**Batch: C1**

**Roll No.: 16010122221**

**Experiment / assignment / tutorial No. 9**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

Title: Create a RESTful API server in Express and Node.js. Implementation + Testing application u postman/Thurderclient

**AIM:** Create a RESTful API server in Express and Node.js. Implementation + Testing application using postman/Thurderclient

## Problem Definition:

In this experiment, we will create a RESTful API server using Express and Node.js. The API will allow users to perform basic CRUD operations such as registering, logging in, deleting users, and retrieving all users. The functionality will be tested using Postman/Thunderclient.

## Resources used:

* Node.js
* Express.js
* Mongoose (for MongoDB integration)
* Postman/Thunderclient (for testing API endpoints)

## Expected OUTCOME of Experiment:

**CO 3: Test the concepts and components of various front-end, back-end web app development technologies & frameworks using web development tools.**

## Books/ Journals/ Websites referred:

* 1. Shelly Powers Learning Node O’ Reilly 2 nd Edition, 2016.

## Pre Lab/ Prior Concepts:

**Write details about the following content**

* Testing in POSTMAN

Postman is a widely used tool for testing APIs by simulating HTTP requests and analyzing responses. It provides a user-friendly interface to make API requests without the need for writing code. The main features of testing in Postman include:

1. **HTTP Methods**: Postman allows testing of all HTTP methods (GET, POST, PUT, DELETE, PATCH, etc.), which are essential for interacting with RESTful APIs. You can specify the request method to test API endpoints for CRUD operations.
2. **Request Configuration**: Postman enables you to set up requests with headers, query parameters, path variables, body data, and authorization (like Bearer tokens, API keys, or basic auth). This allows detailed customization of each request to simulate real-world API consumption scenarios.
3. **Response Inspection**: After making a request, Postman shows the response from the server, including status codes (e.g., 200 OK, 404 Not Found, 401 Unauthorized), response body (JSON, XML, HTML), response time, and headers. It helps to quickly identify if the API is functioning as expected or if there are issues in request handling or data returned.

Overall, Postman is a powerful tool for developers to test APIs, debug errors, and ensure the correctness and security of web services.

**Implementation Details:**

# Step 1: Set Up Your Environment

1. **Create a New Directory for Your Project and Initialize a New Node.js Project:**

# Step 2: Install Required Packages

Install the necessary packages, including express, mongoose, and dotenv: npm install express mongoose dotenv

# Step 3: Create a Basic Server

Create an index.js file for your server:

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}`);

});

# Step 4: Create a Database Model

Create a models directory and a UserModel.js file within it:

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 Routes for CRUD Operations

Create a routes directory and a userRouter.js file within it:

const express = require("express"); const userRouter = express.Router(); const bcrypt = require("bcrypt"); const jwt = require("jsonwebtoken");

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

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

});

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,

});

}

});

});

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,

});

}

});

userRouter.delete("/delete", async (req, res) => { const token = req.headers.authorization.split(" ")[1];

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,

});

}

});

userRouter.get("/all", async (req, res) => { try {

const users = await UserModel.find(); res.status(200).send({

message: "Users fetched successfully", users: users,

status: 1,

});

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

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

});

}

});

module.exports = { userRouter };

# Step 6: Integrate the Routes into Your Server

Update the index.js file to include the user routes:

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

# Step 7: Create a .env File

Create a .env file in the root directory of your project with your MongoDB URI:

MONGO\_URI="mongodb+srv://password@cluster0.u5pu5.mongodb.net/?retryWrites=true&w=majori ty&appName=Cluster0"

