Experiment 4

Aim:REST API Design with MongoDB + Mongoose Integration

Theory:

A REST API (Representational State Transfer API) is an architectural style that enables communication between client and server applications over HTTP using stateless operations such as GET, POST, PUT, and DELETE. It allows clients to access and manipulate resources through well-defined endpoints, making systems modular, scalable, and language-independent.

MongoDB is a NoSQL database that stores data in flexible, JSON-like documents, which makes it highly adaptable for applications requiring dynamic schemas. However, working directly with MongoDB can become complex when enforcing validations or managing data structures.

To address this, Mongoose is used as an Object Data Modeling (ODM) library for MongoDB. Mongoose provides schema-based modeling, validation, middleware, and query-building capabilities, bridging the gap between MongoDB's flexible document structure and application requirements.

By integrating MongoDB with Mongoose in a REST API, developers can:

- Define clear schemas for data consistency.
- Implement validations and constraints at the application layer.
- Use middleware for request handling and data transformations.
- Perform CRUD operations with cleaner and more maintainable code.

This design results in APIs that are scalable, consistent, and easier to maintain, making it a preferred choice in modern backend development.

CODE:

src/config/db.js:

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

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGO_URI);
    console.log(`MongoDB Connected: ${conn.connection.host}`);
  } catch (error) {
    console.error(`Error connecting to MongoDB: ${error.message}`);
    process.exit(1);
  }
};

module.exports = connectDB;
```

src/controllers/userController.js:

```
const User = require('../models/userModel');

const createUser = async (req, res, next) => {
  try {
    const user = await User.create(req.body);
    res.status(201).json(user);
  } catch (err) {
    // Duplicate key, validation etc. will be handled by error middleware
    next(err);
  }
};

// Get list (with simple pagination)
const getUsers = async (req, res, next) => {
  try {
    const page = Number(req.query.page) || 1;
    const limit = Math.min(Number(req.query.limit) || 10, 100);
    const skip = (page - 1) * limit;

    const users = await User.find().skip(skip).limit(limit).sort({ createdAt: -1 });
    const total = await User.countDocuments();
    res.json({ total, page, limit, data: users });
  } catch (err) {
    next(err);
  }
};

const getUserById = async (req, res, next) => {
  try {
    const user = await User.findById(req.params.id);
    if (!user) { res.status(404); throw new Error('User not found'); }
    res.json(user);
  } catch (err) { next(err); }
```

```
const updateUser = async (req, res, next) => {
  try {
    const updated = await User.findByIdAndUpdate(req.params.id, req.body, {
      new: true,
      runValidators: true
    });
    if (!updated) { res.status(404); throw new Error('User not found'); }
    res.json(updated);
  } catch (err) { next(err); }
};

const deleteUser = async (req, res, next) => {
  try {
    const deleted = await User.findByIdAndDelete(req.params.id);
    if (!deleted) { res.status(404); throw new Error('User not found'); }
    res.json({ message: 'User deleted' });
  } catch (err) { next(err); }
};

module.exports = { createUser, getUsers, getUserById, updateUser, deleteUser };
```

src/middleware/errorMiddleware.js

```
const notFound = (req, res, next) => {
  res.status(404);
  const err = new Error(`Not Found - ${req.originalUrl}`);
  next(err);
};

const errorHandler = (err, req, res, next) => {
  const statusCode = res.statusCode === 200 ? 500 : res.statusCode;
  res.status(statusCode);
  res.json({
    message: err.message,
    stack: process.env.NODE_ENV === 'production' ? '🥞' : err.stack
  });
};

