
  })
  .catch((error) => {
    console.error('Error connecting to the database:', error);
```

Connection to the Database:

The code connects to a MongoDB database using the MongoDB connection URL specified in the mongoDBURL variable from the config.js file.

Adding New Products: An array named newProducts contains objects representing new products, each with properties like name, description, price, and image.

Inserting Products into the Database:

The Shoes.insertMany(newProducts) method inserts the new products into the MongoDB collection named 'Shoes'.

Logging Information: The code logs a message indicating that the new products have been added to the database along with details about the inserted products.

Closing the Database Connection: After adding the products, the database connection is closed using mongoose.connection.close().

Error Handling: If there's an error during the connection or product insertion process, the code logs an error message.

config.js > ...

```
1   export const PORT = 5555;
2
3   export const mongoDBURL =  'mongodb+srv://umair:umair123@cluster1.7m1p28n.mongodb.net/';
```

A connection is taken from online Atlas.

```
JS index.js > ...
1   import express from 'express';
2   import { PORT, mongoDBURL } from './config.js';
3   import mongoose from 'mongoose';
4   import shoesRoute from './routes/shoesRoute.js';
5   import cors from 'cors';
6   import bodyParser from 'body-parser';
7   // index.js
8   import authRoute from './routes/authRoute.js';
9   const app = express();
10  app.use(bodyParser.json());
11  app.use('/auth', authRoute);
12  // Middleware for parsing request body
13  app.use(express.json());
14  app.use(cors());
15  app.get('/', (request, response) => {
16    console.log(request);
17    return response.status(234).send('Welcome to My Ecommerce Store');
18  });
19
20  app.use('/Shoes', shoesRoute);
21
22  mongoose
23    .connect(mongoDBURL)
24    .then(() => {
25      console.log('App connected to database');
26      app.listen(PORT, () => {
27        console.log(`App is listening to port: ${PORT}`);
28      });
29    })
30    .catch((error) => {
31      console.log(error);
32    });
33
```

Explanation:

Dependencies: The code imports necessary modules like express, mongoose for MongoDB integration, and additional modules such as body-parser for parsing JSON bodies and cors for handling Cross-Origin Resource Sharing.

Express App Setup: An instance of the Express app is created using const app = express();.

Middleware: body-parser is used to parse JSON request bodies. Note that recent versions of Express have built-in JSON body parsing using express.json().

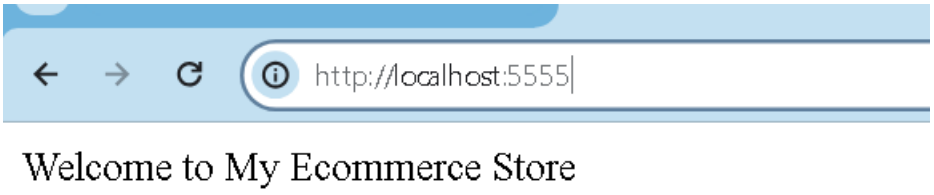
cors middleware is used to enable Cross-Origin Resource Sharing, allowing your server to respond to requests from different origins.

Root Path (/) Handling: A simple route is defined for the root path using app.get('/'). It logs the incoming request object and sends a response with a custom status code (234) and a welcome message.

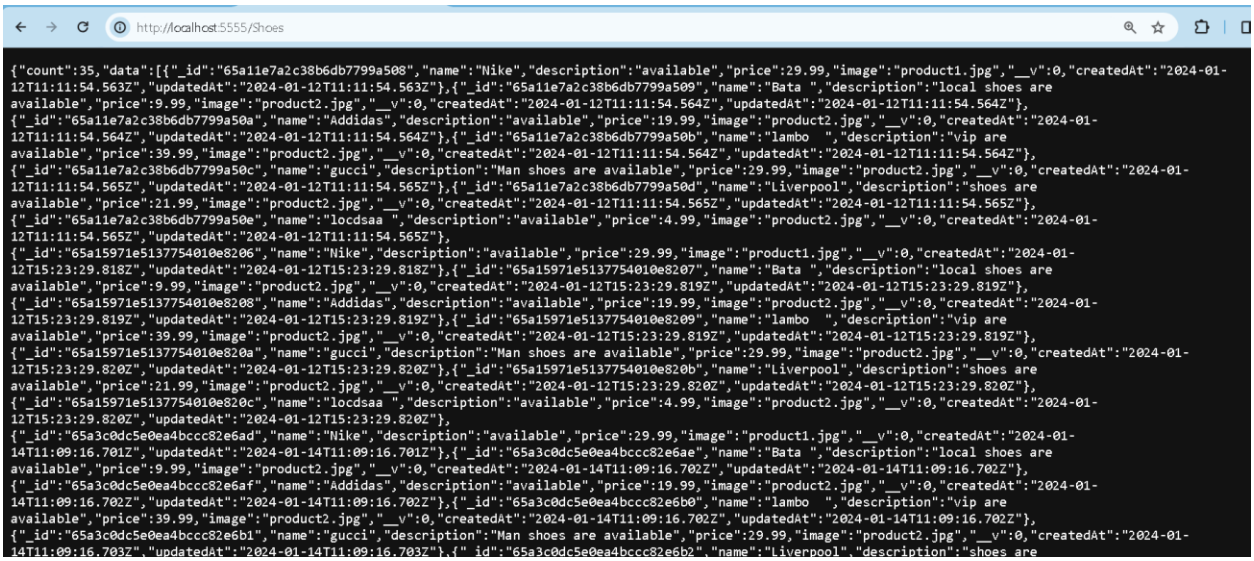
Authentication and Shoes Routes: Two separate routes are defined for handling authentication-related paths (/auth) and shoes-related paths (/Shoes). The actual route handling logic is likely located in the respective authRoute.js and shoesRoute.js files.

Database Connection: Mongoose is used to connect to the MongoDB database using the provided mongoDBURL.

Server Start: If the database connection is successful, the Express app is started using app.listen(PORT, ...), and it listens on the specified port (PORT). The server start logs a message indicating successful connection and the port on which the server is listening.



Checking my website on localhost.



Checking shoes on local host.

```
New products added: [
  {
    name: 'Nike',
    description: 'available',
    price: 29.99,
    image: 'product1.jpg',
    _id: new ObjectId("65a3d8fac4d235ba700856aa"),
    __v: 0,
    createdAt: 2024-01-14T12:52:10.203Z,
    updatedAt: 2024-01-14T12:52:10.203Z
  },
  {
    name: 'Bata ',
    description: 'local shoes are available',
    price: 9.99,
    image: 'product2.jpg',
    _id: new ObjectId("65a3d8fac4d235ba700856ab"),
    __v: 0,
    createdAt: 2024-01-14T12:52:10.204Z,
    updatedAt: 2024-01-14T12:52:10.204Z
  },
  {
    name: 'Addidas',
```

Some of products that I have inserted in my data base