ECE
ÉCOLE D'INGÉNIEURS
ENGINEERING SCHOOL

Title:

Design and Implementation of Microservices Architecture Using Synchronous and Asynchronous Communication Mode

Course:

Distributed Systems Security

Institution: ECE Paris

Instructor:

Yaya Doumbia

Academic Year: 2025-2026

Group Member:

OFUNNEKA JENNIFER OKONKWOABUTU

DATE: 19th January 2026

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 Purpose of this Project

The purpose of this lab is to design, implement, and document a microservices architecture with authentication, product and order management, synchronous, and asynchronous communication..

For this project, we will create three microservices:

- Product Service: Stores product information and responds to requests.
- Order Service: Manages orders and communicates with Product-Service.
- Authentication Service: Handles user registration and login with JWT authentication.

Synchronous communication is implemented using HTTP for direct request-response interactions, while asynchronous communication is implemented using RabbitMQ to enable message-based processing between services.

This report is written as an operational step-by-step guide, allowing the entire system to be reproduced from scratch.

1.2 Objectives of the project

Functional Objectives:

1. Build a microservices-based system
2. Store data using MongoDB for each service (Product-Service, Order-Service, Auth-Service)
3. Enable inter-service communication:
    - Synchronous (HTTP + Axios) for immediate product validation during order creation.
    - Asynchronous (RabbitMQ) for decoupled order processing.

1.3 Source Code Reference

The complete source code for this project is available in a public Github repository and can be used to reproduce the system described in this report.

Repository Link:
Synchronous Implementation:
https://github.com/JanieAbutu/ECE_Distributed-Systems-Security/tree/main/Projects/Microservices/expMicroservice
Asynchronous Implementation:
https://github.com/JanieAbutu/ECE_Distributed-Systems-Security/tree/main/Projects/Microservices/asynchronous_comm_ms_archit

2.0 ARCHITECTURE TYPES

There are two main types of application architectures.

2.1 Monolithic:

- ➔ Single codebase and database
- ➔ All features are tightly coupled.
- ➔ Simple to deploy initially but hard to scale individual components.

2.2 Microservices:

- ➔ Each service is independent with its own database and logic
- ➔ Services communicate with each other using APIs (HTTP) or messaging systems (RabbitMQ).
- ➔ Easier to scale, update, and maintain individual services without affecting others.

2.3 Interim Conclusion

Microservices enable independent development, deployment, and scaling of services, unlike monolithic architecture that is tightly dependent on each other.

3.0 COMMUNICATION MODE

3.1 Synchronous Communication

- ➔ Services call each other and wait for a response before continuing. (Blocking applies)
- ➔ An example as deployed in this project is Order-Service calling Product-Service via HTTP(axios) to validate products before saving the order.
- ➔ Execution cannot proceed until response is received

3.2 Asynchronous Communication

- ➔ Services send messages and do not wait for immediate response, they continue.
- ➔ An example is Order-Service sending productIDs to Product-Service via RabbitMQ, allowing other tasks to continue even if product-service is unavailable.

In this project,the two communication modes (Synchronous and Asynchronous) would be explored and demonstrated.

3.3 Interim Conclusion

Synchronous communication ensures immediate validation but blocks service execution until a response arrives, whereas asynchronous communication allows services to continue processing other tasks while waiting for responses.

4.0 DEVELOPMENT ENVIRONMENT CONFIGURATION

4.1 Tools and Technologies Used

Node.js:
This is used as the JavaScript runtime to build all microservices.

Express.js
This provides routing, middleware support, and HTTP request handling.

MongoDB
This is a NoSQL document-oriented database used for data persistence.

Mongoose
This is an Object Data Modeling (ODM) library that provides schemas, validation, and database abstraction.

Nodemon
Development tool for auto-restarting services

JSON Web Token (JWT)
This is used for stateless authentication and authorization.

bcryptjs
This is used for password hashing.

Axios
The HTTP client that is used for synchronous service-to-service communication.

dotenv
Manages environment variables.

RabbitMQ:
This is the message broker that enables asynchronous communication.

amqplib:
The Node.js client library for interacting with RabbitMQ

Erlang:

Postman
This is used to test and verify API endpoints.

Summary:
Each tool was selected to address a specific responsibility within the system.

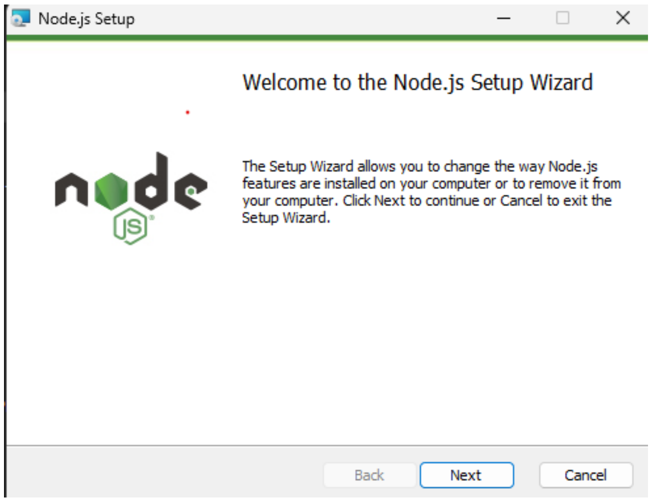
4.2 Environment setup

This section describes how the development environment was configured before starting the project. Proper environment setup is critical for the microservices to run correctly, dependencies are managed consistently, and errors related to tooling are avoided.

Step 1: Install Required Software

1. **Node.js includes npm (Node Package Manager):**

- Download LTS version (recommended) from https://nodejs.org/en/



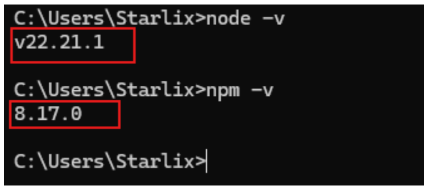
Node.js Setup

Welcome to the Node.js Setup Wizard

The Setup Wizard allows you to change the way Node.js features are installed on your computer or to remove it from your computer. Click Next to continue or Cancel to exit the Setup Wizard.

Fig 4.1: Node.js installation interface

- Verify the installation of Node.js by running the commands below on command prompt using node -v and npm -v

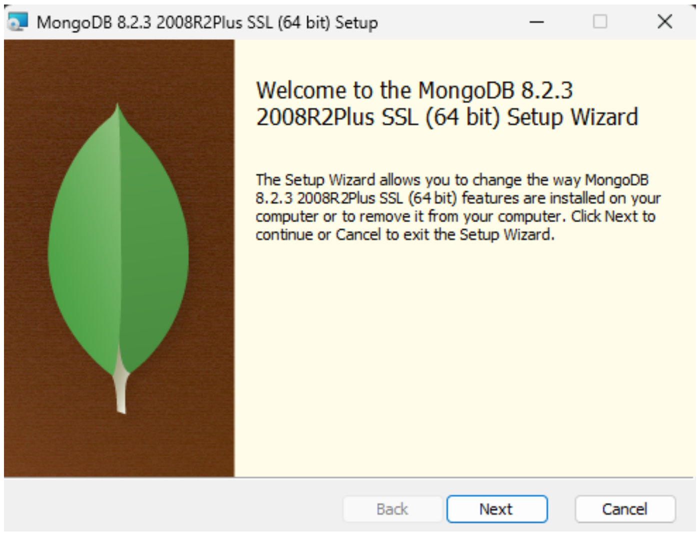


```
C:\Users\Starlix>node -v
v22.21.1

C:\Users\Starlix>npm -v
8.17.0

C:\Users\Starlix>
```

Fig 4.2: Output of command verifying version of node.js and npm installed

2. **MongoDB Community Server**

- Download MongoDB Community Server from https://www.mongodb.com/ and install.



MongoDB 8.2.3 2008R2Plus SSL (64 bit) Setup

Welcome to the MongoDB 8.2.3 2008R2Plus SSL (64 bit) Setup Wizard

The Setup Wizard allows you to change the way MongoDB 8.2.3 2008R2Plus SSL (64 bit) features are installed on your computer or to remove it from your computer. Click Next to continue or Cancel to exit the Setup Wizard.

Fig 4.3:MongoDB Community Server installation interface

-

- The **MongoDB Compass** installs immediately after this and is the graphical user interface as seen below:

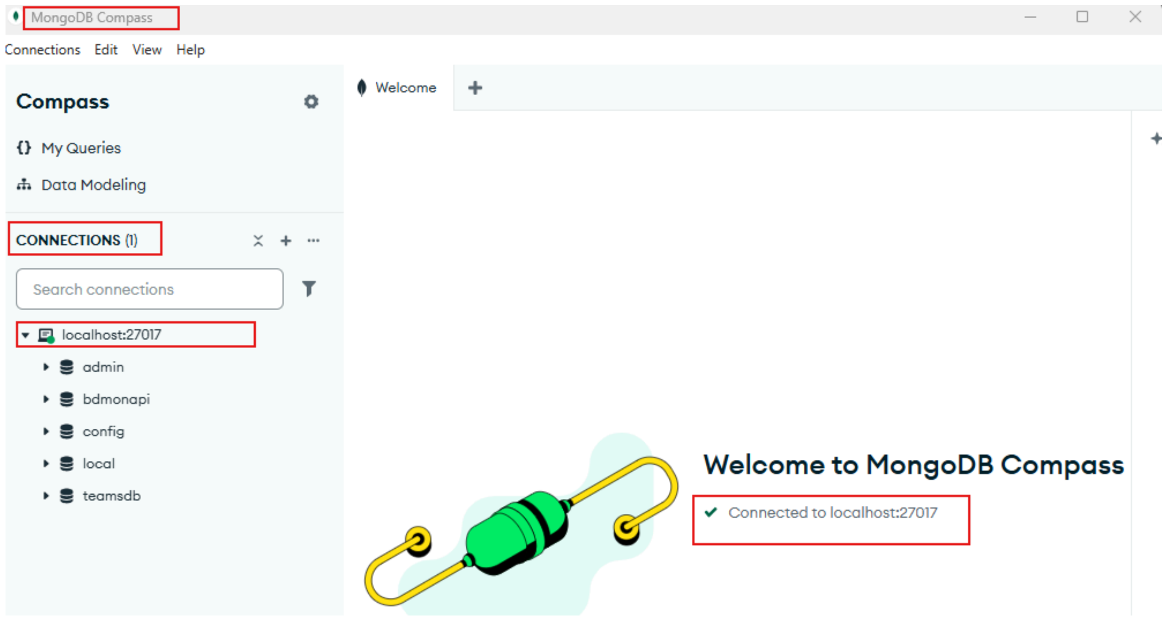


Fig 4.4:MongoDB Compass interface

- After installation, open MongoDB Compass, then click on 'Localhost:27017' to connect. Successful connection will be confirmed as seen in above image.

3. <u>Postman</u>

- Download the postman application from https://www.postman.com/ and install. Register, login, and create a workspace to get started.

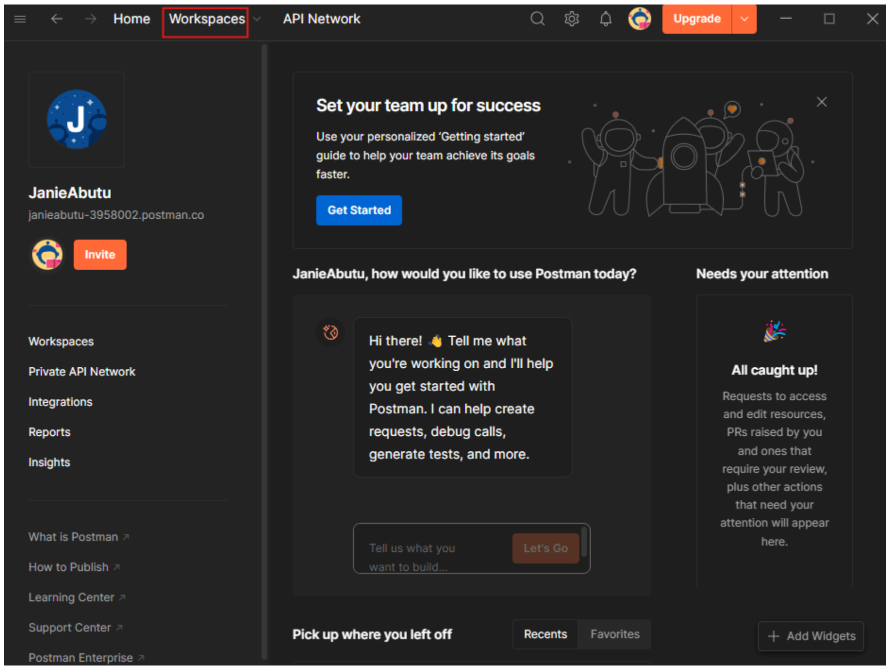
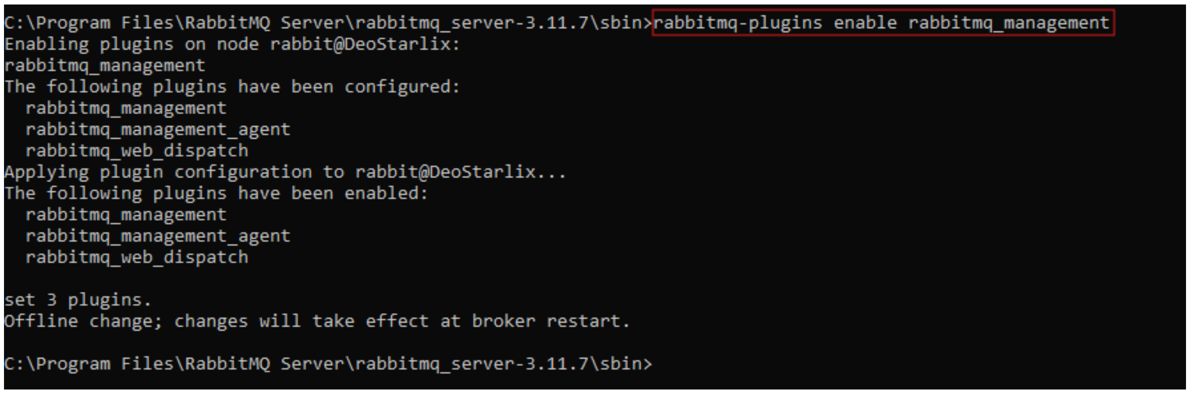

Home
Workspaces
API Network
Upgrade
JanieAbutu
janieabutu-3958002.postman.co
Invite
Workspaces
Private API Network
Integrations
Reports
Insights
What is Postman
How to Publish
Learning Center
Support Center
Postman Enterprise
Set your team up for success
Use your personalized 'Getting started' guide to help your team achieve its goals faster.
Get Started
JanieAbutu, how would you like to use Postman today?
Hi there! Tell me what you're working on and I'll help you get started with Postman. I can help create requests, debug calls, generate tests, and more.
Tell us what you want to build...
Let's Go
Needs your attention
All caught up!
Requests to access and edit resources, PRs raised by you and ones that require your review, plus other actions that need your attention will appear here.
Pick up where you left off
Recents
Favorites
Add Widgets

Fig 4.5:Postman interface

4. **Erlang**
   Download from link and install as administrator **Release OTP 25.3.2.21 · erlang/otp**

5. **RabbitMQ**

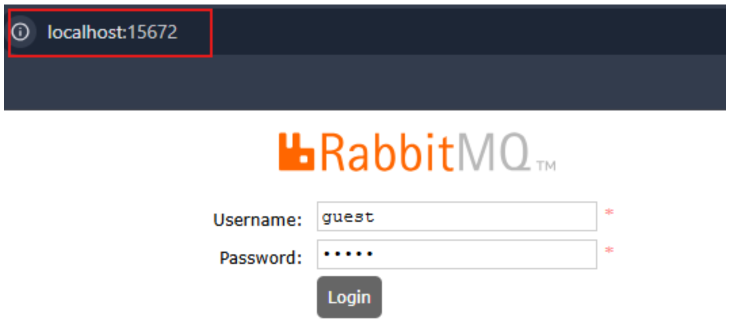
- Download LTS version (recommended) from Installing on Windows | RabbitMQ and install. After installation, enable management plugin as seen below:



```
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.11.7\sbin>rabbitmq-plugins enable rabbitmq_management
Enabling plugins on node rabbit@DeoStarlix:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@DeoStarlix...
The following plugins have been enabled:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch

set 3 plugins.
Offline change; changes will take effect at broker restart.

C:\Program Files\RabbitMQ Server\rabbitmq_server-3.11.7\sbin>
```

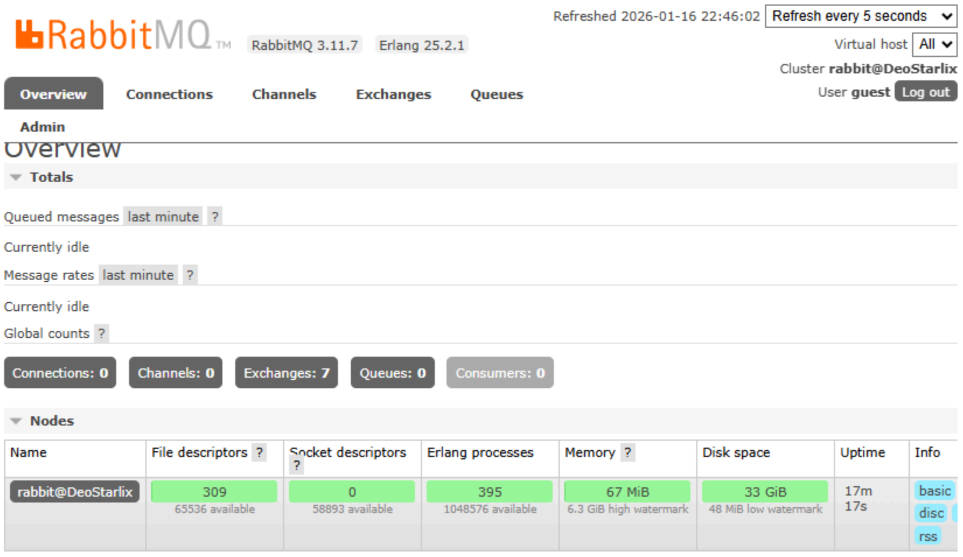
Fig 4.6: Management plugin enabled for RabbitMQ to be functional

- Restart RabbitMQ service and open via http://localhost:15672



Username: guest
Password: •••••
Login

Fig 4.7: Login page to Management Interface via localhost:15672



RabbitMQ 3.11.7 Erlang 25.2.1

Refreshed 2026-01-16 22:46:02 Refresh every 5 seconds
Virtual host All
Cluster rabbit@DeoStarlix
User guest

Overview

Totals

Queued messages last minute ?

Currently idle

Message rates last minute ?

Currently idle

Global counts ?

