



**AIM: Demonstrate the use of Mongoose with CRUD operation. Problem Definition:**

1. **Generate Database model**
2. **Create RESTful API**
3. **Demonstrate the Endpoints.**

**Resources used:**

* + Node.js
  + Express.js
  + Mongoose
  + MongoDB
  + Postman (for testing API)



# Expected OUTCOME of Experiment:

**CO 2:** Ability to implement CRUD operations using Mongoose and create RESTful APIs.



# Books/ Journals/ Websites referred:

1. Shelly Powers Learning Node O’ Reilly 2 nd Edition, 2016.
2. Mongoose Documentation
3. Express Documentation

# Pre Lab/ Prior Concepts:

**Write details about the following content**

* + - Mongoose CRUD operation

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js, designed to simplify the interaction between an application and a MongoDB database. It provides a schema-based solution to model data, ensuring that the application has a clear structure for its data and that it adheres to defined constraints. The primary operations that can be performed using Mongoose are known as CRUD operations, which stand for Create, Read, Update, and Delete.

# Create:

* + The **Create** operation allows you to insert new documents into a MongoDB collection. In Mongoose, this is typically achieved through the save() method or the create() method. When creating a document, Mongoose validates the data against the defined schema, ensuring that all required fields are present and meet any specified constraints (like data types or unique values).

# Read:

* + The **Read** operation retrieves documents from a collection. Mongoose provides several methods for reading data, including find(), findOne(), and findById(). These methods allow you to query the database using various conditions, and they return the documents that match those conditions. The ability to read data is crucial for displaying information to users and performing data analysis.

# Update:

* + The **Update** operation modifies existing documents within a collection. Mongoose offers several methods for updating data, such as updateOne(), updateMany(), and findByIdAndUpdate(). These methods allow developers to change specific fields in a document, ensuring that the application's data remains current and accurate. The update process can also include validations based on the schema.

# Delete:

* + The **Delete** operation removes documents from a collection. In Mongoose, you can use methods like deleteOne(), deleteMany(), and findByIdAndDelete() to remove one or multiple documents based on certain criteria. This operation is essential for data management, allowing applications to clear out outdated or irrelevant data.

These CRUD operations form the foundation for interacting with a MongoDB database using Mongoose, enabling developers to perform essential data management tasks effectively

* RESTFul API

**REST (Representational State Transfer)** is an architectural style for designing networked applications. It relies on a stateless, client-server communication protocol (usually HTTP) and uses standard HTTP methods (GET, POST, PUT, DELETE).

# Key principles of RESTful API:

* + Stateless: Each request from the client contains all the information the server needs to fulfill that request.
  + Client-Server: Separation of concerns between the user interface and data storage.
  + Uniform Interface: A standard way to communicate (e.g., using standard HTTP methods).

A REST API fundamentally relies on three major elements:

* **Client.** The client is the software code or application that requests a resource from a server.
* **Server.** The server is the software code or application that controls the resource and responds to client requests for the resource.
* **Resource.** The resource is any data or content, such as text, video and images, the server controls and makes available in response to client requests.

# Implementation Details:

**Step 1**: Set Up Environment

* Create a new directory for your project and run npm init to set up the package.json file.

**Step 2**: Install Required Packages

* Install the required dependencies using npm:

**Step 3**: Generate Database Model

* Create a Mongo data base Index.js

const express = require("express"); const mongoose = require("mongoose"); require("dotenv").config();

const cors = require("cors");

const { userRouter } = require("./routes/userRouter"); const { noteRouter } = require("./routes/noteRouter");

const port = process.env.PORT || 3000; const mongoURI = process.env.MONGO\_URI;

if (!mongoURI) {

console.error("MongoDB URI is not defined. Please check your .env file."); process.exit(1); }

const app = express(); app.use(cors()); app.use(express.json()); app.use("/user", userRouter); app.use("/notes", noteRouter);

app.get("/", (req, res) => { res.send({ message: "API working" });