module.exports = { notFound, errorHandler };
```

src/models/userModel.js:

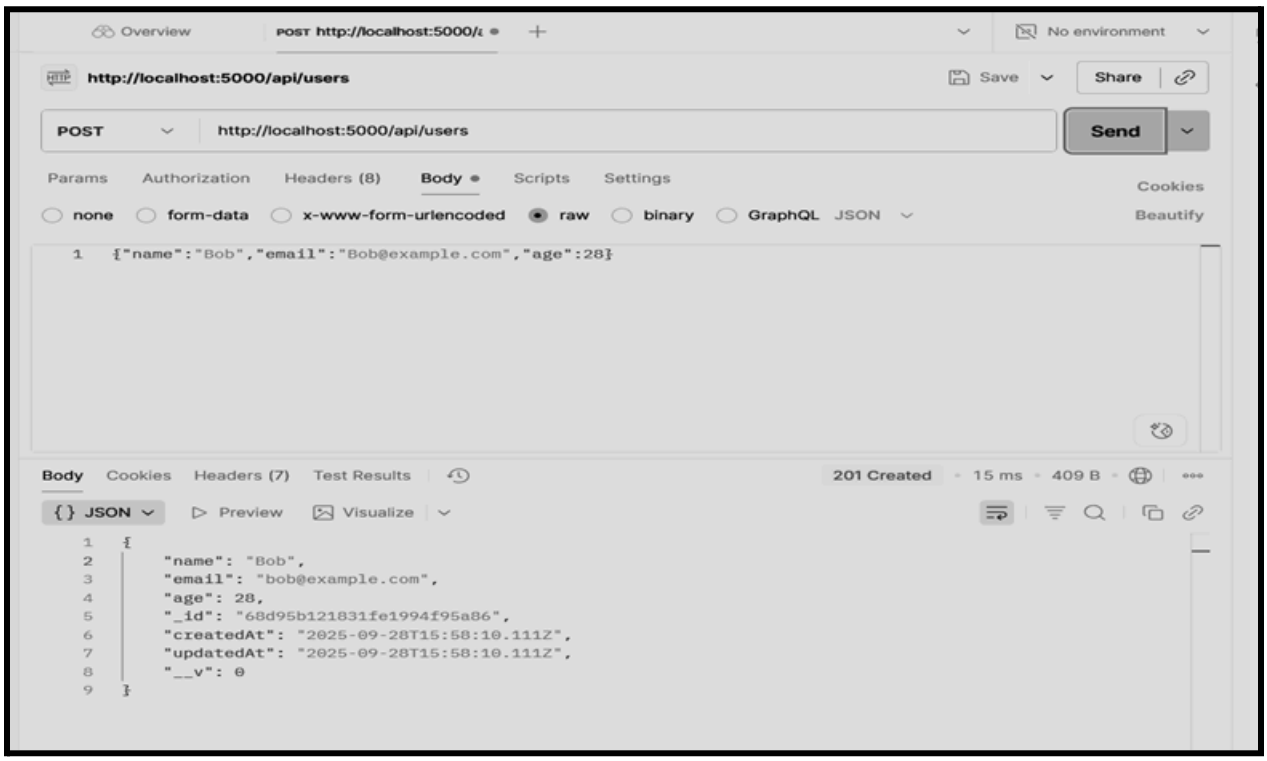
```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: { type: String, required: [true, 'Name required'] },
  email: { type: String, required: [true, 'Email required'], unique: true, lowercase: true },
  age: { type: Number, min: 0 },
}, {
  timestamps: true
});