Connections: 0 Channels: 0 Exchanges: 7 Queues: 0 Consumers: 0

Nodes

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Info
rabbit@DeoStarlix	309 65536 available	0 58893 available	395 1048576 available	67 MiB 6.3 GiB high watermark	33 GiB 48 MiB low watermark	17m 17s	basic disc rss

Fig 4.8: Management interface via localhost:15672 after login

Step 2: Create the Project Workspace and Subfolders

- Create a dedicated folder to contain all application files, all source code, configuration files, and dependencies for one Node.js project. The name of the folder created for this project is ***Microservices/expMicroservice***
- Create subfolders for the 3 microservices (auth-service, order-service, product-service)
- Open the folder in a code editor (Visual Studio - VS Code recommended).

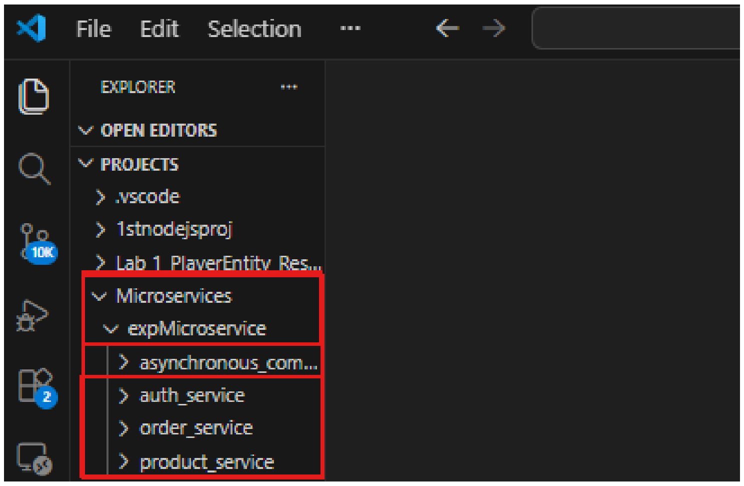
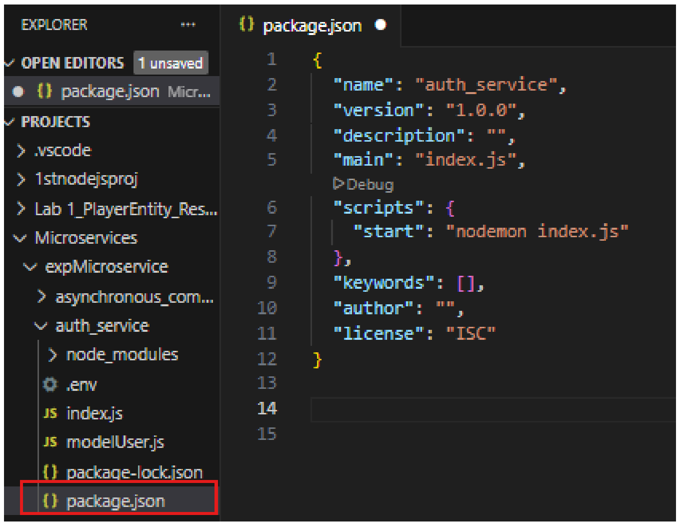


Fig 4.9:Folder ***Microservices/expMicroservice and subfolders*** opened on VS Code

Step 3: Initialize Node.js Project and install dependencies

- On the VS Code terminal navigate to the different subfolders and run the command **npm init** in each subfolder.
- This initializes the Node.js project and generates a package.json file which describes the project, lists the installed dependencies, and defines the application entry point as seen below



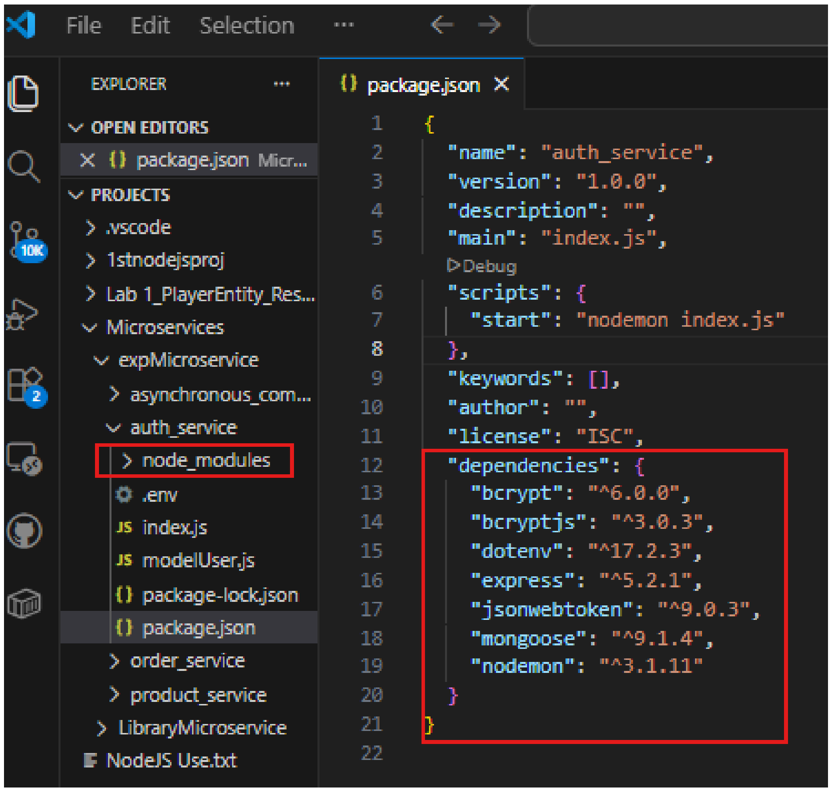
```json
{
  "name": "auth_service",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷Debug
  "scripts": {
    "start": "nodemon index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Fig 8: package.json file generated after initialization (This appears in all the 3 subfolders where the command npm init was run)

Install Project Dependencies

- The following dependencies are installed locally for the project.
  - express: a web framework
  - mongoose: mongoDB ODM
  - jsonwebtoken: jwt authentication
  - bcryptjs: password hashing
  - dotenv
  - amqplib
  - axios: service to service HTTP communication
  - nodemon

- The dependencies are stored in node_modules/ and referenced in package.json as seen below:

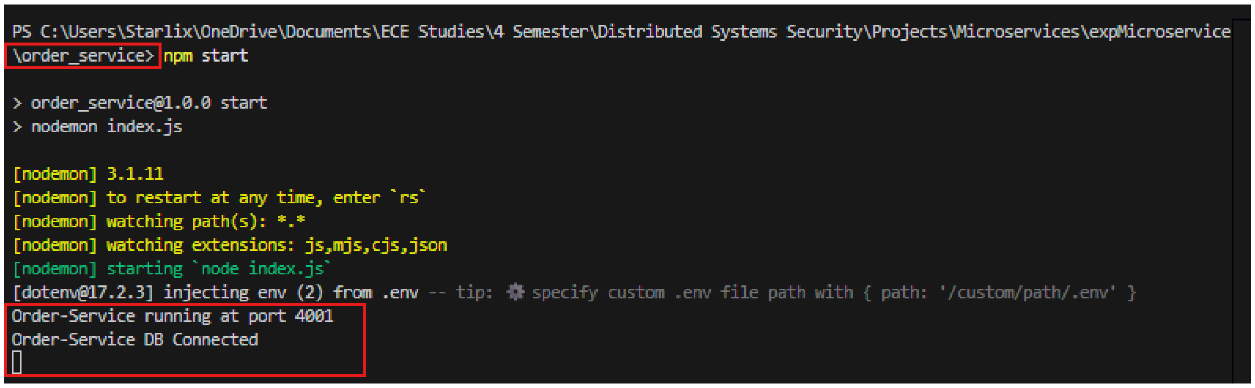


```json
{
  "name": "auth_service",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bcrypt": "^6.0.0",
    "bcryptjs": "^3.0.3",
    "dotenv": "^17.2.3",
    "express": "^5.2.1",
    "jsonwebtoken": "^9.0.3",
    "mongoose": "^9.1.4",
    "nodemon": "^3.1.11"
  }
}
```

Fig 9: node_modules and Installed dependencies referenced in the package.json

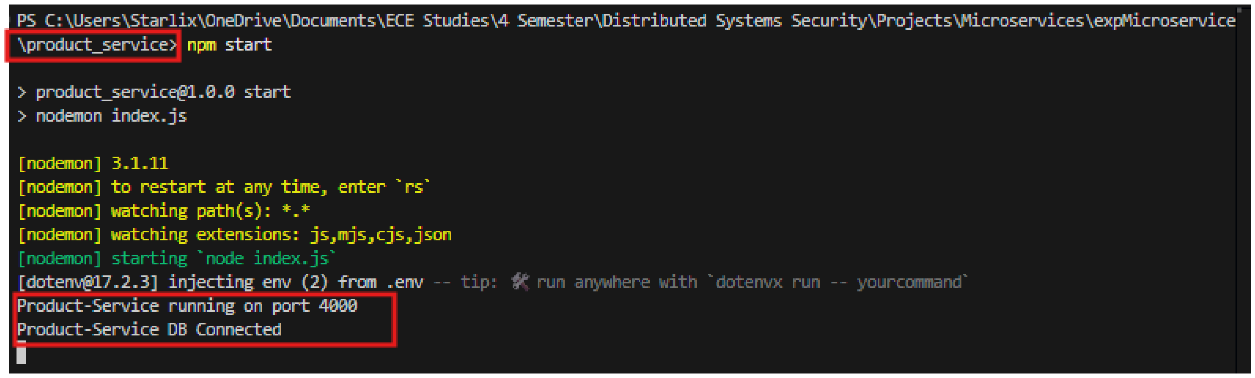
Step 5: Verify Environment Configuration

- The command ***npm start*** was run to test the configuration.
- Node.js server running without errors, mongoDB connecting as seen below confirms the environment configuration was successfully done.



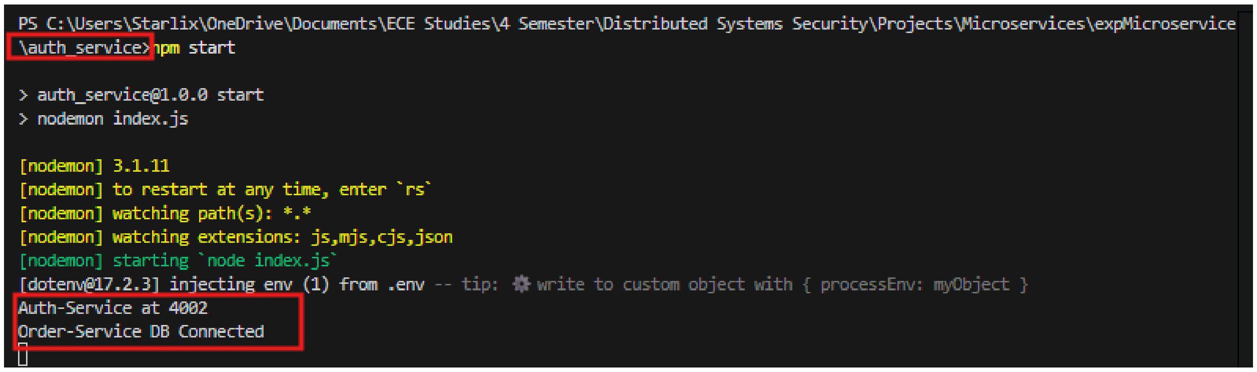
```
PS C:\Users\Starlix\OneDrive\Documents\ECE Studies\4 Semester\Distributed Systems Security\Projects\Microservices\expMicroservice
\order_service> npm start

> order_service@1.0.0 start
> nodemon index.js

[nodemon] 3.1.11
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
[dotenv@17.2.3] injecting env (2) from .env -- tip: ⚙ specify custom .env file path with { path: '/custom/path/.env' }
Order-Service running at port 4001
Order-Service DB Connected
```

Fig 10: Successful server and database connection for order-service



```
PS C:\Users\Starlix\OneDrive\Documents\ECE Studies\4 Semester\Distributed Systems Security\Projects\Microservices\expMicroservice
\product_service> npm start

> product_service@1.0.0 start
> nodemon index.js

[nodemon] 3.1.11
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
[dotenv@17.2.3] injecting env (2) from .env -- tip: 🛠 run anywhere with `dotenvx run -- yourcommand`
Product-Service running on port 4000
Product-Service DB Connected
```

Fig 11: Successful server and database connection for product-service



```
PS C:\Users\Starlix\OneDrive\Documents\ECE Studies\4 Semester\Distributed Systems Security\Projects\Microservices\expMicroservice
\auth_service> npm start

> auth_service@1.0.0 start
> nodemon index.js

[nodemon] 3.1.11
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
[dotenv@17.2.3] injecting env (1) from .env -- tip: ⚙ write to custom object with { processEnv: myObject }
Auth-Service at 4002
Order-Service DB Connected
```

Fig 12: Successful server and database connection for auth-service (why is auth service connected to order service db)

4.3 Conclusion

For this, all services can run independently and connect with their databases. This configuration is essential before implementation as it ensures that errors are minimized during development, and that the application can be executed and tested reliably.

5.0 GLOBAL SYSTEM ARCHITECTURE

This section explains how responsibilities are divided inside the application for structure, clarity, and to enforce clean code organisation and predictable behaviour.

3.1 Architectural Layers

1. Client (Postman)
2. Service Layer
   - ➔ Product Service
   - ➔ Order Service
   - ➔ Authentication Service
3. Data Layer
   - ➔ MongoDB databases per service
4. Communication Layer
   - ➔ HTTP (synchronous)
   - ➔ RabbitMQ (Asynchronous)

3.2 Project Structure

Action: Manually create the following folders and files in the order below:

For Synchronous Communication we have the structure as:

```
Microservice/expMicroservice
├── product_service/              # Handles product management and retrieval
│   ├── package.json              # Product service dependencies and scripts
│   ├── modelProduct.js           # Product database schema and model
│   ├── isAuthenticated.js        # Authentication middleware for product routes
│   ├── consumer.js               # Consumes product IDs from RabbitMQ
│   └── index.js                  # Product service application entry point
│
├── order_service/                # Handles order creation and processing
│   ├── package.json              # Order service dependencies and scripts
│   ├── modelOrder.js             # Order database schema and model
│   ├── isAuthenticated.js        # Authentication middleware for order routes
│   ├── producer.js               # Publishes product IDs to RabbitMQ
│   └── index.js                  # Order service application entry point
│
├── auth_service/                 # Handles user authentication and authorization
│   ├── package.json              # Authentication service dependencies
│   ├── modelUser.js              # User database schema and model
│   └── index.js                  # Auth service application entry point
│
```

For Asynchronous Communication we have the structure as:

```
Microservice/expMicroservice
├── product_service/                  # Product service
│   ├── package.json                  # Product service dependencies
│   ├── modelProduct.js               # Product data model
│   ├── isAuthenticated.js            # Authentication middleware
│   ├── consumer.js                   # RabbitMQ consumer
│   └── index.js                      # Product service entry point
│
├── order_service/                    # Order service
│   ├── package.json                  # Order service dependencies
│   ├── modelOrder.js                 # Order data model
│   ├── isAuthenticated.js            # Authentication middleware
│   ├── producer.js                   # RabbitMQ Producer
│   └── index.js                      # Order service entry point
│
├── auth_service/                     # Authentication service
│   ├── package.json                  # Auth service dependencies
│   ├── modelUser.js                  # User data model
│   └── index.js                      # Auth service entry point
```

3.3 Conclusion

The system follows the microservice architecture, where each service owns its own logic, database, and lifecycle. Communication is handled explicitly through HTTP (synchronous) or messaging (Asynchronous).

4.0 DATABASE DESIGN AND DATA MODELING

The database layer stores all application data persistently.

4.1 Entities

Database for product-service with entity ‘products’:

- ➔ Id (Automatically assigned by MongoDB)
- ➔ name
- ➔ description
- ➔ price
- ➔ created at

Database for order-service with entities:

order:

- ➔ Id (Automatically assigned by MongoDB)
- ➔ products (array of product IDs)
- ➔ user_email
- ➔ total_price
- ➔ created_at

users:

- ➔ Id
- ➔ name
- ➔ email
- ➔ password
- ➔ created_at

4.2 Database Creation and Setup

Step 1: Database Creation

The databases are created automatically when the application connects, MongoDB creates the database.

Step 2: Collections Creation

Collections are automatically created by Mongoose when documents are inserted:

- ➔ products
- ➔ orders
- ➔ users

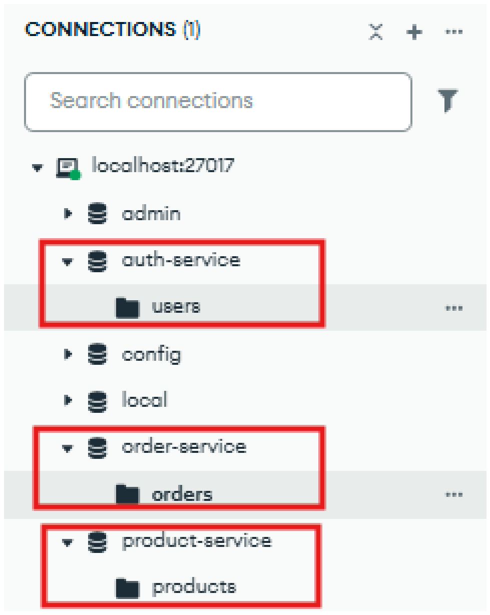


Fig 4.2: Created Databases and Collections on MongoDB (auth-service, order-service and product-service database with their respective collections under them)

4.3 Implement Database Models (Schemas)

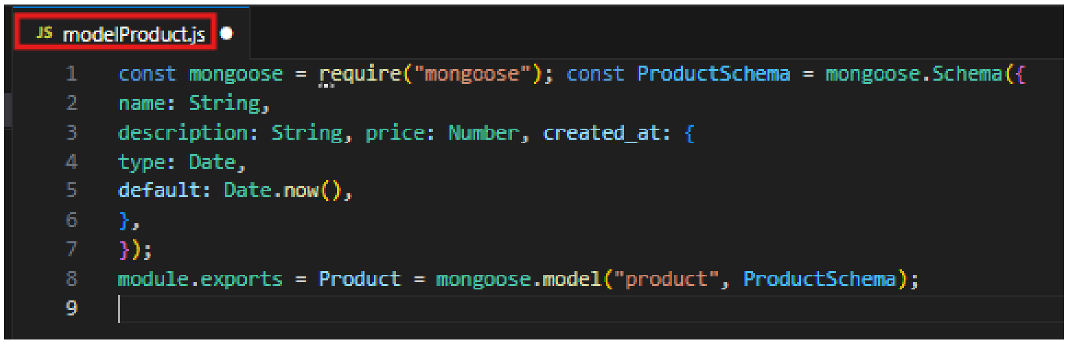
Objective: Define how data is stored and validated in MongoDB.

Step 1: Product Model (modelProduct.js)

Purpose - Represent a product in the product-service database.

Actions Performed:

- Create modelProduct.js
- Define fields: name, description, price, created_at