});

mongoose

.connect(mongoURI, { serverSelectionTimeoutMS: 5000, })

.then(() => console.log("Database connected"))

.catch((err) => {

console.log("MongoDB connection error: ", err); process.exit(1);

});

app.listen(port, () => {

console.log(`Server running on port ${port}`);

});

Db.js and .env

const mongoose = require("mongoose");

require("dotenv").config();

const connection = mongoose.connection(); module.exports = {

connection,

};

MONGO\_URI="mongodb+srv://sharwarp:TPI2kE1Y2ptrTGWS@cluster0.u5pu5.mongodb.net/?retryWri tes=true&w=majority&appName=Cluster0"

PORT=4000

Generate Database Model

* Create a file named UserModel.js in a models directory and define the schem Create a file named UserModel.js in a models directory and define the schema

const mongoose = require("mongoose");

const userSchema = new mongoose.Schema({ name: {

type: String, required: true,

},

email: {

type: String, required: true, unique: true,

},

password: { type: String, required: true,

},

});

const UserModel = mongoose.model("User", userSchema);

module.exports = { UserModel };

# Step 5: Set Up User Routes (userRouter.js)

* Create a file named userRouter.js in the routes directory and set up user-related endpoints

# Register and Get user

const express = require("express"); const userRouter = express.Router();

const { UserModel } = require("../models/UserModel");

userRouter.get("/", (req, res) => { res.send("Hello World!");

});

userRouter.post("/register", (req, res) => { const { name, email, password } = req.body;

bcrypt.hash(password, 5, async function (err, hash) {

if (err) return res.send({ message: "something went wrong", status: 0 }); try {

let user = new UserModel({ name, email, password: hash }); await user.save();

res.send({

message: "User created", status: 1,

});