module.exports = mongoose.model('User', userSchema);
```

src/routes/userRoutes.js:

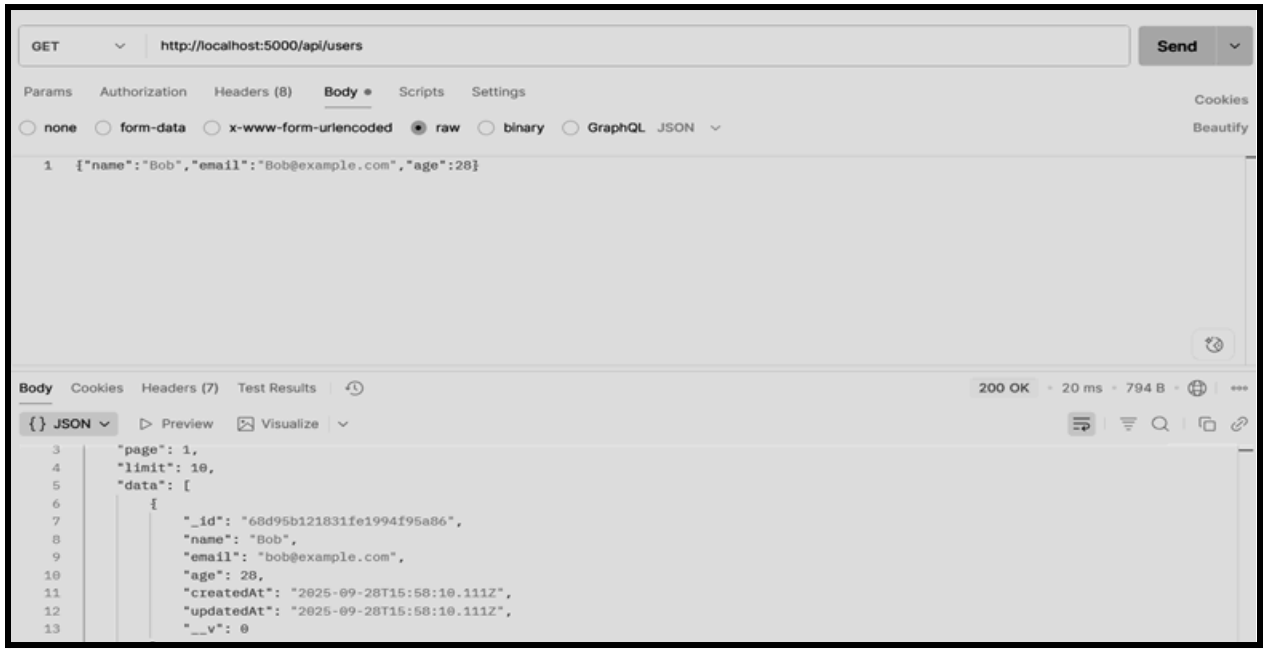
```
const express = require('express');
const router = express.Router();
const {
  createUser, getUsers, getUserById, updateUser, deleteUser
} = require('../controllers/userController');

router.route('/')
  .post(createUser)
  .get(getUsers);

router.route('/:id')
  .get(getUserById)
  .put(updateUser)
  .delete(deleteUser);