```
const mongoose = require("mongoose"); const ProductSchema = mongoose.Schema({
name: String,
description: String, price: Number, created_at: {
type: Date,
default: Date.now(),
},
});
module.exports = Product = mongoose.model("product", ProductSchema);
```

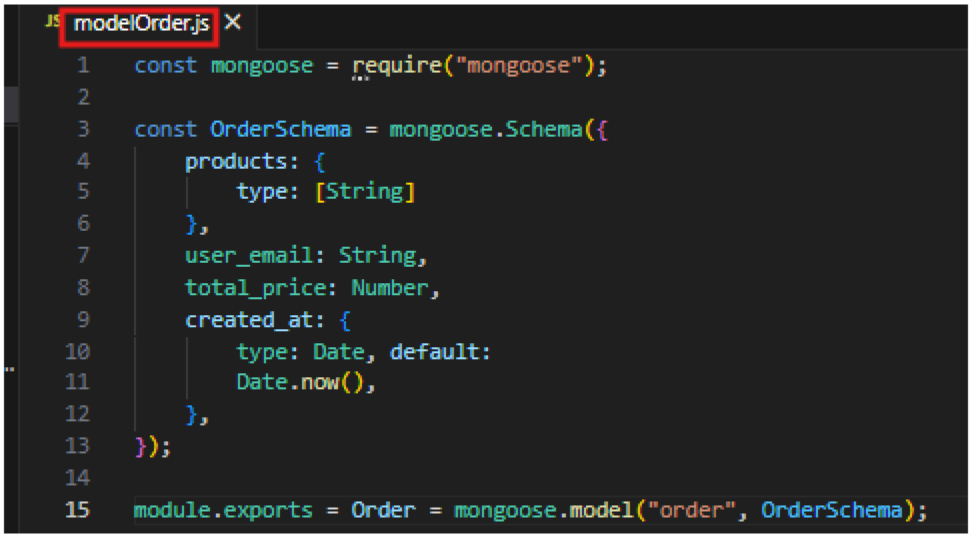
Fig 4.3.1: modelProduct.js Schema Code

Step 2: Order Model (modelOrder.js)

Purpose - Represent an order in the order-service database

Actions Performed:

- Create **modelOrder.js**
- Define fields: products, user_email, total_price, created_at


modelOrder.js

```
const mongoose = require("mongoose");

const OrderSchema = mongoose.Schema({
    products: {
        type: [String]
    },
    user_email: String,
    total_price: Number,
    created_at: {
        type: Date, default:
        Date.now(),
    },
});

module.exports = Order = mongoose.model("order", OrderSchema);
```

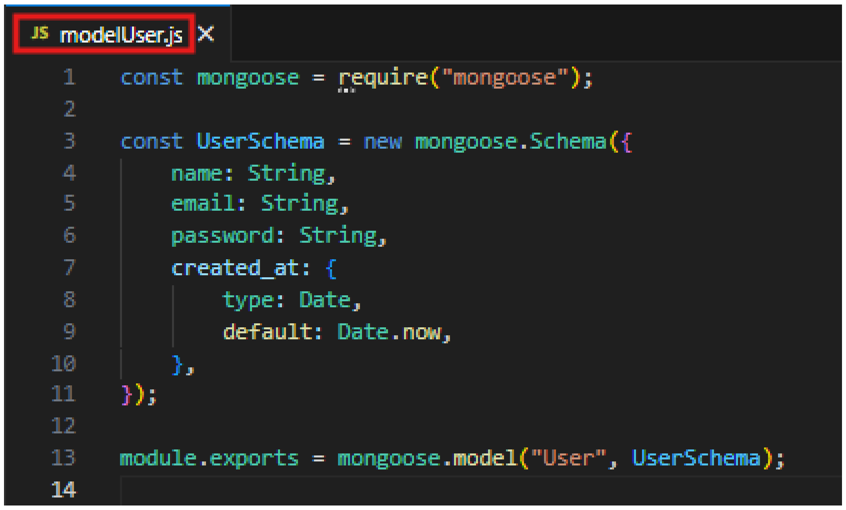
Fig 4.3.2: modelOrder.js Schema Code

Step 3: User Model (modelUser.js)

Purpose - Stores users.

Actions Performed:

- Create modelUser.js
- Define fields: name, email, password, created_at


modelUser.js

```
const mongoose = require("mongoose");

const UserSchema = new mongoose.Schema({
    name: String,
    email: String,
    password: String,
    created_at: {
        type: Date,
        default: Date.now,
    },
});

module.exports = mongoose.model("User", UserSchema);

```

Fig 4.3.3: modelUser.js Schema Code

4.4 Conclusion:

Database isolation ensures service independence and avoids tight coupling. Relationships are maintained through identifiers rather than joins which aligns with the best practices for distributed systems.

5.0 SERVICES IMPLEMENTATION (SYNCHRONOUS COMMUNICATION MODE)

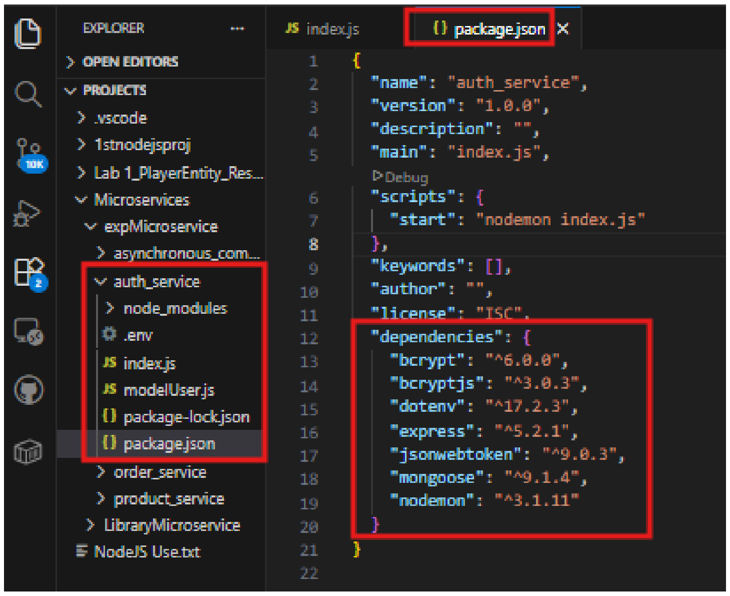
5.1 Authentication Service Implementation

Objective: Implement authentication and user management functionality using JWT to protect microservice endpoints.

Operations Supported:

➔ User registration
➔ User login
➔ Token verification

Step 1 - Confirm Auth Service Project Directory
'Microservice/expMicroservice/auth_service/' already created during the environment setup section and dependencies installed and are as below:



```json
{
  "name": "auth_service",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bcrypt": "^6.0.0",
    "bcryptjs": "^3.0.3",
    "dotenv": "^17.2.3",
    "express": "^5.2.1",
    "jsonwebtoken": "^9.0.3",
    "mongoose": "^9.1.4",
    "nodemon": "^3.1.11"
  }
}
```

Fig 5.1.1: *Microservice/expMicroservice/auth_service/* directory with the dependencies needed for auth_service.

Step 2: Verify product data model modelUser.js defined in the database creation and setup section.

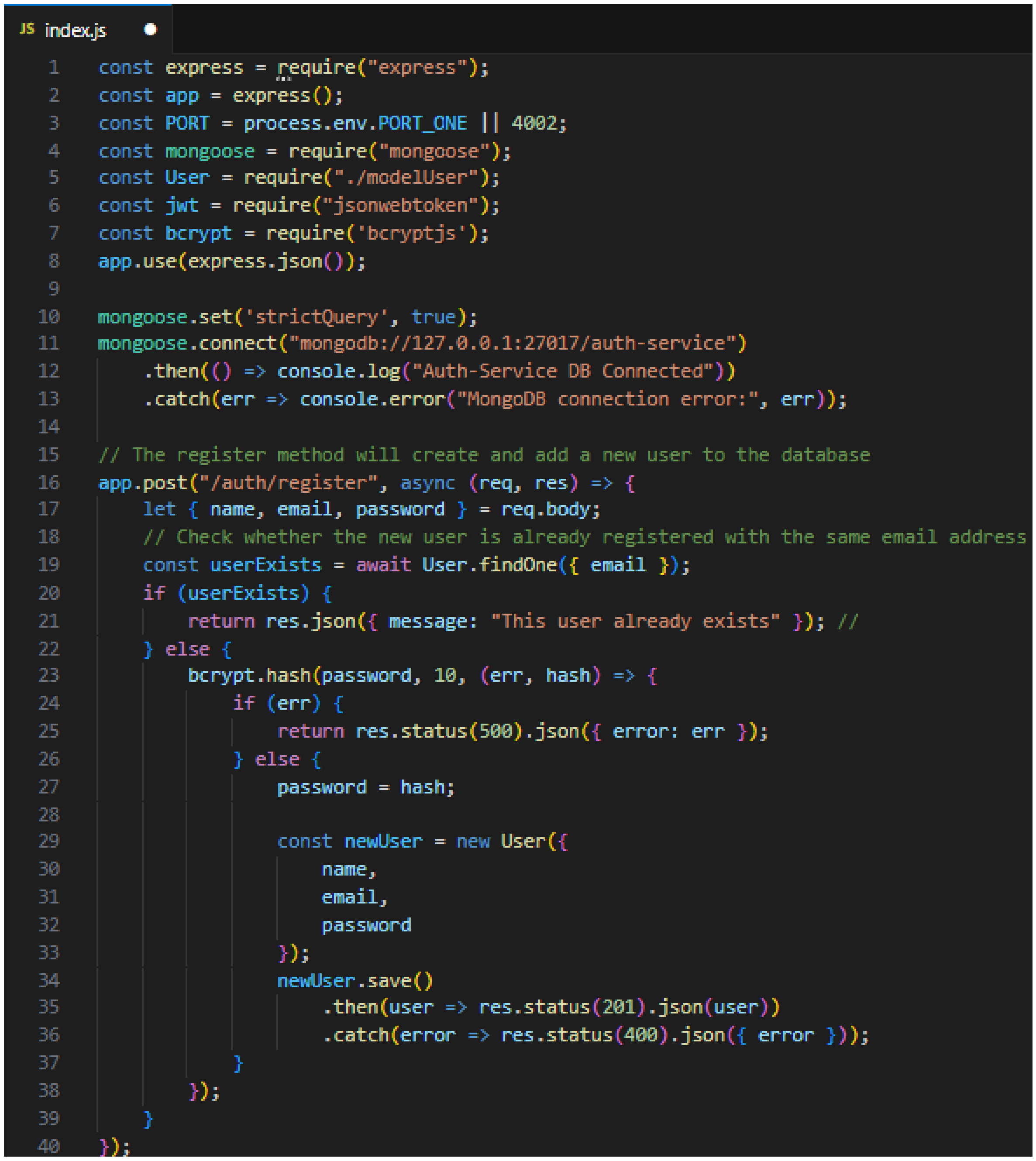
Step 3: Auth Service(auth_service/index.js)

Purpose: Handles user creation, authentication, and authorization services.

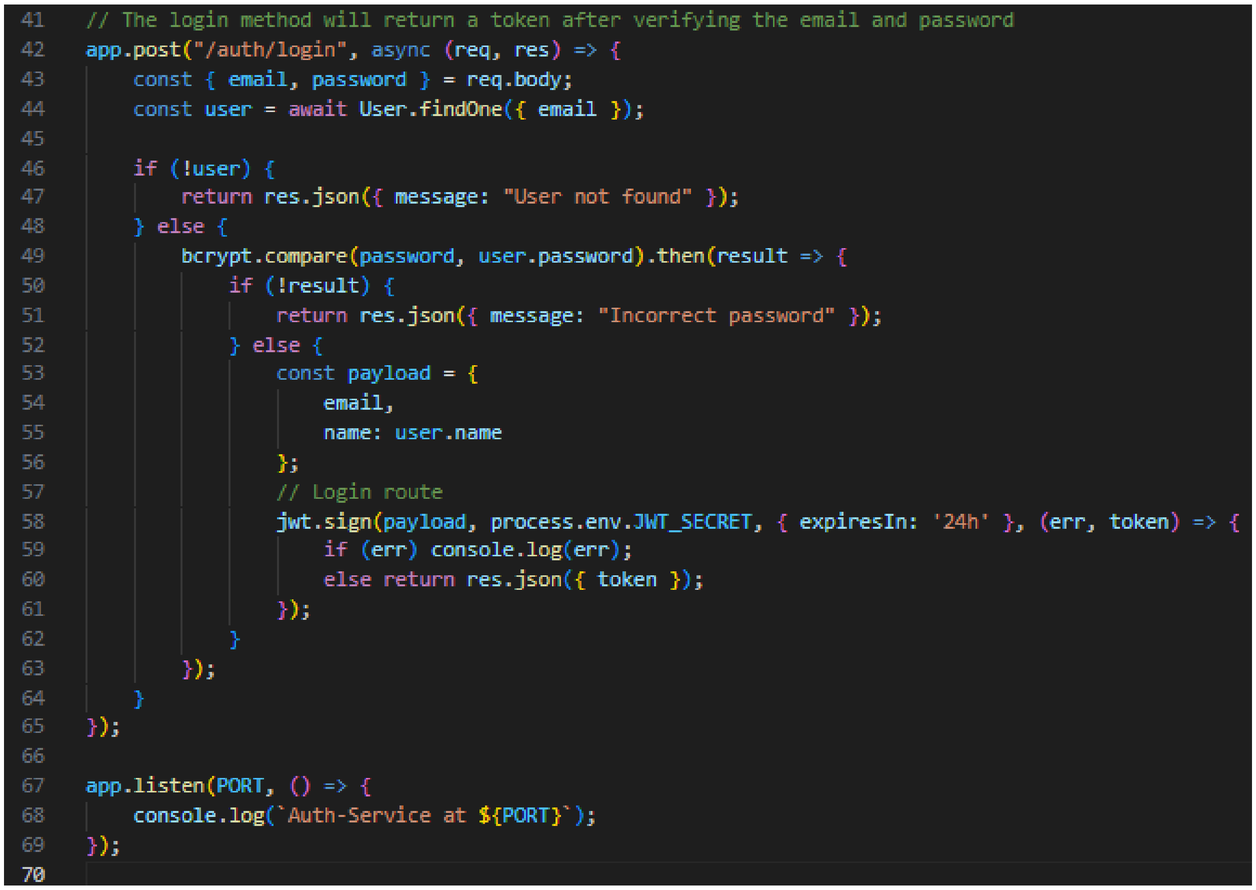
Actions Performed:

- Create *auth_service/index.js*
- Implement authentication service logic:
    ➔ Initialize all dependencies and connect to database

- ➔ Register new users
- ➔ Hash and store users passwords
- ➔ Authenticate users using credentials
- ➔ Generate JWT for authenticated users


index.js

```javascript
const express = require("express");
const app = express();
const PORT = process.env.PORT_ONE || 4002;
const mongoose = require("mongoose");
const User = require("./modelUser");
const jwt = require("jsonwebtoken");
const bcrypt = require('bcryptjs');
app.use(express.json());

mongoose.set('strictQuery', true);
mongoose.connect("mongodb://127.0.0.1:27017/auth-service")
    .then(() => console.log("Auth-Service DB Connected"))
    .catch(err => console.error("MongoDB connection error:", err));

// The register method will create and add a new user to the database
app.post("/auth/register", async (req, res) => {
    let { name, email, password } = req.body;
    // Check whether the new user is already registered with the same email address
    const userExists = await User.findOne({ email });
    if (userExists) {
        return res.json({ message: "This user already exists" }); //
    } else {
        bcrypt.hash(password, 10, (err, hash) => {
            if (err) {
                return res.status(500).json({ error: err });
            } else {
                password = hash;

                const newUser = new User({
                    name,
                    email,
                    password
                });
                newUser.save()
                    .then(user => res.status(201).json(user))
                    .catch(error => res.status(400).json({ error }));
            }
        });
    }
});
```



```
// The login method will return a token after verifying the email and password
app.post("/auth/login", async (req, res) => {
    const { email, password } = req.body;
    const user = await User.findOne({ email });

    if (!user) {
        return res.json({ message: "User not found" });
    } else {
        bcrypt.compare(password, user.password).then(result => {
            if (!result) {
                return res.json({ message: "Incorrect password" });
            } else {
                const payload = {
                    email,
                    name: user.name
                };
                // Login route
                jwt.sign(payload, process.env.JWT_SECRET, { expiresIn: '24h' }, (err, token) => {
                    if (err) console.log(err);
                    else return res.json({ token });
                });
            }
        });
    }
});

app.listen(PORT, () => {
    console.log(`Auth-Service at ${PORT}`);
});

```

Fig 5.1.2: auth_service/index.js showing db connection, user registration, login, JWT generation, and server startup at port 4002

Conclusion:
The Authentication service provides secure, stateless authentication for all other services using JWT, ensuring access control across the system.

5.2 Product-Service Implementation

Objective: Implement a standalone microservice responsible for managing product data and interacting with the database.

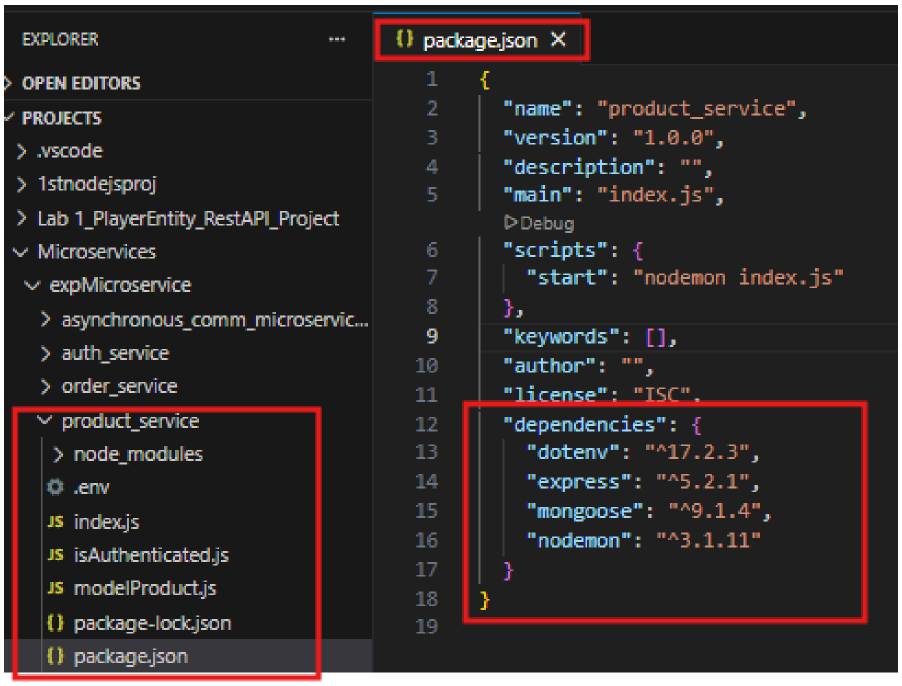
Operations Supported :

Create (POST) - Create product records for new products

- ➔ Accepts JSON input
- ➔ Validates data
- ➔ Stores it in MongoDB

Read (GET) - Retrieves product data

Step 1 - Confirm Product Service Project Directory
'Microservice/expMicroservice/product_service/' already created during the environment setup section and dependencies installed and are as below:



package.json