PORT=4000

# Step 8: Start the Server

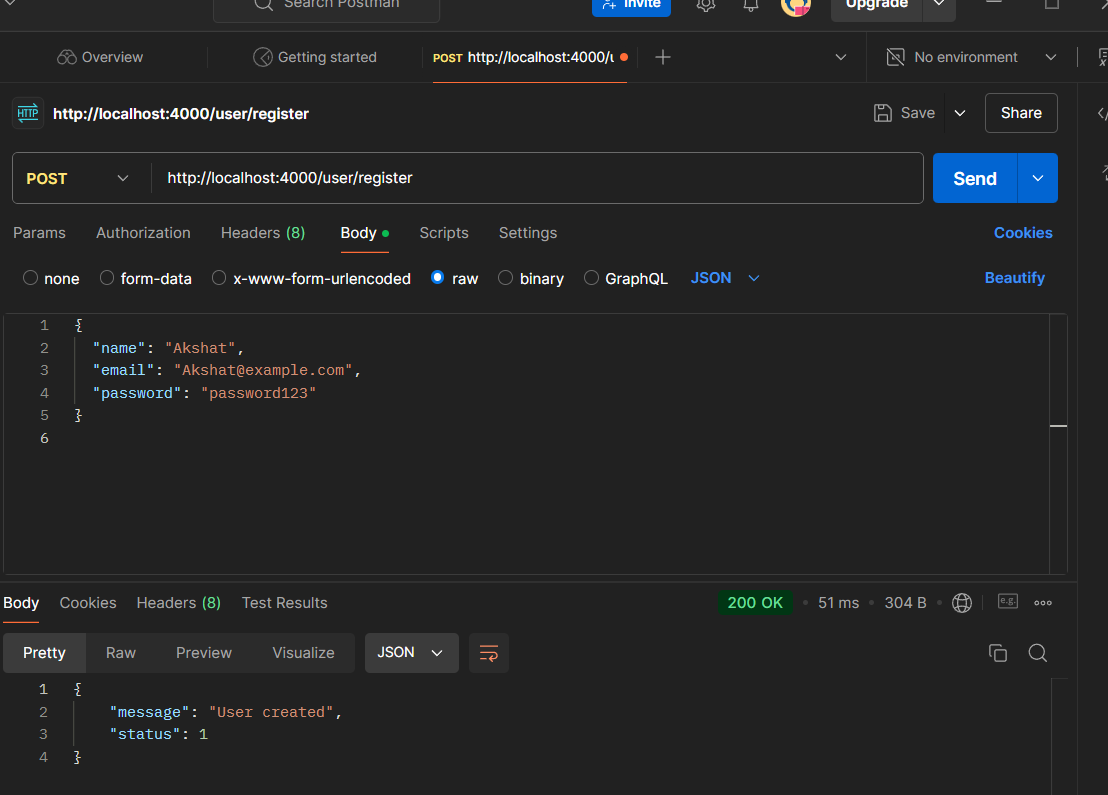
Run your server:

node index.js

# Step 9: Testing the API with Postman or Thunder Client

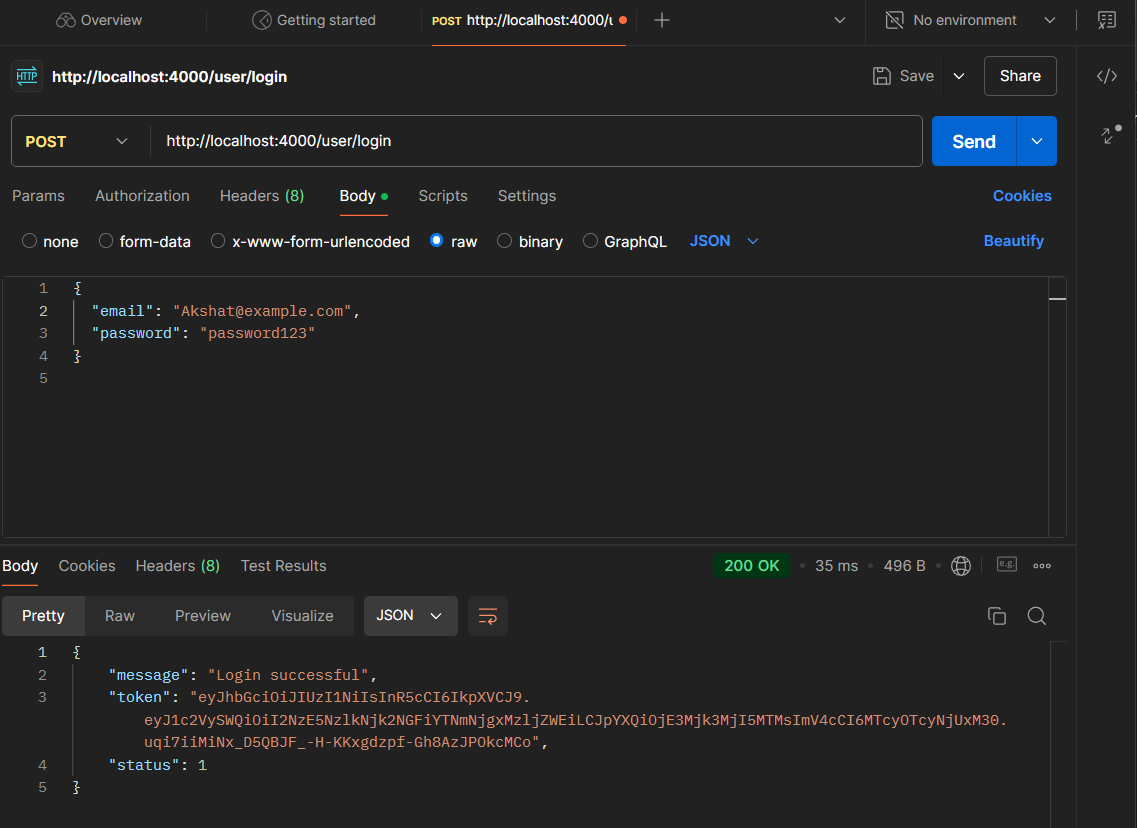
## Register a User

* + **URL**: http://localhost:4000/user/register
  + **Method**: POST
  + **Body** (JSON):