} catch (error) { res.send({

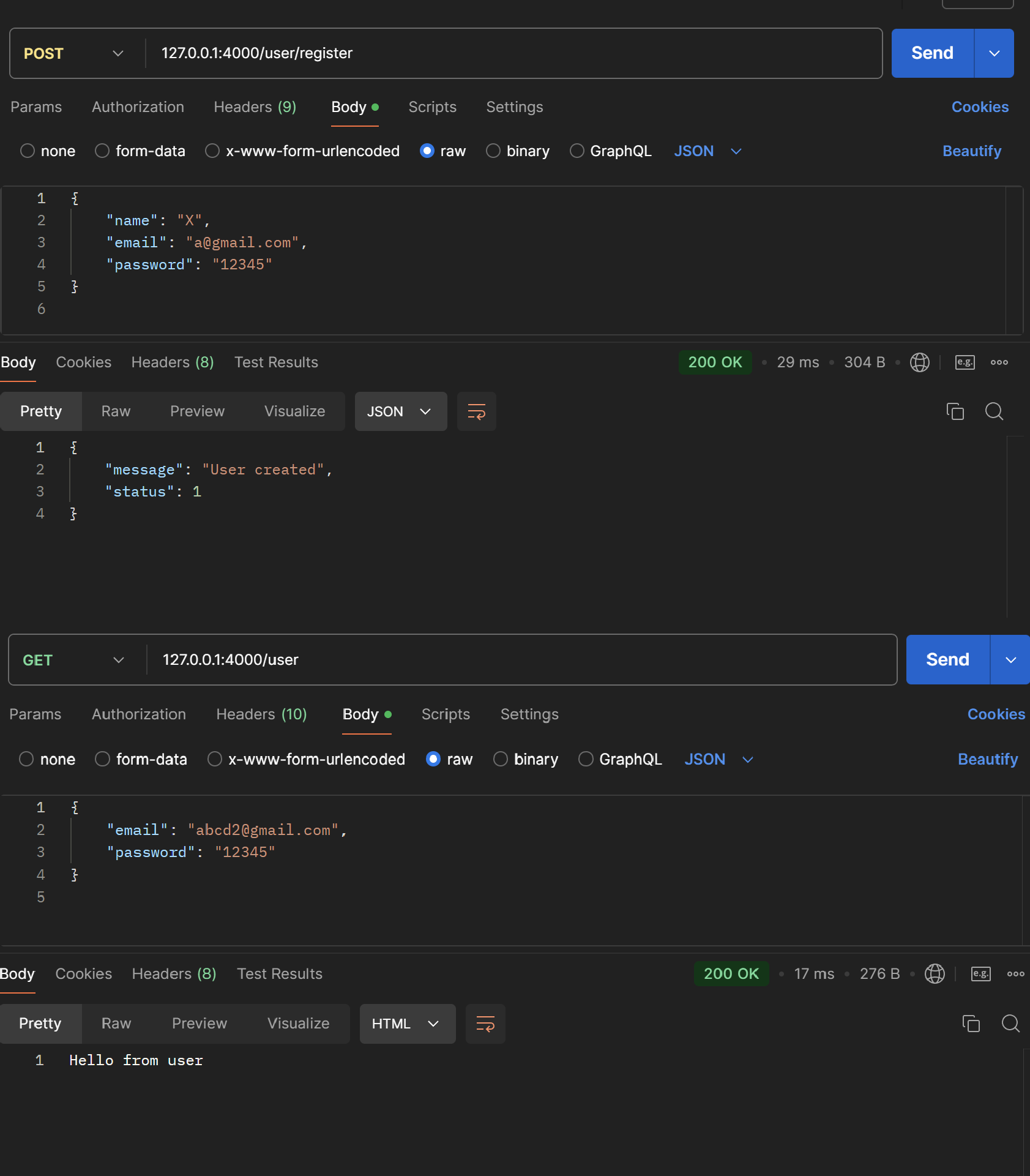
message: error.message, status: 0,

});

}

});

});



2. Login

userRouter.post("/login", async (req, res) => { const { email, password } = req.body;

const option = { expiresIn: "1h" };

try {

const user = await UserModel.findOne({ email });

if (!user) {

return res.status(404).send({ message: "User does not exist", status: 0,

});

}

const isMatch = await bcrypt.compare(password, user.password);

if (!isMatch) {

return res.status(401).send({ message: "Incorrect password", status: 0,

});

}

const token = jwt.sign({ userId: user.\_id }, "xyzm", option);

res.status(200).send({ message: "Login successful", token: token,

status: 1,

});