```
{
  "name": "product_service",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷Debug
  "scripts": {
    "start": "nodemon index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "dotenv": "^17.2.3",
    "express": "^5.2.1",
    "mongoose": "^9.1.4",
    "nodemon": "^3.1.11"
  }
}
```

Fig 5.2.1: *Microservice/expMicroservice/product_service/* directory with the dependencies needed for product_service.

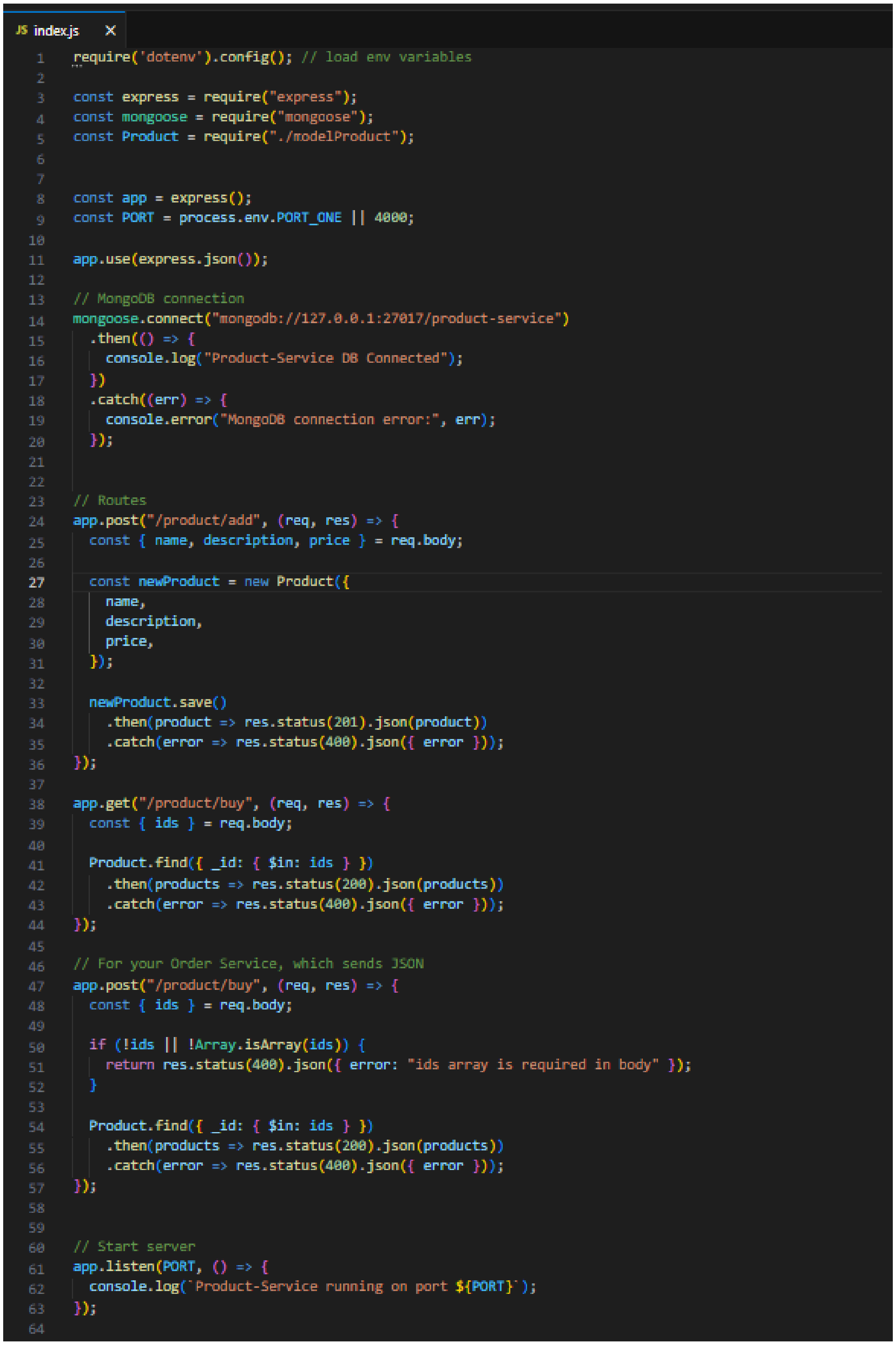
Step 2: Verify product data model modelProduct.js defined in the database creation and setup section.

Step 3: Product Service(index.js)

Purpose: Handles all produce related operations

Actions Performed:

- Create *product_service/index.js*
- Implement logic for product:
    - ➔ Initialize all dependencies and connect to database
    - ➔ Create product
    - ➔ Retrieve products by ID list
    - ➔ Start Product Service server



```
index.js
require('dotenv').config(); // load env variables

const express = require("express");
const mongoose = require("mongoose");
const Product = require("./modelProduct");


const app = express();
const PORT = process.env.PORT_ONE || 4000;

app.use(express.json());

// MongoDB connection
mongoose.connect("mongodb://127.0.0.1:27017/product-service")
  .then(() => {
    console.log("Product-Service DB Connected");
  })
  .catch((err) => {
    console.error("MongoDB connection error:", err);
  });


// Routes
app.post("/product/add", (req, res) => {
  const { name, description, price } = req.body;

  const newProduct = new Product({
    name,
    description,
    price,
  });

  newProduct.save()
    .then(product => res.status(201).json(product))
    .catch(error => res.status(400).json({ error }));
});

app.get("/product/buy", (req, res) => {
  const { ids } = req.body;

  Product.find({ _id: { $in: ids } })
    .then(products => res.status(200).json(products))
    .catch(error => res.status(400).json({ error }));
});

// For your Order Service, which sends JSON
app.post("/product/buy", (req, res) => {
  const { ids } = req.body;

  if (!ids || !Array.isArray(ids)) {
    return res.status(400).json({ error: "ids array is required in body" });
  }

  Product.find({ _id: { $in: ids } })
    .then(products => res.status(200).json(products))
    .catch(error => res.status(400).json({ error }));
});


// Start server
app.listen(PORT, () => {
  console.log(`Product-Service running on port ${PORT}`);
});
```

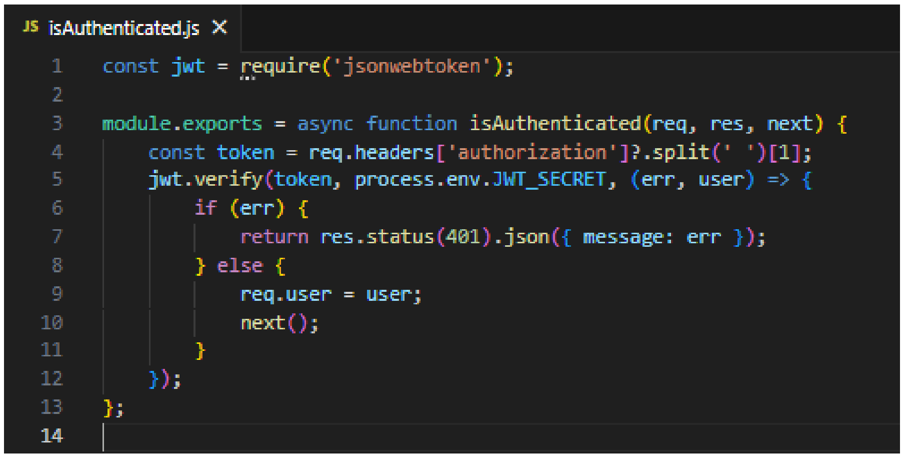
Fig 5.2.2: product_service/index.js showing Express setup, MongoDB connection, routes, and server startup at port 4000

Step 4 - Authentication Middleware (isAuthenticated.js)

Purpose: Handles authentication for protected routes

Actions Performed:

- Create *product_service/isAuthenticated.js*
- Implement JWT verification logic:
    - ➔ Read token from Authorization header
    - ➔ Verify token using secret key



isAuthenticated.js

```
const jwt = require('jsonwebtoken');

module.exports = async function isAuthenticated(req, res, next) {
    const token = req.headers['authorization']?.split(' ')[1];
    jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
        if (err) {
            return res.status(401).json({ message: err });
        } else {
            req.user = user;
            next();
        }
    });
};
```

Fig 5.2.3: product_service/isAuthenticated.js showing JWT verification middleware

Conclusion:
The product-service is implemented as an independent microservice with a MongoDB backed data model. Here, the implementation operates as an HTTP service for synchronous communication.

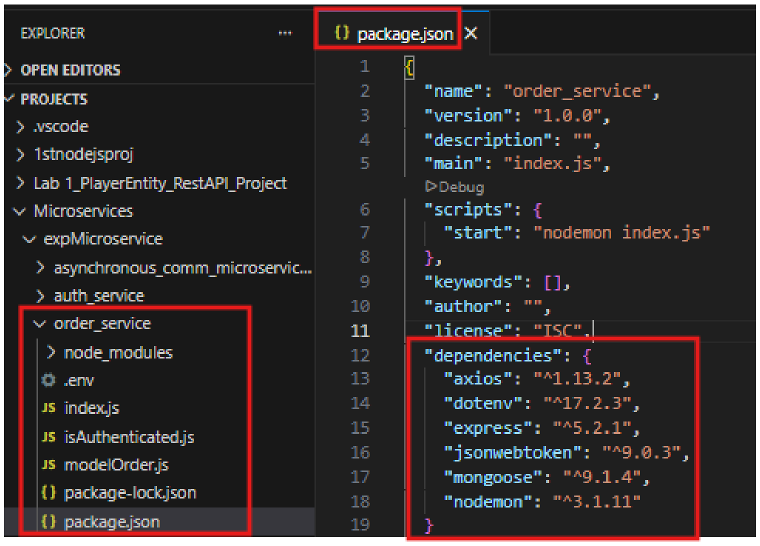
5.3 Order-Service Implementation

Objective: Implement a microservice responsible for order creation and communicate with product-service.

Operations Supported :

- **Create (POST)** - Create new orders

Step 1 - Confirm Order Service Project Directory
Ensure '*Microservice/expMicroservice/order_service/*' exists and dependencies installed and are as below:



```json
{
  "name": "order_service",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "axios": "^1.13.2",
    "dotenv": "^17.2.3",
    "express": "^5.2.1",
    "jsonwebtoken": "^9.0.3",
    "mongoose": "^9.1.4",
    "nodemon": "^3.1.11"
  }
```

Fig 5.3.1: *Microservice/expMicroservice/order_service/* directory with the dependencies needed for order_service.

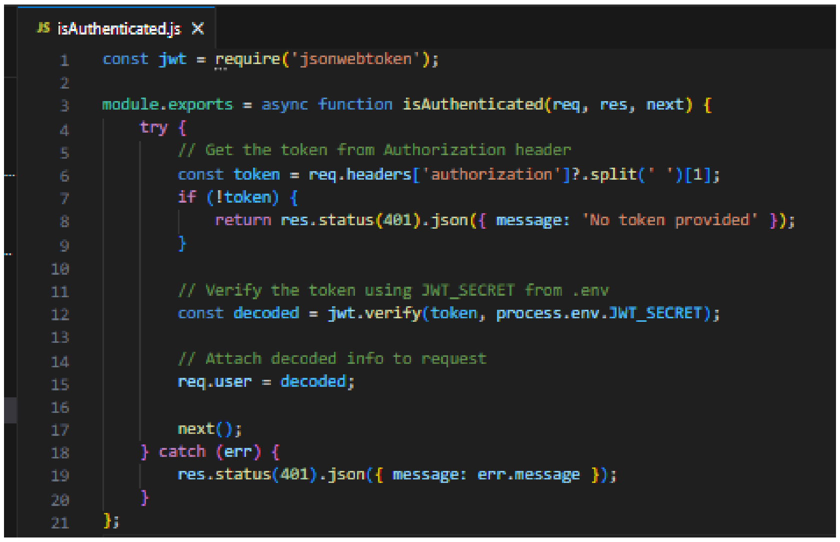
Step 2: Verify order data model modelOrder.js defined as part of the database setup.

Step 3 - Authentication Middleware (isAuthenticated.js)

Purpose: Handles authentication enforcement for protected routes

Actions Performed:

- Create *order_service/isAuthenticated.js*
- Implement JWT verification logic:
  - ➔ Read token from Authorization header
  - ➔ Verify token using secret key



```javascript
const jwt = require('jsonwebtoken');

module.exports = async function isAuthenticated(req, res, next) {
    try {
        // Get the token from Authorization header
        const token = req.headers['authorization']?.split(' ')[1];
        if (!token) {
            return res.status(401).json({ message: 'No token provided' });
        }

        // Verify the token using JWT_SECRET from .env
        const decoded = jwt.verify(token, process.env.JWT_SECRET);

        // Attach decoded info to request
        req.user = decoded;

        next();
    } catch (err) {
        res.status(401).json({ message: err.message });
    }
};
```

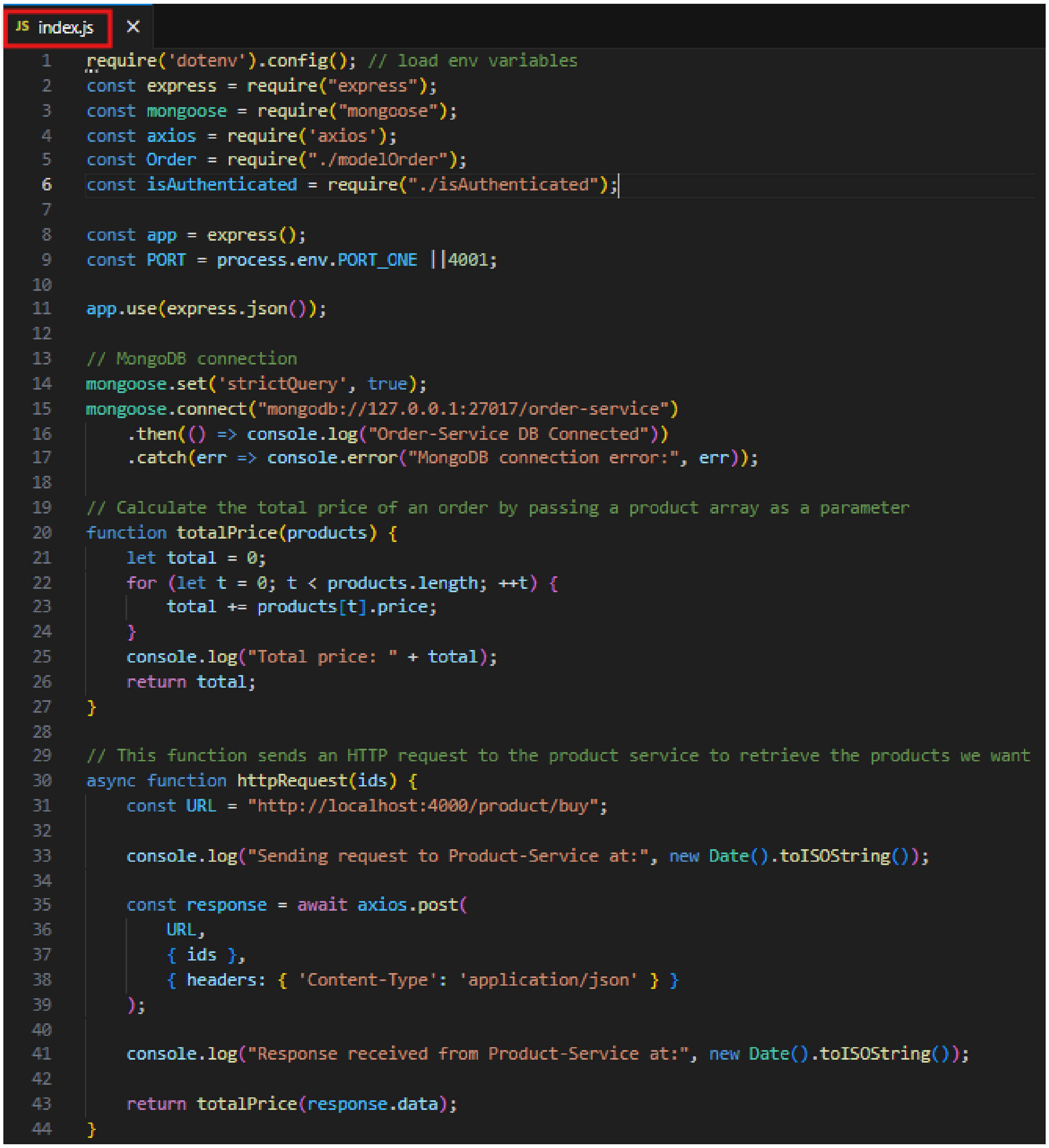
Fig 5.3.2: order_service/isAuthenticated.js showing JWT verification middleware

Step 4: Order Service(order_service/index.js)

Purpose: Handles all order related operations

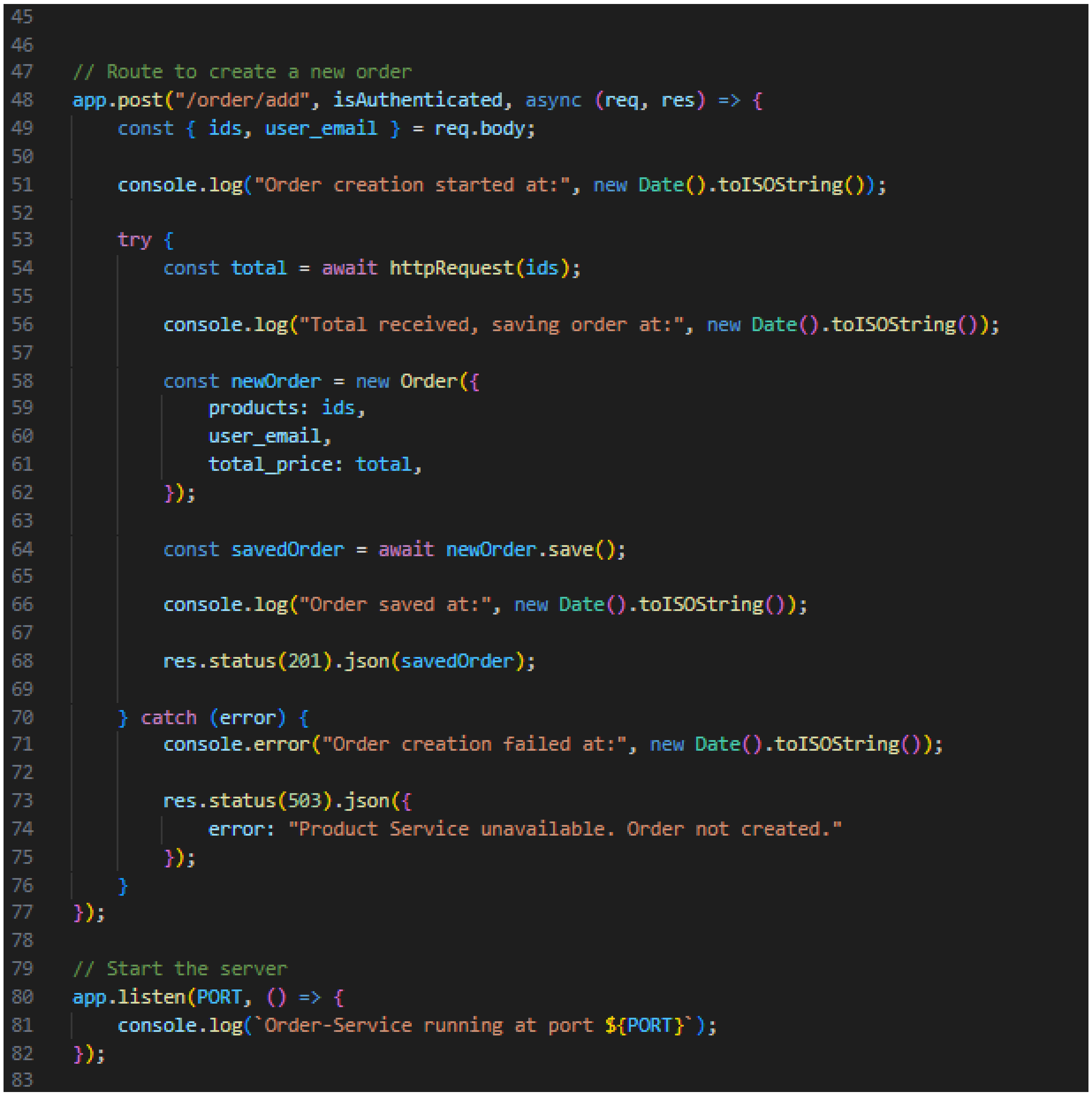
Actions Performed:

- Create *order_service/index.js*
- Implement order service logic:
    - ➔ Request product data from Product Service using axios
    - ➔ Calculate total order price
    - ➔ Create new order
    - ➔ Protect order creation using authentication middleware
    - ➔ Start Order Service server



index.js