module.exports = router;
```

OUTPUT:

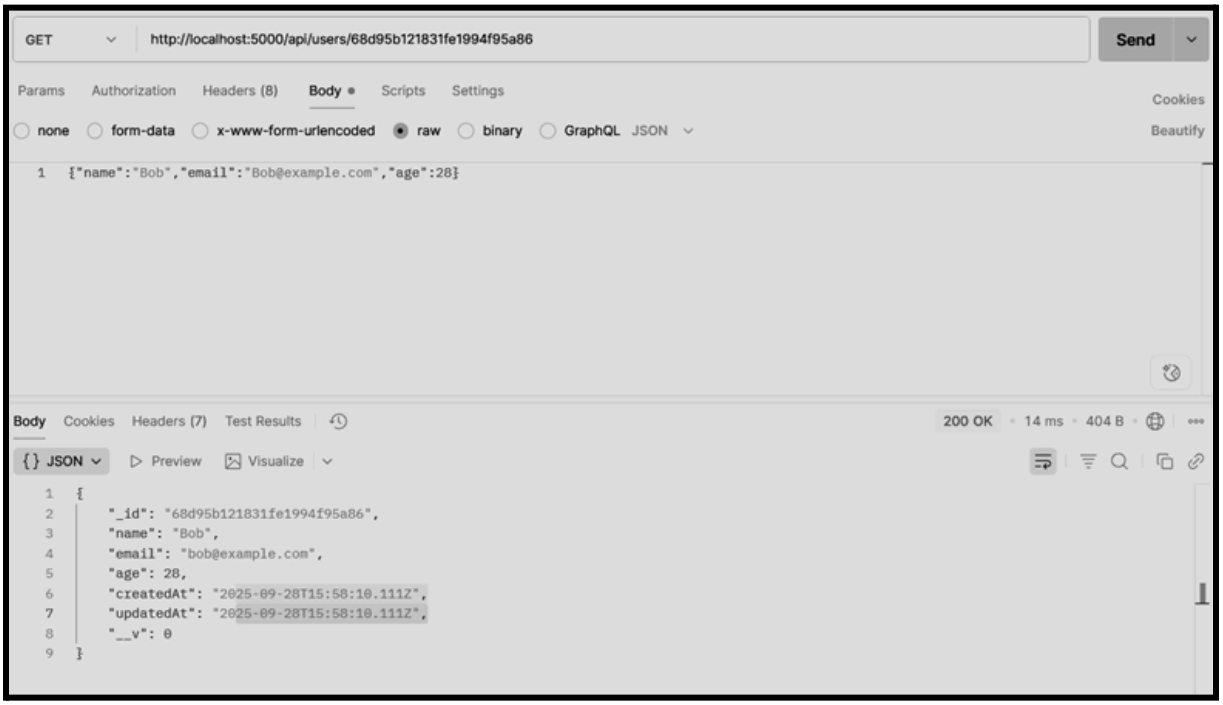
POST:



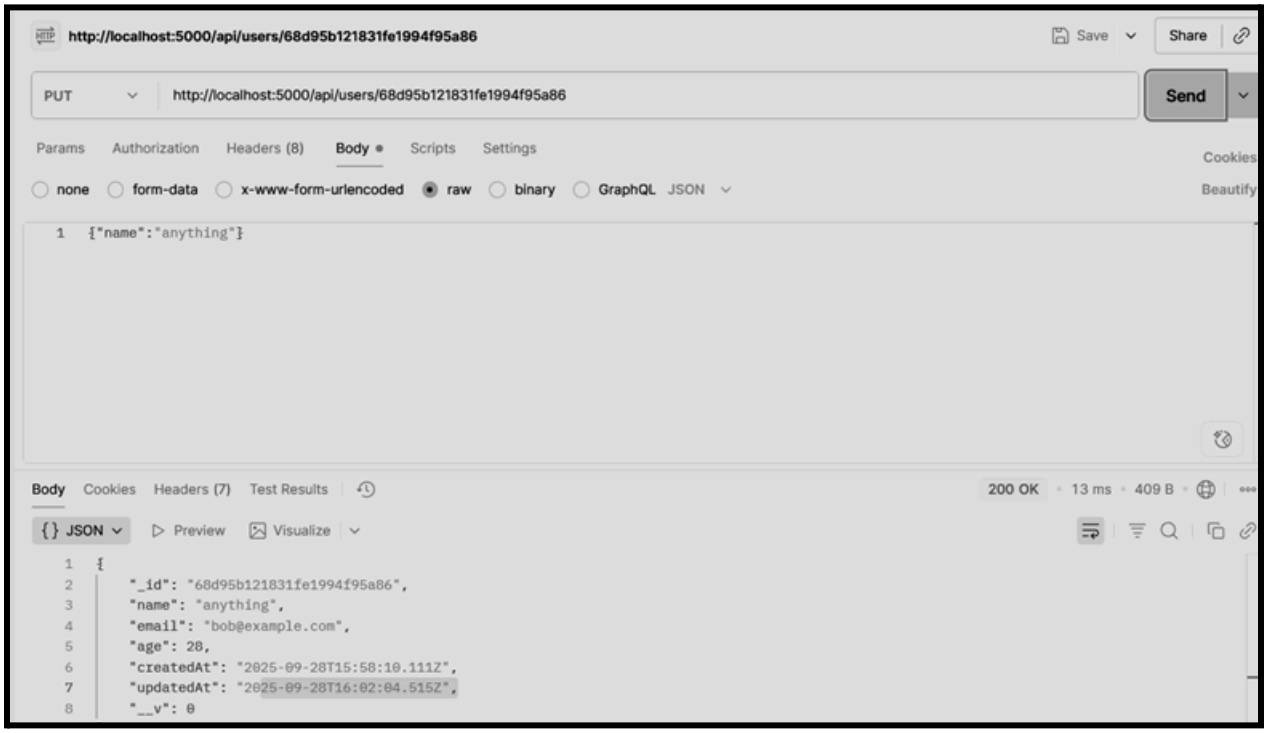
GET:



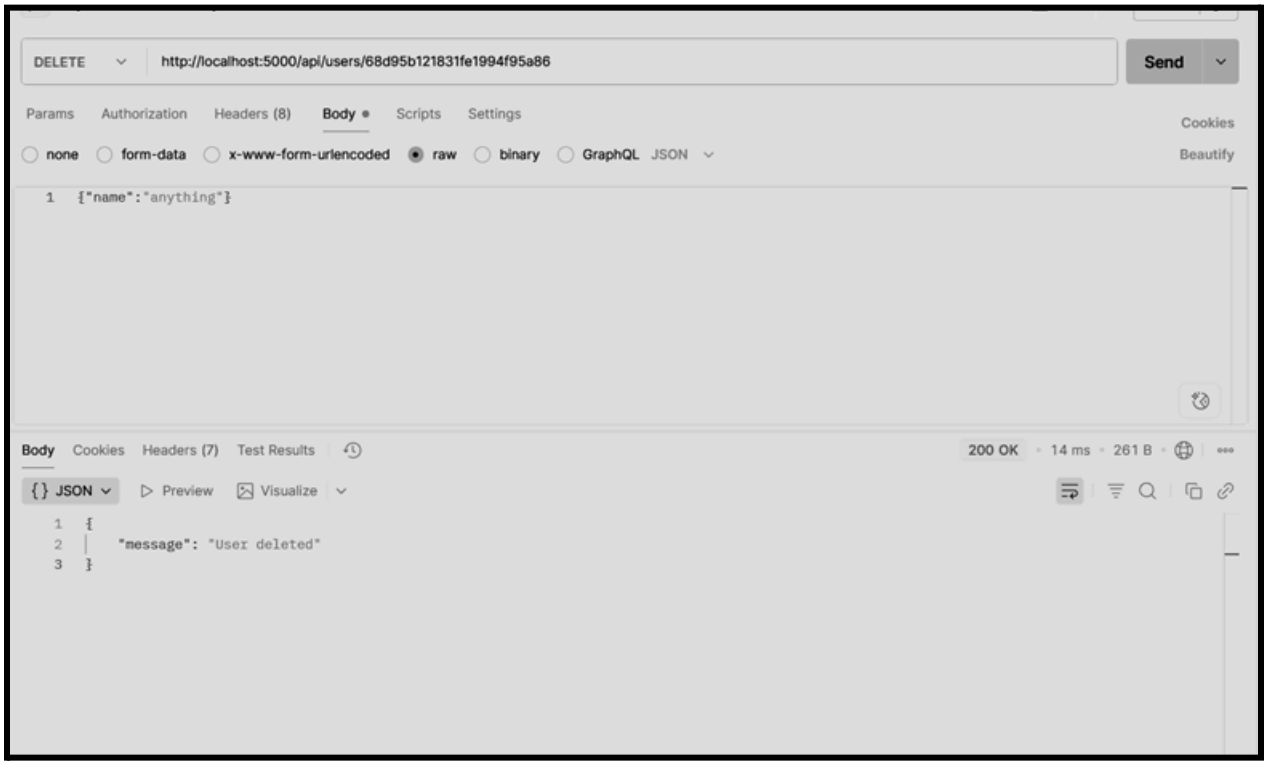
GET:


GET
http://localhost:5000/api/users/68d95b121831fe1994f95a86
Send
Params Authorization Headers (8) Body Scripts Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
1 {"name":"Bob","email":"Bob@example.com","age":28}
Body Cookies Headers (7) Test Results
200 OK 14 ms 404 B
{} JSON Preview Visualize
1 {
2 "_id": "68d95b121831fe1994f95a86",
3 "name": "Bob",
4 "email": "bob@example.com",
5 "age": 28,
6 "createdAt": "2025-09-28T15:58:10.111Z",
7 "updatedAt": "2025-09-28T15:58:10.111Z",
8 "__v": 0
9 }

PUT:


http://localhost:5000/api/users/68d95b121831fe1994f95a86
Save
Share
PUT
http://localhost:5000/api/users/68d95b121831fe1994f95a86
Send
Params Authorization Headers (8) Body Scripts Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
1 {"name":"anything"}
Body Cookies Headers (7) Test Results
200 OK 13 ms 409 B
{} JSON Preview Visualize
1 {
2 "_id": "68d95b121831fe1994f95a86",
3 "name": "anything",
4 "email": "bob@example.com",
5 "age": 28,
6 "createdAt": "2025-09-28T15:58:10.111Z",
7 "updatedAt": "2025-09-28T16:02:04.515Z",
8 "__v": 0

DELETE:



Conclusion :

REST API with MongoDB and Mongoose enables structured, scalable, and efficient data handling.
It bridges flexibility of NoSQL with schema validation for reliability.
Overall, it simplifies backend development and improves API maintainability.