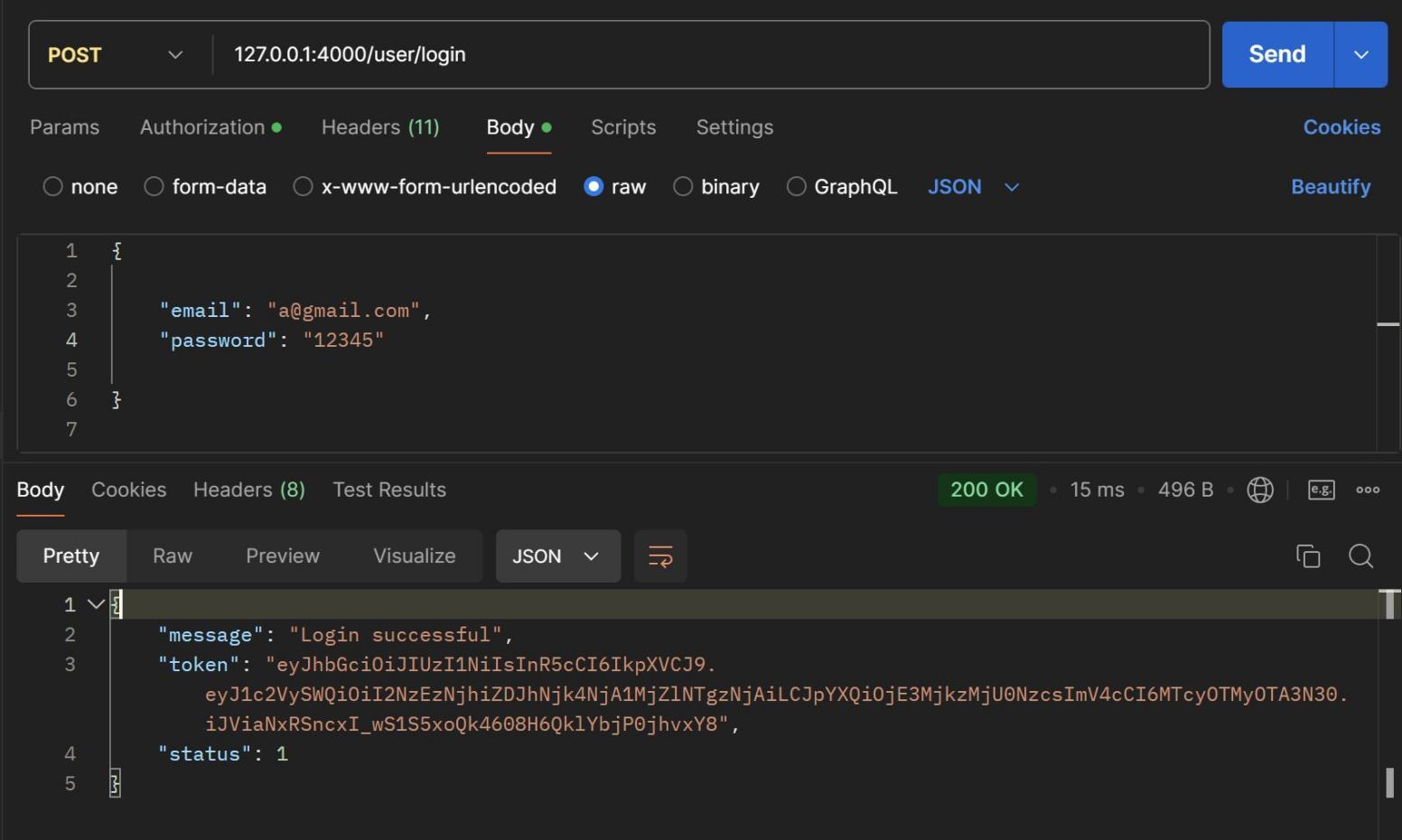
} catch (error) { res.status(500).send({

message: "Something went wrong", status: 0,

});

}

});



1. Delete a user

userRouter.delete("/delete", async (req, res) => {

const token = req.headers.authorization.split(" ")[1]; // Expecting token in Bearer format

if (!token) {

return res.status(401).send({ message: "No token provided", status: 0,

});

}

try {

const decoded = jwt.verify(token, "xyzm");

const deletedUser = await UserModel.findByIdAndDelete(decoded.userId);

if (!deletedUser) {

return res.status(404).send({ message: "User not found",

status: 0,

});

}

res.send({

message: "User deleted successfully", status: 1,

});