```javascript
require('dotenv').config(); // load env variables
const express = require("express");
const mongoose = require("mongoose");
const axios = require('axios');
const Order = require("./modelOrder");
const isAuthenticated = require("./isAuthenticated");

const app = express();
const PORT = process.env.PORT_ONE ||4001;

app.use(express.json());

// MongoDB connection
mongoose.set('strictQuery', true);
mongoose.connect("mongodb://127.0.0.1:27017/order-service")
    .then(() => console.log("Order-Service DB Connected"))
    .catch(err => console.error("MongoDB connection error:", err));

// Calculate the total price of an order by passing a product array as a parameter
function totalPrice(products) {
    let total = 0;
    for (let t = 0; t < products.length; ++t) {
        total += products[t].price;
    }
    console.log("Total price: " + total);
    return total;
}

// This function sends an HTTP request to the product service to retrieve the products we want
async function httpRequest(ids) {
    const URL = "http://localhost:4000/product/buy";

    console.log("Sending request to Product-Service at:", new Date().toISOString());

    const response = await axios.post(
        URL,
        { ids },
        { headers: { 'Content-Type': 'application/json' } }
    );

    console.log("Response received from Product-Service at:", new Date().toISOString());

    return totalPrice(response.data);
}
```



```javascript


// Route to create a new order
app.post("/order/add", isAuthenticated, async (req, res) => {
    const { ids, user_email } = req.body;

    console.log("Order creation started at:", new Date().toISOString());

    try {
        const total = await httpRequest(ids);

        console.log("Total received, saving order at:", new Date().toISOString());

        const newOrder = new Order({
            products: ids,
            user_email,
            total_price: total,
        });

        const savedOrder = await newOrder.save();

        console.log("Order saved at:", new Date().toISOString());

        res.status(201).json(savedOrder);

    } catch (error) {
        console.error("Order creation failed at:", new Date().toISOString());

        res.status(503).json({
            error: "Product Service unavailable. Order not created."
        });
    }
});

// Start the server
app.listen(PORT, () => {
    console.log(`Order-Service running at port ${PORT}`);
});

```

Fig 5.3.3: order_service/index.js showing db connection, order creation route, inter-service communication, and server startup at port 4001

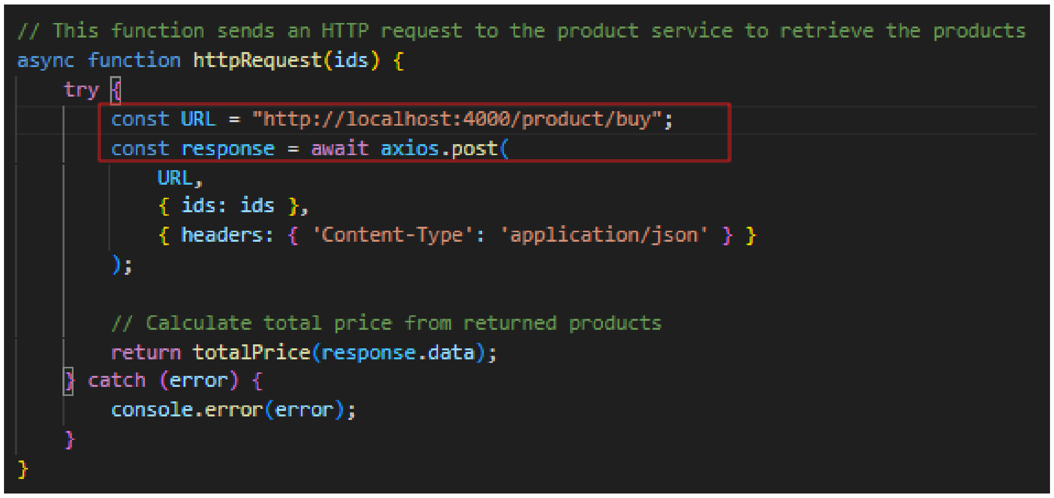
Synchronous Communication (HTTP) FLOW

Objective

Demonstrate blocking, request-response communication between Order-Service and Product-Service using REST over HTTP.

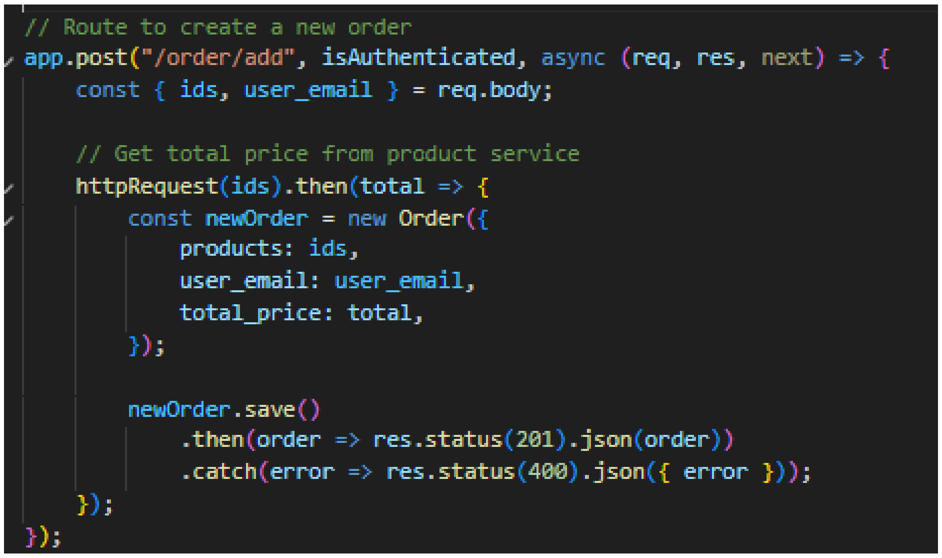
Operations:

- ➔ Order validation / Product availability check
- ➔ Order creation dependent on Product-Service



```
// This function sends an HTTP request to the product service to retrieve the products
async function httpRequest(ids) {
    try {
        const URL = "http://localhost:4000/product/buy";
        const response = await axios.post(
            URL,
            { ids: ids },
            { headers: { 'Content-Type': 'application/json' } }
        );

        // Calculate total price from returned products
        return totalPrice(response.data);
    } catch (error) {
        console.error(error);
    }
}
```

Fig 5.3.4: httpRequest(ids) Helper Function sends a POST request to Product-Service via Axios. Execution blocks here until product data is returned.



```
// Route to create a new order
app.post("/order/add", isAuthenticated, async (req, res, next) => {
    const { ids, user_email } = req.body;

    // Get total price from product service
    httpRequest(ids).then(total => {
        const newOrder = new Order({
            products: ids,
            user_email: user_email,
            total_price: total,
        });

        newOrder.save()
            .then(order => res.status(201).json(order))
            .catch(error => res.status(400).json({ error }));
    });
});
```

Fig 5.3.5: POST /order/add route calls httpRequest(ids) demonstrates synchronous enforcement at route level since order is only saved after helper returns.

Conclusion:

Order_Service orchestrators business logic. It waits for product_service to respond before it can create an order. This confirms synchronous communication which introduces runtime dependency on product_service.

6.0 SERVICES IMPLEMENTATION (ASYNCHRONOUS COMMUNICATION MODE)

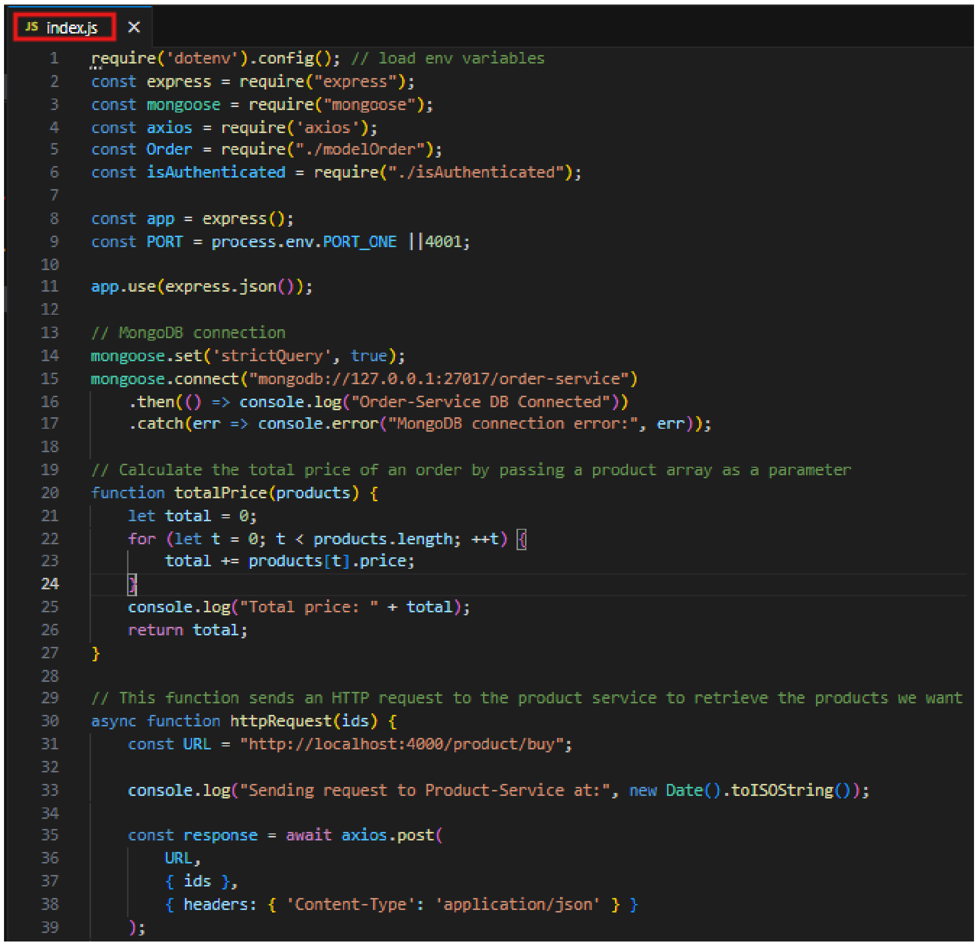
Objective: Demonstrate non-blocking communication between Order-Service and Product-Service using RabbitMQ, the service does not wait for immediate response.

Step 1: Order Service (Asynchronous) (asynchronous_comm_ms_archit/order_service/index.js)

Purpose: Handles order creation and publishes product-related events for asynchronous processing.

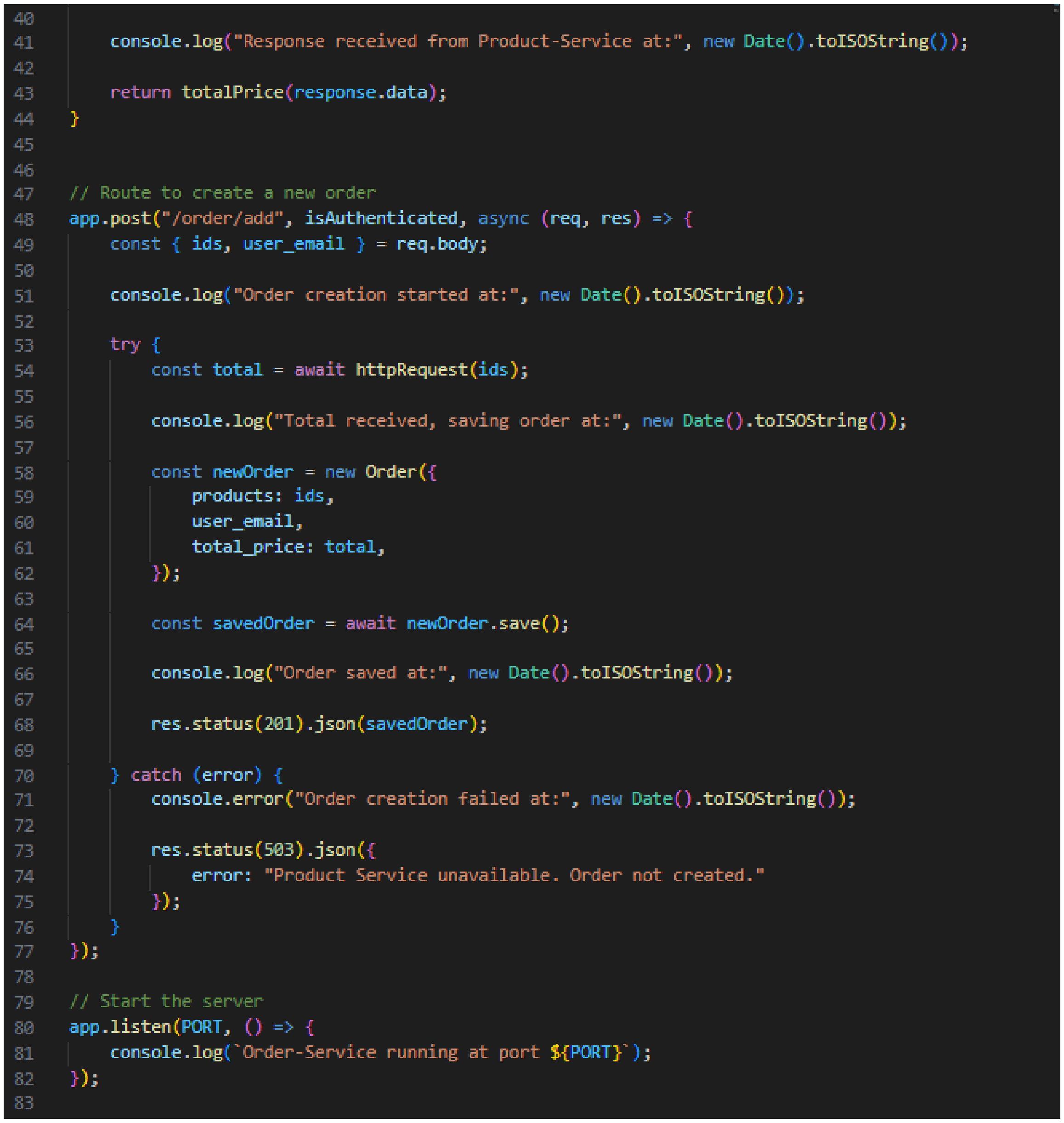
Actions Performed:

- Create **asynchronous_comm_ms_archit/order_service/index.js**
- Implement order service logic to:
    - ➔ Initialize all dependencies and connect to database
    - ➔ Establish RabbitMQ connection
    - ➔ Create new order
    - ➔ Call to publish productIDs to RabbitMQ
    - ➔ Persist order locally with initial state



index.js

```javascript
require('dotenv').config(); // load env variables
const express = require("express");
const mongoose = require("mongoose");
const axios = require('axios');
const Order = require("./modelOrder");
const isAuthenticated = require("./isAuthenticated");

const app = express();
const PORT = process.env.PORT_ONE ||4001;

app.use(express.json());

// MongoDB connection
mongoose.set('strictQuery', true);
mongoose.connect("mongodb://127.0.0.1:27017/order-service")
    .then(() => console.log("Order-Service DB Connected"))
    .catch(err => console.error("MongoDB connection error:", err));

// Calculate the total price of an order by passing a product array as a parameter
function totalPrice(products) {
    let total = 0;
    for (let t = 0; t < products.length; ++t) {
        total += products[t].price;
    }
    console.log("Total price: " + total);
    return total;
}

// This function sends an HTTP request to the product service to retrieve the products we want
async function httpRequest(ids) {
    const URL = "http://localhost:4000/product/buy";

    console.log("Sending request to Product-Service at:", new Date().toISOString());

    const response = await axios.post(
        URL,
        { ids },
        { headers: { 'Content-Type': 'application/json' } }
    );
```



```

    console.log("Response received from Product-Service at:", new Date().toISOString());

    return totalPrice(response.data);
}


// Route to create a new order
app.post("/order/add", isAuthenticated, async (req, res) => {
    const { ids, user_email } = req.body;

    console.log("Order creation started at:", new Date().toISOString());

    try {
        const total = await httpRequest(ids);

        console.log("Total received, saving order at:", new Date().toISOString());

        const newOrder = new Order({
            products: ids,
            user_email,
            total_price: total,
        });

        const savedOrder = await newOrder.save();

        console.log("Order saved at:", new Date().toISOString());

        res.status(201).json(savedOrder);

    } catch (error) {
        console.error("Order creation failed at:", new Date().toISOString());

        res.status(503).json({
            error: "Product Service unavailable. Order not created."
        });
    }
});