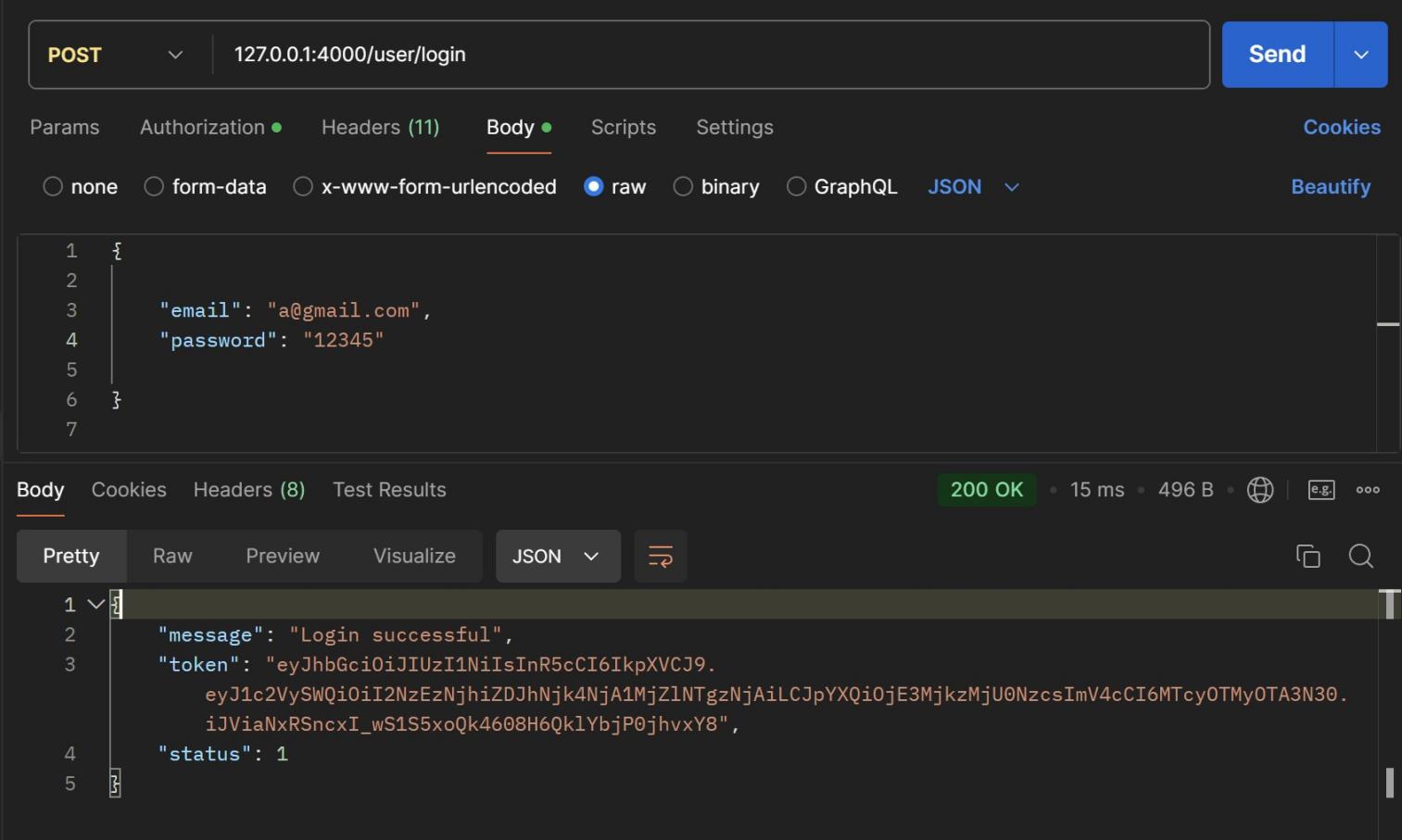
## Get All Users

* + **URL**: http://localhost:4000/user/login
  + **Method**: GET



## Login a User

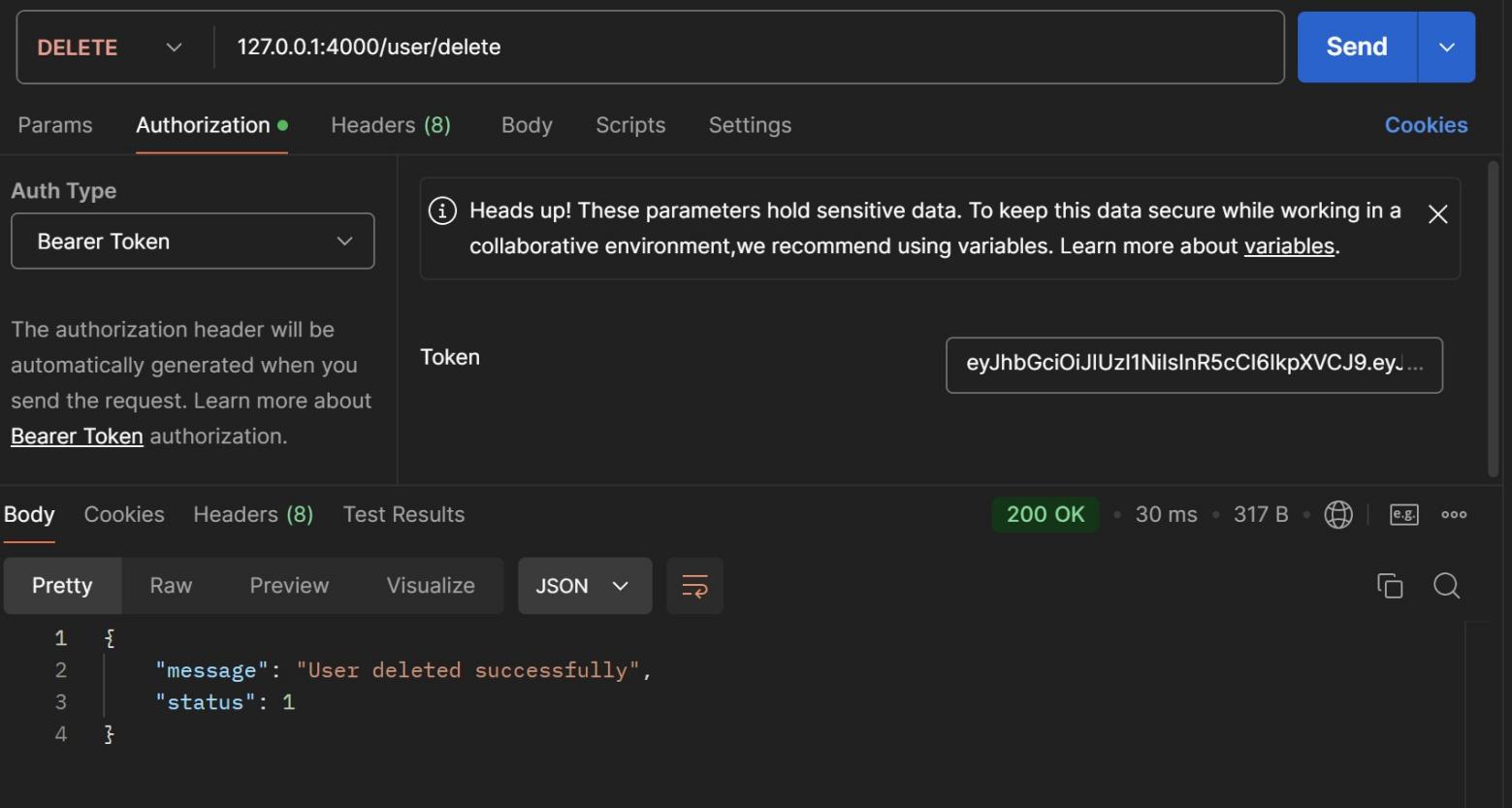
* + **URL**: http://localhost:4000/users/login
  + **Method**: POST
  + **Body** (JSON):



## Delete a User

* + **URL**: http://localhost:4000/users/{userId}
  + **Method**: DELETE

Replace {userId} with the actual user ID from the database.



## Conclusion:

In this experiment, we successfully implemented and tested a RESTful API server using Express and Node.js. The API's endpoints were tested with Postman/Thunderclient, validating the correct functionality of user registration, login, deletion, and retrieval processes. The experiment demonstrated effective integration of backend technologies and testing tools in real-world API development scenarios.