} catch (error) { res.status(500).send({

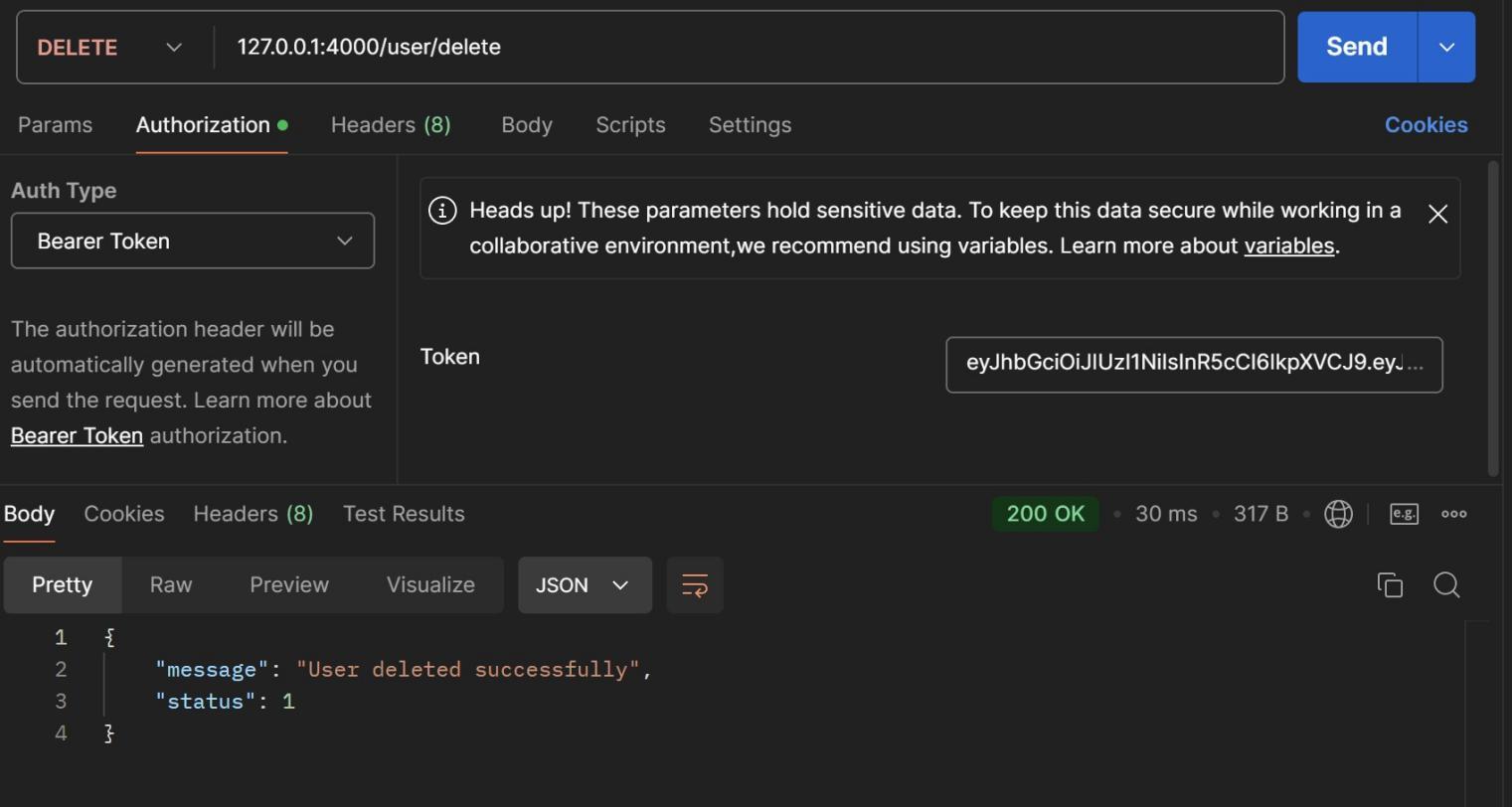
message: "Something went wrong", status: 0,

});

}

});

module.exports = { userRouter };



# Steps for execution:

* 1. **Run MongoDB**: Ensure your MongoDB instance is running.
  2. **Start the Server**: Run the Express server using: node index.js

# Test with Postman:

**Registration**: POST http://localhost:4000/user/register

**Login**: POST http://localhost:4000/user/login

**Delete User**: DELETE http://localhost:4000/user/delete (Include token in headers)

# Conclusion:

In this project, we successfully implemented a user management system using Node.js, Express, and MongoDB, allowing users to register, log in, and delete their accounts securely. We learned to create a robust backend environment with Mongoose for database modeling and JWT for user authentication. This experience enhanced our understanding of RESTful APIs and the importance of secure coding practices in web development.