// Start the server
app.listen(PORT, () => {
    console.log(`Order-Service running at port ${PORT}`);
});

```

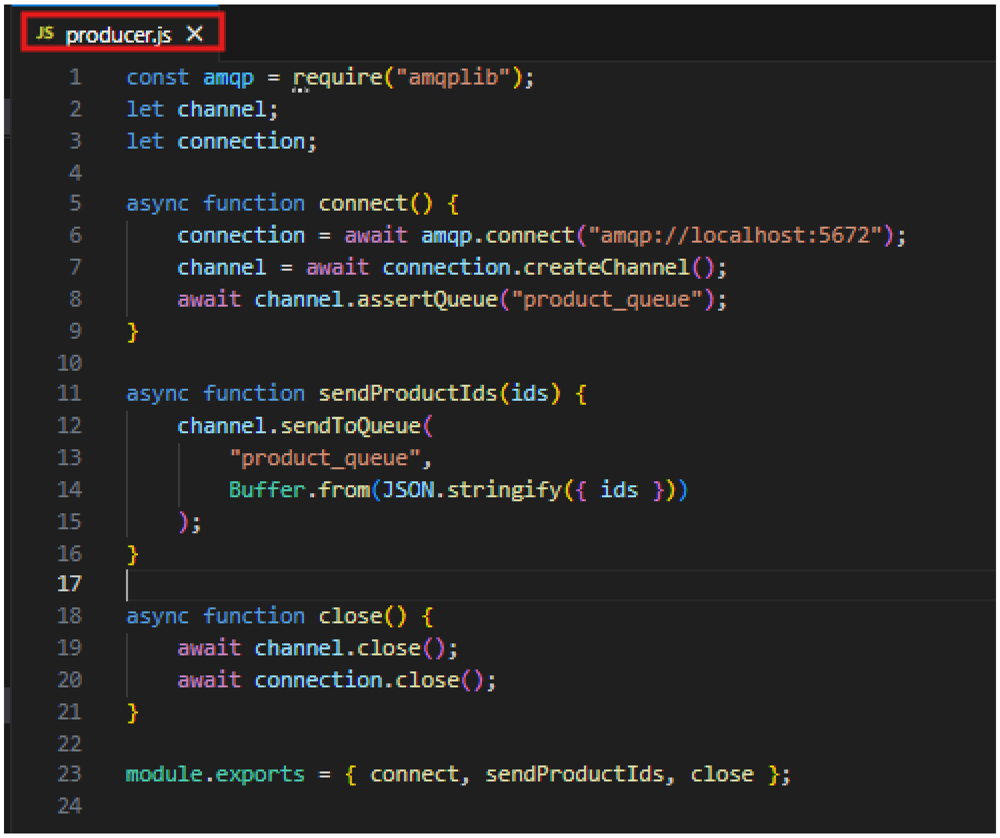
Fig 6.1.1: order_service/index.js containing DB and RabbitMQ connection, /order/add endpoint

Step 2: RabbitMQ Producer under Order_Service(producer.js)

Purpose: Handles RabbitMQ connection and published product IDs to product queue.

Actions Performed:

- Create **asynchronous_comm_ms_archit/order_service/producer.js**
- **Implement logic:**
  - ➔ Connect to RabbitMQ server at amqp://localhost:15672
  - ➔ Create a channel and assert product_queue to send messages
  - ➔ Publish product IDs to product_queue
  - ➔ Close channel and connection



```javascript
const amqp = require("amqplib");
let channel;
let connection;

async function connect() {
    connection = await amqp.connect("amqp://localhost:5672");
    channel = await connection.createChannel();
    await channel.assertQueue("product_queue");
}

async function sendProductIds(ids) {
    channel.sendToQueue(
        "product_queue",
        Buffer.from(JSON.stringify({ ids }))
    );
}

async function close() {
    await channel.close();
    await connection.close();
}

module.exports = { connect, sendProductIds, close };
```

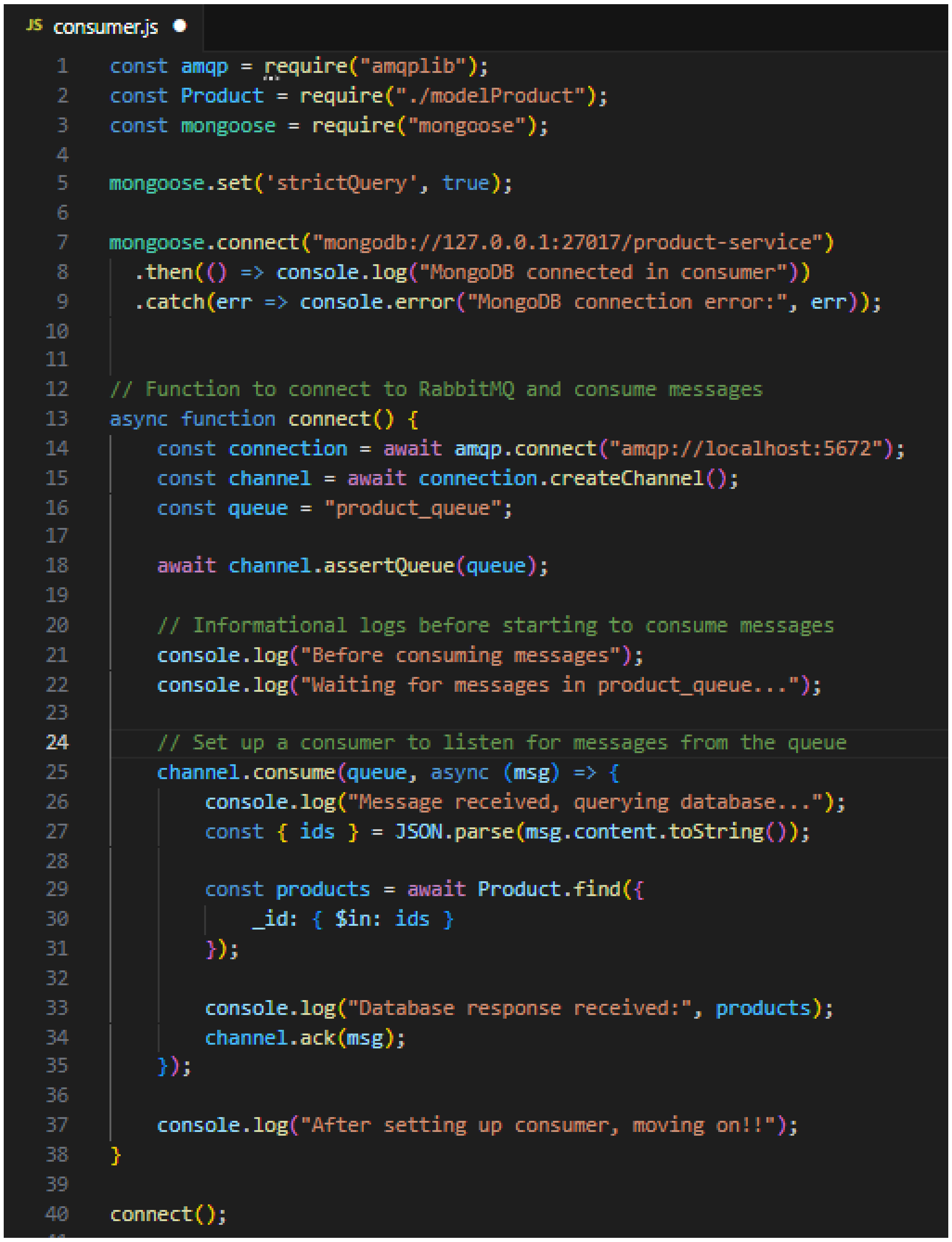
Fig 6.1.2: order_service/producer.js to establish connection to RabbitMQ, assert queue, publish productID

Step 3: RabbitMQ Consumer under Product_Service (consumer.js)

Purpose: Handles consuming productIDs from RabbitMQ and querying database for product details.

Actions Performed:

- Create **asynchronous_comm_ms_archit/product_service/consumer.js**
- **Implement logic:**
    - ➔ Connect to RabbitMQ server at amqp://localhost:15672
    - ➔ Create a channel and assert product_queue to receive message
    - ➔ Consume messages from queue
    - ➔ Query database for products matching IDs received



```javascript
const amqp = require("amqplib");
const Product = require("./modelProduct");
const mongoose = require("mongoose");

mongoose.set('strictQuery', true);

mongoose.connect("mongodb://127.0.0.1:27017/product-service")
  .then(() => console.log("MongoDB connected in consumer"))
  .catch(err => console.error("MongoDB connection error:", err));


// Function to connect to RabbitMQ and consume messages
async function connect() {
    const connection = await amqp.connect("amqp://localhost:5672");
    const channel = await connection.createChannel();
    const queue = "product_queue";

    await channel.assertQueue(queue);

    // Informational logs before starting to consume messages
    console.log("Before consuming messages");
    console.log("Waiting for messages in product_queue...");

    // Set up a consumer to listen for messages from the queue
    channel.consume(queue, async (msg) => {
        console.log("Message received, querying database...");
        const { ids } = JSON.parse(msg.content.toString());

        const products = await Product.find({
            _id: { $in: ids }
        });

        console.log("Database response received:", products);
        channel.ack(msg);
    });

    console.log("After setting up consumer, moving on!!");
}

connect();
```

Fig 6.1.3: product_service/consumer.js showing message consumption and database query

Conclusion

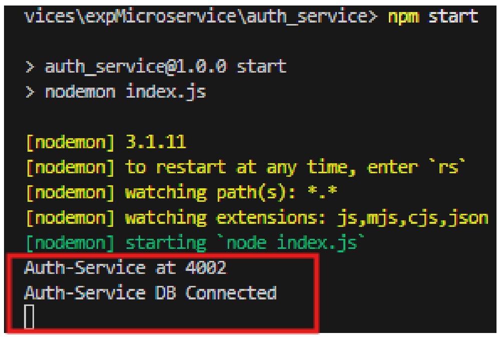
The asynchronous communication mode extends the existing implementation by use of RabbitMQ for messaging. Order creation no longer waits for product processing to complete, rather it publishes productIDs to a message queue, and Product-Service processes them independently. This enables non-blocking interaction.

7.0 TESTING AND VERIFICATION USING POSTMAN

7.1 Step by Step Testing of All Services (Synchronous)

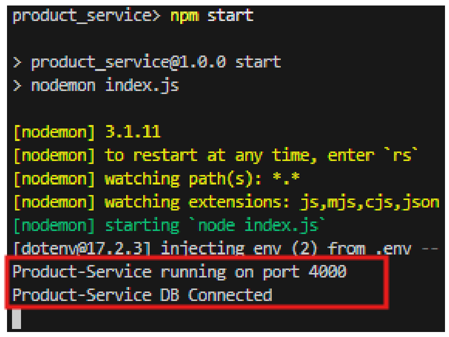
7.1.1 Precondition

- Node.js servers for each of the services are running
- MongoDB service is running
- Collections exists



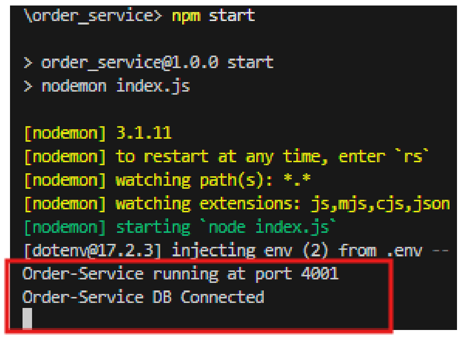
```
vices\expMicroservice\auth_service> npm start

> auth_service@1.0.0 start
> nodemon index.js

[nodemon] 3.1.11
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
Auth-Service at 4002
Auth-Service DB Connected
```

Auth-Services running



```
product_service> npm start

> product_service@1.0.0 start
> nodemon index.js

[nodemon] 3.1.11
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
[dotenv@17.2.3] injecting env (2) from .env --
Product-Service running on port 4000
Product-Service DB Connected
```

Product-Services running



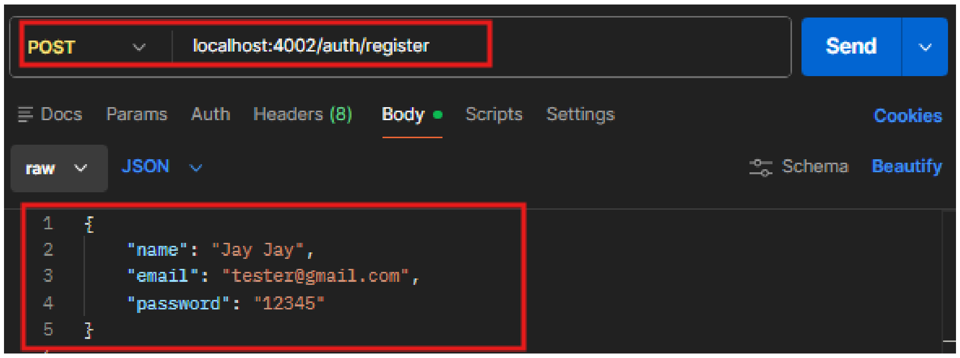
```
\order_service> npm start

> order_service@1.0.0 start
> nodemon index.js

[nodemon] 3.1.11
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
[dotenv@17.2.3] injecting env (2) from .env --
Order-Service running at port 4001
Order-Service DB Connected
```

Order-Services running

Fig 7.0: Running services

Step 1: User Register Test (auth_service)

➔ Send POST requests to /auth/register
➔ Provide name, email, password
➔ **INPUT**:

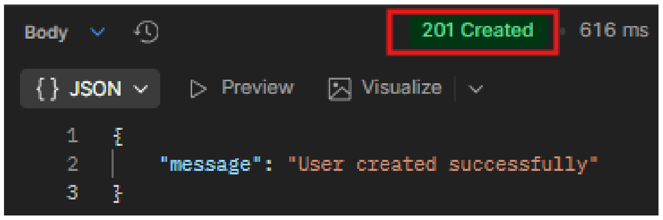


POST localhost:4002/auth/register

```
{
    "name": "Jay Jay",
    "email": "tester@gmail.com",
    "password": "12345"
}
```

Fig 7.1.1a: Register Request

➔ **OUTPUT:**

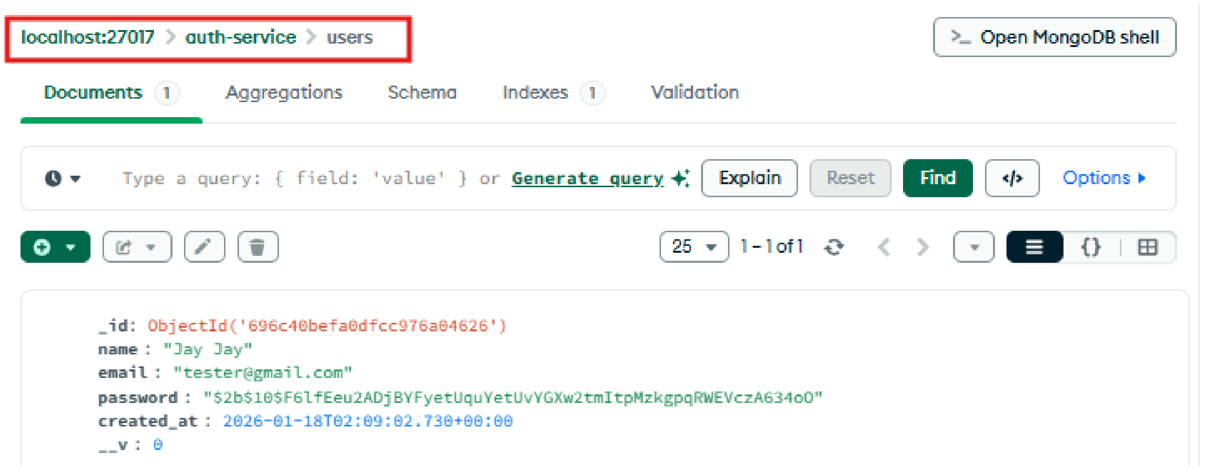


201 Created 616 ms

```
{
    "message": "User created successfully"
}
```

Fig 7.1.1b: Output of Register Request (Successful user creation)

➔ **Database Output**



```
_id: ObjectId('696c40befa0dfcc976a04626')
name : "Jay Jay"
email : "tester@gmail.com"
password : "$2b$10$F6lfEeu2ADjBYFyetUquYetUvYGXw2tmItpMzkgpqRWEVczA634oO"
created_at : 2026-01-18T02:09:02.730+00:00
__v : 0
```

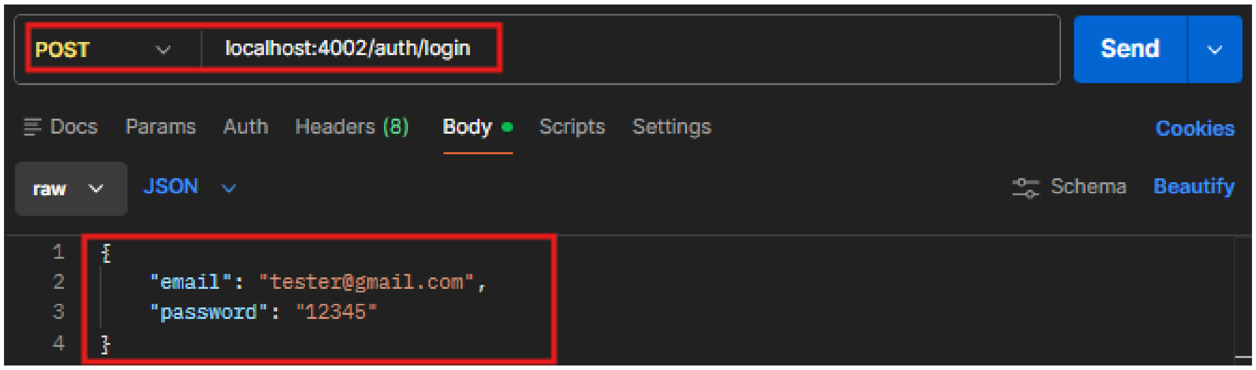
Fig 7.1.1c: Created user as saved in the auth_service database

Step 2: User Login Test (auth_service)

➔ Send POST requests to /auth/login

➔ Receive JWT token

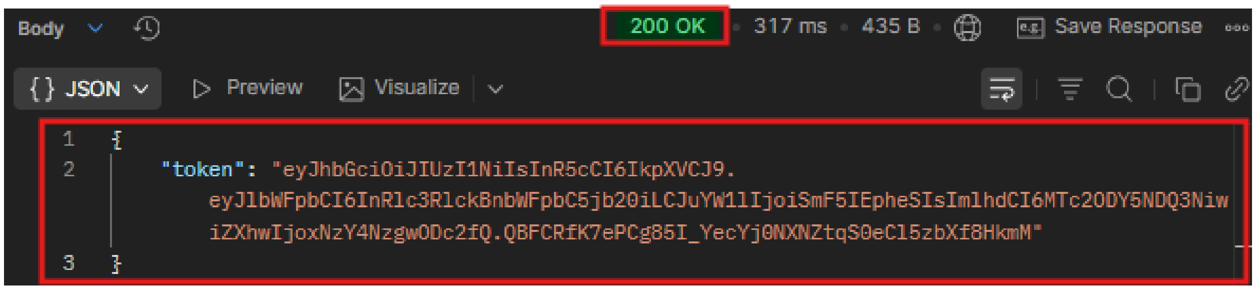
➔ **INPUT**:



```
{
    "email": "tester@gmail.com",
    "password": "12345"
}
```

Fig 7.1.2a: Login request using created user credentials

➔ **OUTPUT:**



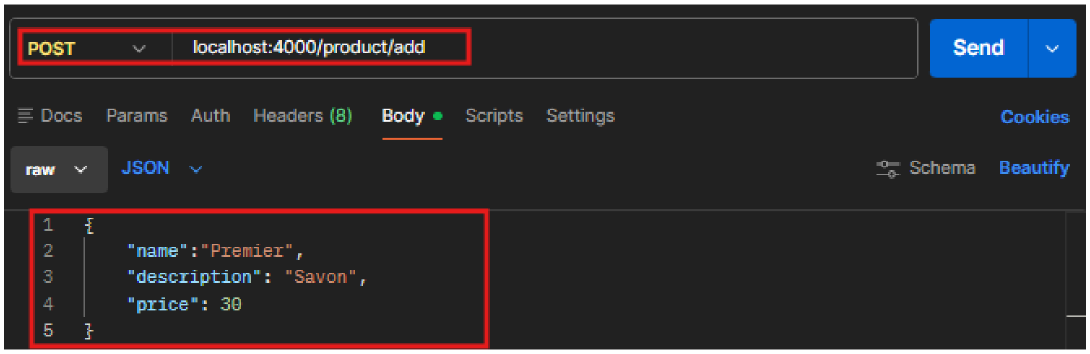
```
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJlbWFpbCI6InRlc3RlckBnbWFpbC5jb20iLCJuYW1lIjoiSmF5IEpheSIsImlhdCI6MTc2ODY5NDQ3Niw
        iZXhwIjoxNzY4NzgwODc2fQ.QBFCRfK7ePCg85I_YecYj0NXNZtqS0eCl5zbXf8HkmM"
}
```

Fig 7.1.2b: Login response containing JWT

Step 3: Add Products (product_service)

➔ Call POST /product/add

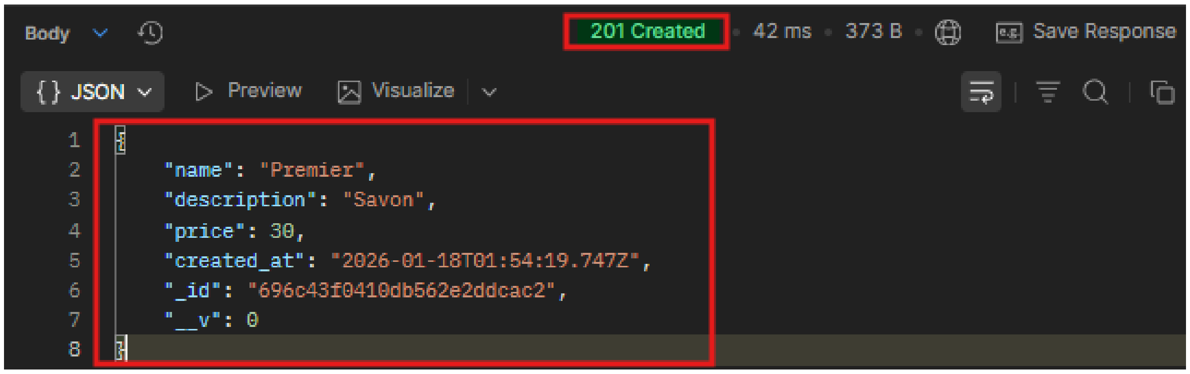
➔ **INPUT**:



```
{
    "name":"Premier",
    "description": "Savon",
    "price": 30
}
```

Fig 7.1.3a: Add product to product_service

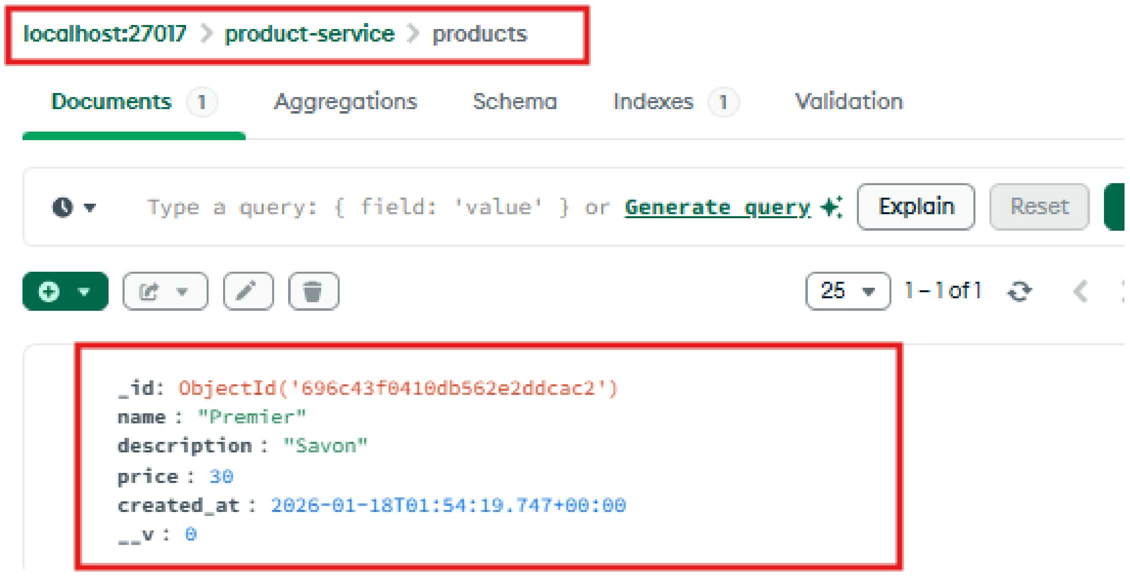
➔ **OUTPUT**:



```
{
    "name": "Premier",
    "description": "Savon",
    "price": 30,
    "created_at": "2026-01-18T01:54:19.747Z",
    "_id": "696c43f0410db562e2ddcac2",
    "__v": 0
}
```

Fig 7.1.3b: Successful add product request

➔ **Database Output:**


localhost:27017 > product-service > products
Documents 1
Aggregations
Schema
Indexes 1
Validation
Type a query: { field: 'value' } or Generate query
Explain
Reset
25
1 – 1 of 1

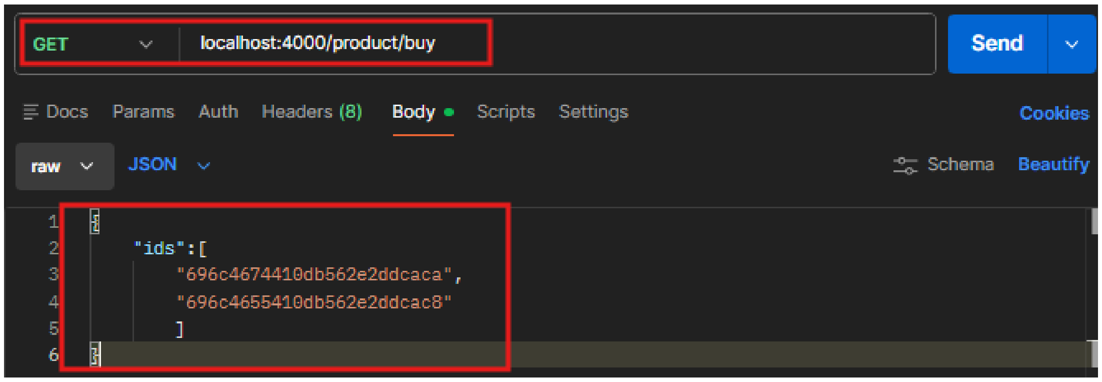
```
_id: ObjectId('696c43f0410db562e2ddcac2')
name : "Premier"
description : "Savon"
price : 30
created_at : 2026-01-18T01:54:19.747+00:00
__v : 0
```

Fig 7.1.3c: Added Product on product_service database

Step 4: Buy Products using product ID (product_service)

➔ Call GET/product/buy

➔ **INPUT**:

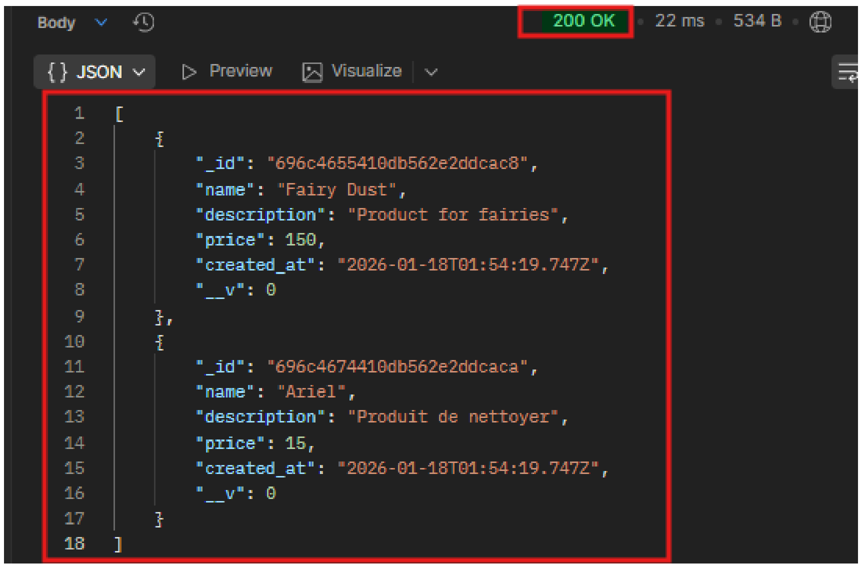


localhost:4000/product/buy

```
{
    "ids":[
        "696c4674410db562e2ddcaca",
        "696c4655410db562e2ddcac8"
        ]
}
```

Fig 7.1.4a: Buy product request

➔ **OUTPUT**:



```
[
    {
        "_id": "696c4655410db562e2ddcac8",
        "name": "Fairy Dust",
        "description": "Product for fairies",
        "price": 150,
        "created_at": "2026-01-18T01:54:19.747Z",
        "__v": 0
    },
    {
        "_id": "696c4674410db562e2ddcaca",
        "name": "Ariel",
        "description": "Produit de nettoyer",
        "price": 15,
        "created_at": "2026-01-18T01:54:19.747Z",
        "__v": 0
    }
]
```

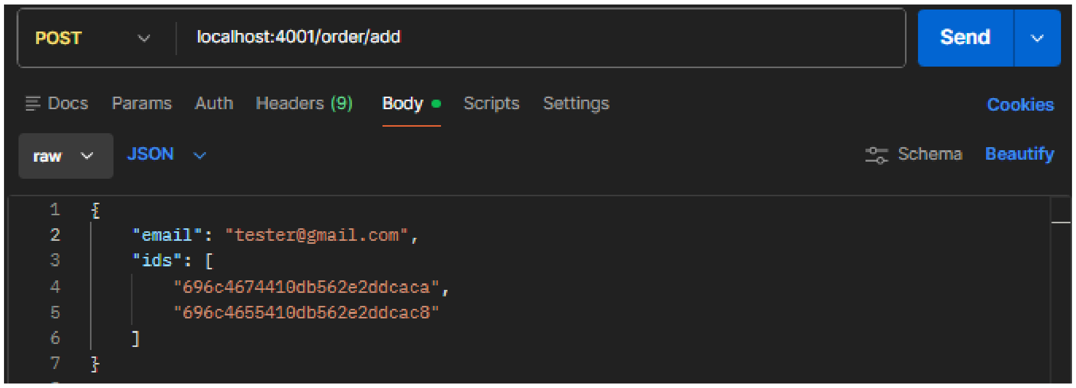
Fig 7.1.4b: Successful buy product request

Step 5: Create a new order (order_service)

➔ Call POST http://localhost:4001/order/add without Authorization header

➔ **INPUT:**

Create an order through a protected route without an authorization header.

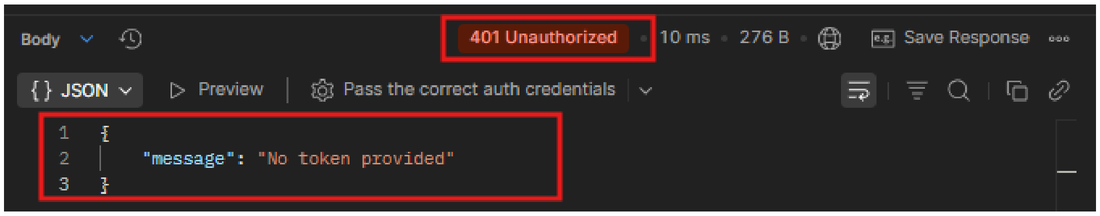


localhost:4001/order/add

```
{
    "email": "tester@gmail.com",
    "ids": [
        "696c4674410db562e2ddcaca",
        "696c4655410db562e2ddcac8"
    ]
}
```

Fig 7.1.5a: Create order without Authorization header

➔ **OUTPUT**:



```
{
    "message": "No token provided"
}
```

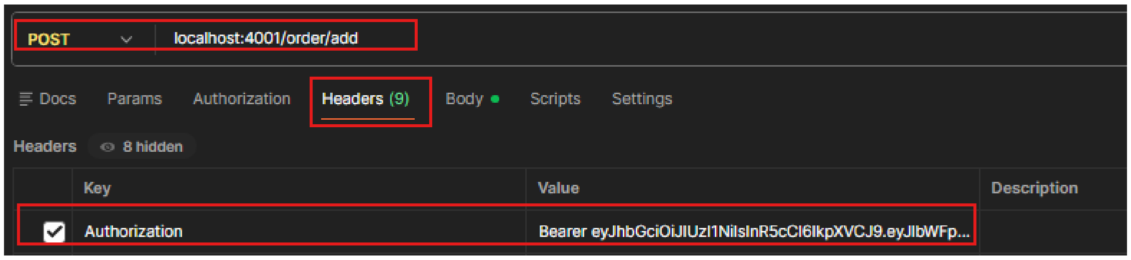
Fig 7.1.5b: 401 Unauthorized response

Step 6: Create a new order (order_service)

➔ Add header: Authorization: Bearer <token>

➔ Call POST http://localhost:4001/order/add with Authorization header

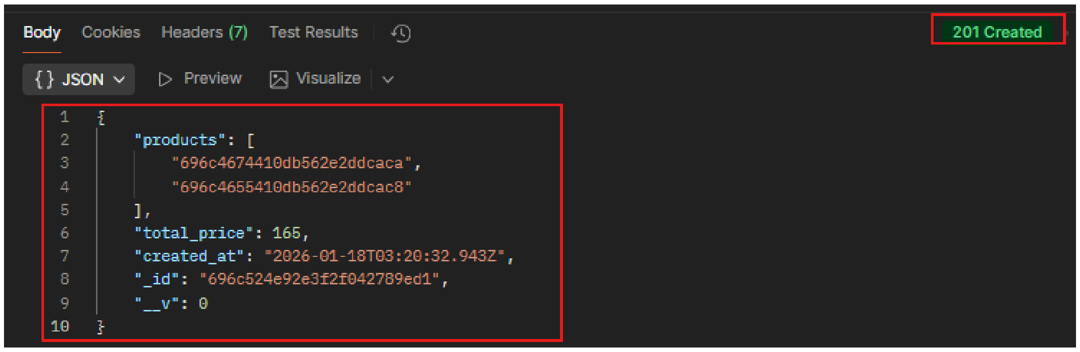
➔ **INPUT**:



Key	Value
Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFp...

Fig 7.1.6a: Create order with Authorization header

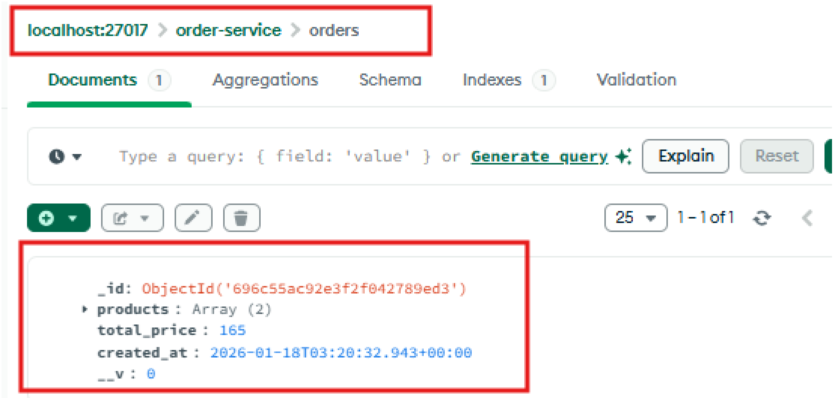
➔ **OUTPUT**:



```
{
    "products": [
        "696c4674410db562e2ddcaca",
        "696c4655410db562e2ddcac8"
    ],
    "total_price": 165,
    "created_at": "2026-01-18T03:20:32.943Z",
    "_id": "696c524e92e3f2f042789ed1",
    "__v": 0
}
```

Fig 7.1.6b: Successful protected create order request

➔ **Database Output:**


localhost:27017 > order-service > orders
Documents 1
Aggregations
Schema
Indexes 1
Validation
Type a query: { field: 'value' } or Generate query
Explain
Reset
25
1 – 1 of 1

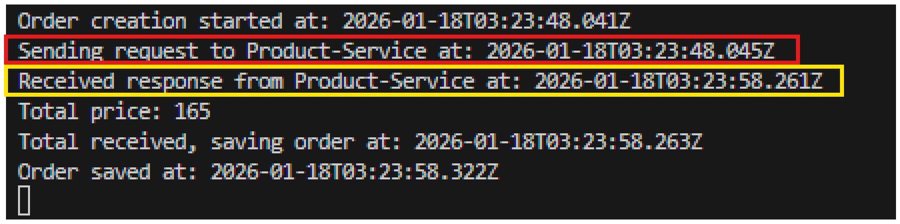
```
_id: ObjectId('696c55ac92e3f2f042789ed3')
products : Array (2)
total_price : 165
created_at : 2026-01-18T03:20:32.943+00:00
__v : 0
```

Fig 7.1.6c: Created order on the order_service database

Step 7: Proof of Synchronous Behavior via Simulated Delay

➔ Check Order-Service console logs for the effect of simulated delay between sending and receiving response from product_service on order creation.

➔ **OUTPUT**:



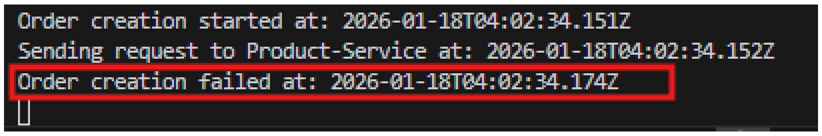
```
Order creation started at: 2026-01-18T03:23:48.041Z
Sending request to Product-Service at: 2026-01-18T03:23:48.045Z
Received response from Product-Service at: 2026-01-18T03:23:58.261Z
Total price: 165
Total received, saving order at: 2026-01-18T03:23:58.263Z
Order saved at: 2026-01-18T03:23:58.322Z
```

Fig 7.1.7a: The POST /order/add route waited for the product_service response, showing synchronous blocking behavior.

Step 8: Proof of Synchronous Behavior via Unavailable Product Service

➔ Check Order-Service console logs for the effect of unavailable product_service.

OUTPUT:



```
Order creation started at: 2026-01-18T04:02:34.151Z
Sending request to Product-Service at: 2026-01-18T04:02:34.152Z
Order creation failed at: 2026-01-18T04:02:34.174Z
```

Fig 7.1.8a: Order_service stops execution and fails when product_service did not respond.

7.2 Step by Step Testing of Asynchronous Setup

Step 1: Precondition

➔ All services running

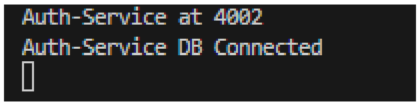
➔ MongoDB active

➔ RabbitMQ server running and management UI open

➔ Start services independently (auth_service/index.js, product_service/index.js, order_service/index.js, product_service/consumer.js),

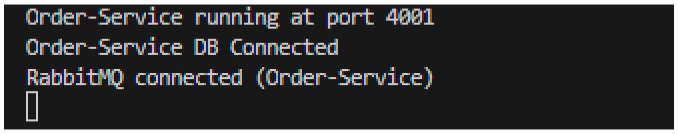
:

➔ **OUTPUT**:



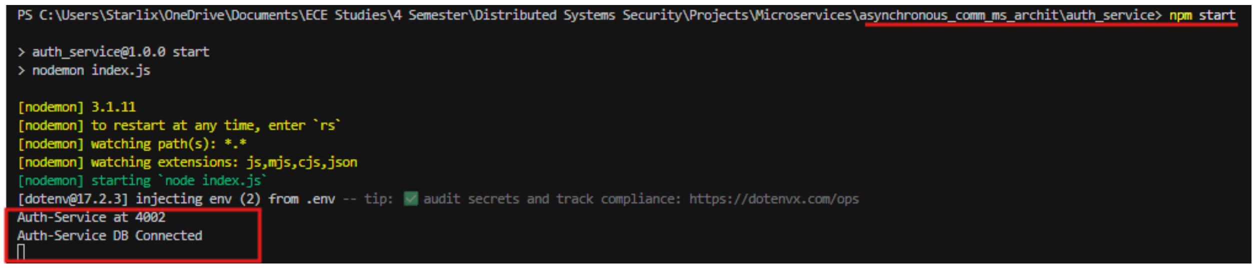
```
Auth-Service at 4002
Auth-Service DB Connected
```

Fig 7.2.1a: *Auth Service running* :



```
Order-Service running at port 4001
Order-Service DB Connected
RabbitMQ connected (Order-Service)
```

Fig 7.2.1b: *order_service running with RabbitMQ connected*



```
PS C:\Users\Starlix\OneDrive\Documents\ECE Studies\4 Semester\Distributed Systems Security\Projects\Microservices\asynchronous_comm_ms_archit\auth_service> npm start

> auth_service@1.0.0 start
> nodemon index.js

[nodemon] 3.1.11
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
[dotenv@17.2.3] injecting env (2) from .env -- tip: ✅ audit secrets and track compliance: https://dotenvx.com/ops
Auth-Service at 4002
Auth-Service DB Connected
```

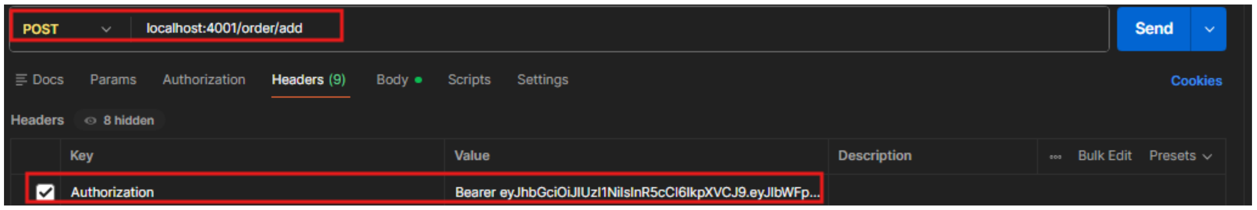
Fig 7.2.1c: *product_service/consumer.js running*

Step 2: Create a new order (order_service)

➔ Add header: Authorization: Bearer <token>

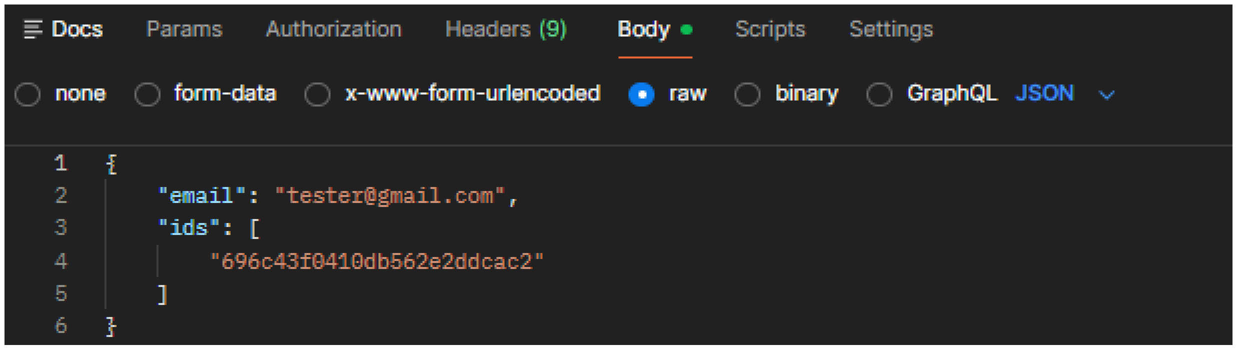
➔ Call POST http://localhost:4001/order/add (RabbitMQ enabled) with Authorization header

➔ **INPUT**



POST localhost:4001/order/add

Key	Value	Description
✓ Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFp...	

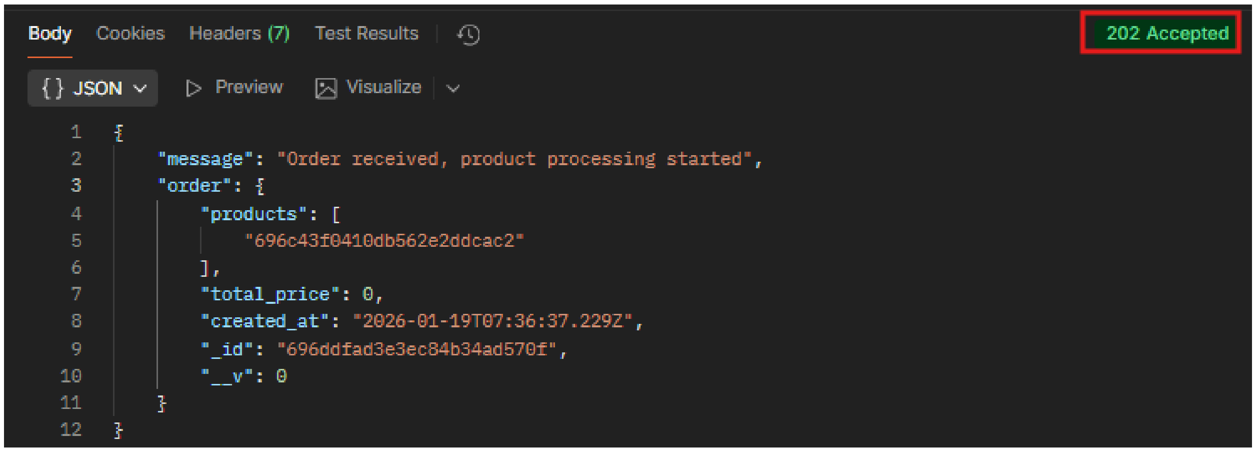


[✓ raw] JSON

```
{
    "email": "tester@gmail.com",
    "ids": [
        "696c43f0410db562e2ddcac2"
    ]
}
```

Fig 7.2.2a: Order creation request

➔ **OUTPUT**:

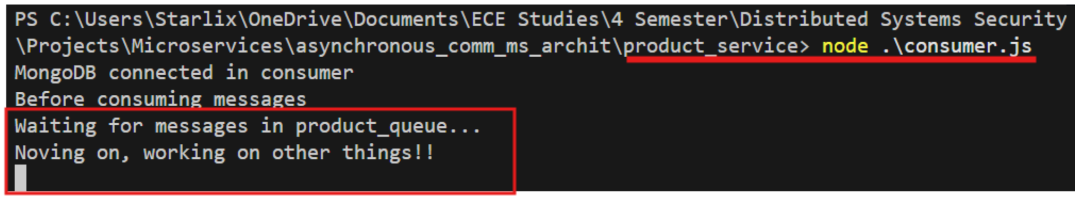


202 Accepted

```
{
    "message": "Order received, product processing started",
    "order": {
        "products": [
            "696c43f0410db562e2ddcac2"
        ],
        "total_price": 0,
        "created_at": "2026-01-19T07:36:37.229Z",
        "_id": "696ddfad3e3ec84b34ad570f",
        "__v": 0
    }
}
```

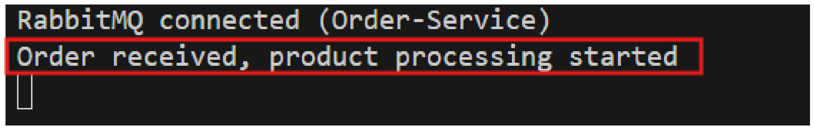
Fig 7.2.2b: Successful Order Creation

7.3: Observation for case 1: Consumer.js running Before Order request



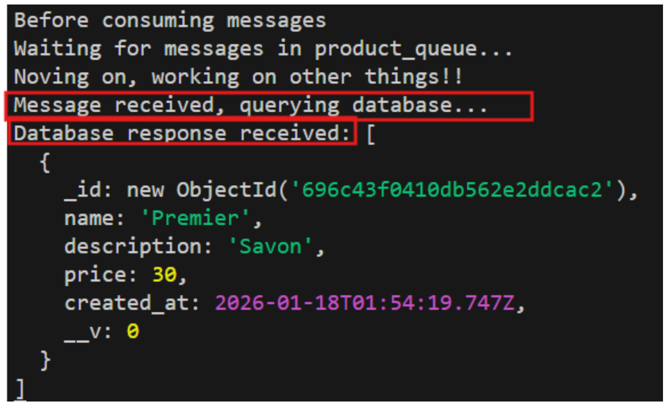
```
PS C:\Users\Starlix\OneDrive\Documents\ECE Studies\4 Semester\Distributed Systems Security
\Projects\Microservices\asynchronous_comm_ms_archit\product_service> node .\consumer.js
MongoDB connected in consumer
Before consuming messages
Waiting for messages in product_queue...
Noving on, working on other things!!
```

Fig 7.3.1a: Consumer log showing that next task executed while previous is still processing



```
RabbitMQ connected (Order-Service)
Order received, product processing started
```

Fig 7.3.1b: Order_Service log showing that message has been sent



```
Before consuming messages
Waiting for messages in product_queue...
Noving on, working on other things!!
Message received, querying database...
Database response received: [
  {
    _id: new ObjectId('696c43f0410db562e2ddcac2'),
    name: 'Premier',
    description: 'Savon',
    price: 30,
    created_at: 2026-01-18T01:54:19.747Z,
    __v: 0
  }
]
```

Fig 7.3.1c: Consumer log showing that message has been delivered by RabbitMQ to the consumer and processed

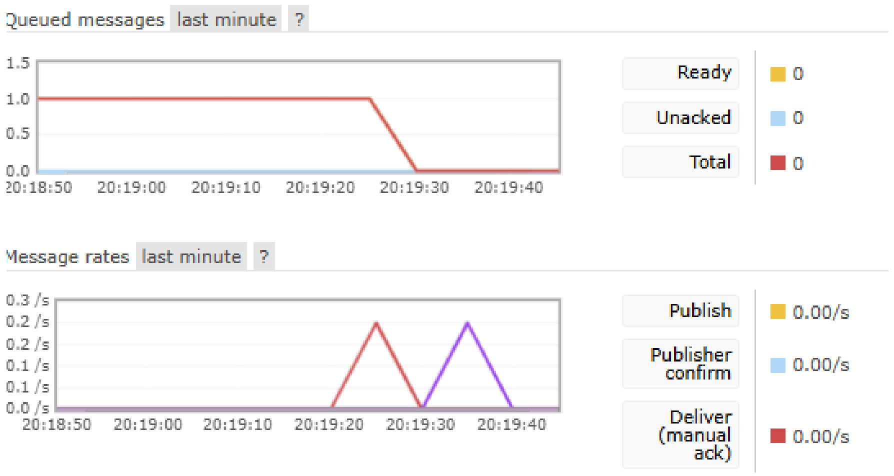
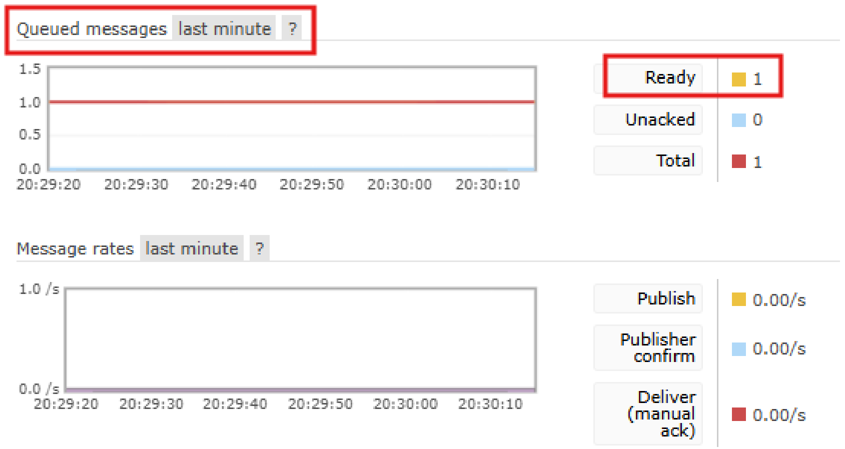


Fig 7.3.1d: Message delivery status(delivered) as seen on RabbitMQ UI

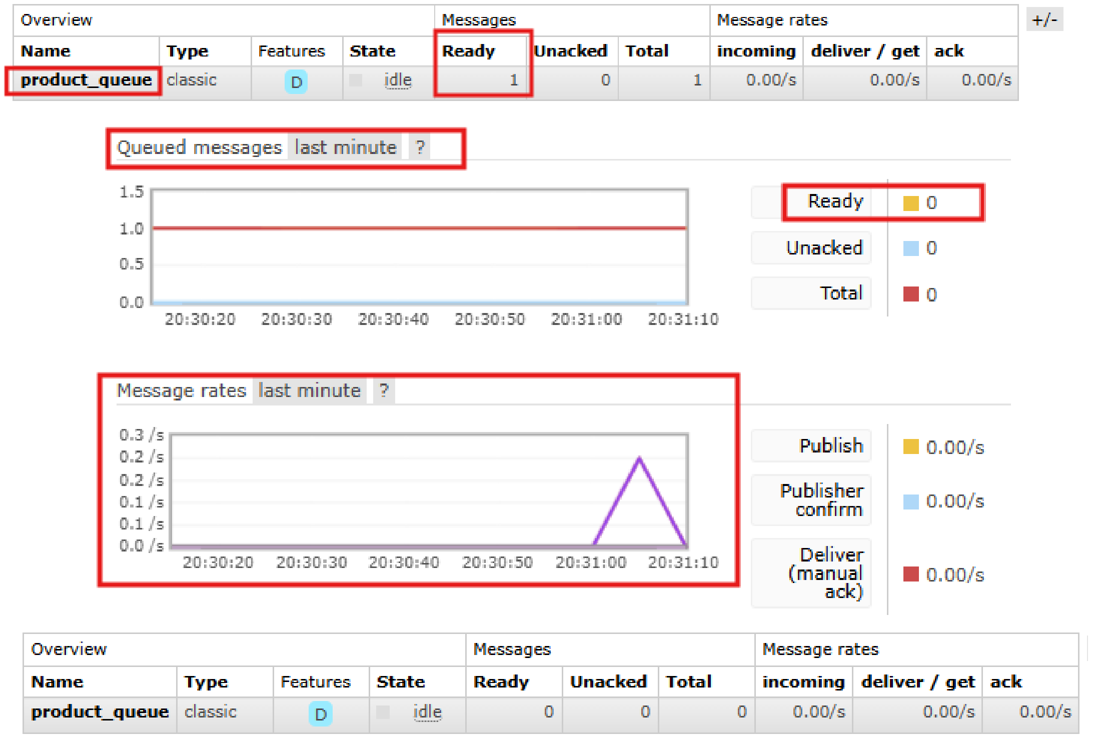
7.4: Observation for case 2 where is Consumer.js Stopped Before Order Creation request



Ready	🟨 1
Unacked	🟦 0
Total	🟥 1

Publish	🟨 0.00/s
Publisher confirm	🟦 0.00/s
Deliver (manual ack)	🟥 0.00/s

Fig 7.4.1: Message delivery status(on queue) when consumer.js is not running



Overview				Messages			Message rates		
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
product_queue	classic	D	idle	1	0	1	0.00/s	0.00/s	0.00/s

Ready	🟨 0
Unacked	🟦 0
Total	🟥 0

Publish	🟨 0.00/s
Publisher confirm	🟦 0.00/s
Deliver (manual ack)	🟥 0.00/s

Overview				Messages			Message rates		
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
product_queue	classic	D	idle	0	0	0	0.00/s	0.00/s	0.00/s

Fig 7.4.2: Message delivery status(delivered) after consumer.js was started - message was delivered

Conclusion:

The tests confirm that synchronous communication tightly couples services and fails immediately when the service that is supposed to provide response is unavailable, while asynchronous communication decouples services by using a message queue, allowing orders to succeed even when consumer is down. Messages are stored in the queue and processed later, when service starts running.

FINAL CONCLUSION

This step-by-step implementation shows how a secure microservices-based system is built and validated from the ground up. Each service is defined with a clearly defined responsibility.

The implementation was carried out using Node.js and Express for server-side service logic, MongoDB for data storage, JSON Web Tokens (JWT) for authentication and authorization, HTTP and Axios for synchronous communication, and RabbitMQ for asynchronous message-based communication. The workflow covered environment setup, database configuration, data modeling, business logic and routing implementation, middleware integration, and systematic testing with Postman and RabbitMQ Management Interface.

The resulting system supports:

- Independent microservices for Product, Order, and Authentication
- Persistent data storage using MongoDB within each service.
- Synchronous service interaction using HTTP and axios where immediate responses are required.
- Asynchronous processing using RabbitMQ to enable delayed and resilient communication between services.
- Clear separation of services across services, models, business logic, and message handlers.

The project reinforced key concepts such as service isolation, HTTP communication, message-based processing, and structured microservices.

Finally, this report provides a reproducible, step-by-step operational guide that allows the entire system to be implemented, tested, and validated reliably in both synchronous and asynchronous modes.

APPENDIX

A. API Endpoint Reference

Method	Endpoint	Description	Authentication
POST	/auth/register	Create user	No
POST	/auth/login	Login	No
POST	/productAdd	Add products	No
GET	/productBuy	Retrieve product	No
POST	/orderAdd	Create order	Yes

B. Collection and Requests Created on Postman

Add requests for each endpoint using HTTP method + URL:

- - POST localhost:4002/auth/register	-> create user (password hashed)
- - POST localhost:4002/auth/login	-> login, returns JWT
- - POST localhost:4000/product/add	->add product
- - GET localhost:4000/product/buy	-> Retrieve product by their ID
- - POST localhost:4001/order/add	-> create new order

C. How Order-Service communicates with Product-Service

Aspect	Synchronous	Asynchronous
Communication trigger	HTTP request	RabbitMQ message
Blocking	Yes	No
Transport	REST (Axios / Fetch)	AMQP
Files involved	`index.js` only	`index.js` + `producer.js` / `consumer.js`

Repository Link: