Experiment 4

### **Aim**

To design and implement a **RESTful API** integrated with **MongoDB** using **Mongoose** in a Node.js and Express environment.  
 The objective is to enable secure, structured, and scalable communication between the frontend and backend of the Interview Simulator Website for managing users, questions, and interview data.

### **Theory**

**1. Introduction to REST APIs** A **REST (Representational State Transfer)** API is an architectural style for designing web services that allow systems to communicate over HTTP using standard methods like **GET**, **POST**, **PUT**, and **DELETE**.  
 Each resource (e.g., user, question, or interview) is represented by a unique URL endpoint, and clients interact with these endpoints using stateless requests.

**2. REST Principles**

* **Statelessness:** Each request contains all necessary information; the server does not store client state.
* **Uniform Interface:** Consistent structure for accessing resources.
* **Resource Representation:** Data is usually exchanged in **JSON** format.
* **Layered System:** Separation of client and server responsibilities.

**3. Introduction to MongoDB and Mongoose**

* **MongoDB:** A NoSQL database that stores data in JSON-like documents. It’s highly scalable and flexible for applications like interview simulation platforms.
* **Mongoose:** An ODM (Object Data Modeling) library for MongoDB that provides schema-based structure, validation, and query helpers.

**4. Why Mongoose?**

* Simplifies CRUD operations (Create, Read, Update, Delete).
* Adds schema validation and model-based organization.
* Helps avoid direct, unstructured interactions with MongoDB.
* Supports middleware and hooks for pre/post-processing.

### **Procedure**

#### **Step 1 – Setup Project and Install Dependencies**

mkdir rest-api-demo

cd rest-api-demo

npm init -y

npm install express mongoose dotenv cors

Create folder structure:

rest-api-demo/

├── server.js

├── .env

├── models/

│ └── Question.js

├── routes/

│ └── questionRoutes.js

├── controllers/

│ └── questionController.js

#### **Step 2 – Connect to MongoDB using Mongoose**

In server.js:

const express = require('express');

const mongoose = require('mongoose');

const cors = require('cors');

require('dotenv').config();

const app = express();

app.use(express.json());

app.use(cors());

// Connect to MongoDB

mongoose.connect(process.env.MONGO\_URI)

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

.catch((err) => console.error(err));

// Import routes

const questionRoutes = require('./routes/questionRoutes');

app.use('/api/questions', questionRoutes);

app.listen(5000, () => console.log('Server running on port 5000'));

Add .env:

MONGO\_URI = mongodb+srv://<username>:<password>@cluster.mongodb.net/interviewDB

#### **Step 3 – Define a Mongoose Schema and Model**

Create models/Question.js:

const mongoose = require('mongoose');

const questionSchema = new mongoose.Schema({

title: { type: String, required: true },

difficulty: { type: String, enum: ['Easy', 'Medium', 'Hard'], default: 'Medium' },

category: { type: String, required: true },

createdAt: { type: Date, default: Date.now }

});

module.exports = mongoose.model('Question', questionSchema);

#### **Step 4 – Create Controller Functions**

In controllers/questionController.js:

const Question = require('../models/Question');

// Create a new question

exports.createQuestion = async (req, res) => {

try {

const question = await Question.create(req.body);

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

} catch (err) {

res.status(400).json({ message: err.message });

}

};

// Get all questions

exports.getQuestions = async (req, res) => {

try {

const questions = await Question.find();

res.json(questions);

} catch (err) {

res.status(500).json({ message: err.message });

}

};

// Update a question

exports.updateQuestion = async (req, res) => {

try {

const updated = await Question.findByIdAndUpdate(req.params.id, req.body, { new: true });

res.json(updated);

} catch (err) {

res.status(400).json({ message: err.message });

}

};

// Delete a question

exports.deleteQuestion = async (req, res) => {

try {

await Question.findByIdAndDelete(req.params.id);

res.json({ message: "Question deleted" });

} catch (err) {

res.status(500).json({ message: err.message });

}

};

#### **Step 5 – Define API Routes**

Create routes/questionRoutes.js:

const express = require('express');

const router = express.Router();

const { createQuestion, getQuestions, updateQuestion, deleteQuestion } = require('../controllers/questionController');

router.post('/', createQuestion); // CREATE

router.get('/', getQuestions); // READ

router.put('/:id', updateQuestion); // UPDATE

router.delete('/:id', deleteQuestion); // DELETE

module.exports = router;

#### **Step 6 – Test API Endpoints**

Start the server:

node server.js

Use **Postman** or **curl** to test:

GET http://localhost:5000/api/questions

POST http://localhost:5000/api/questions

PUT http://localhost:5000/api/questions/:id

DELETE http://localhost:5000/api/questions/:id

Example response:

{

"\_id": "671c2f1e9a1a73c2b3f2f2d7",

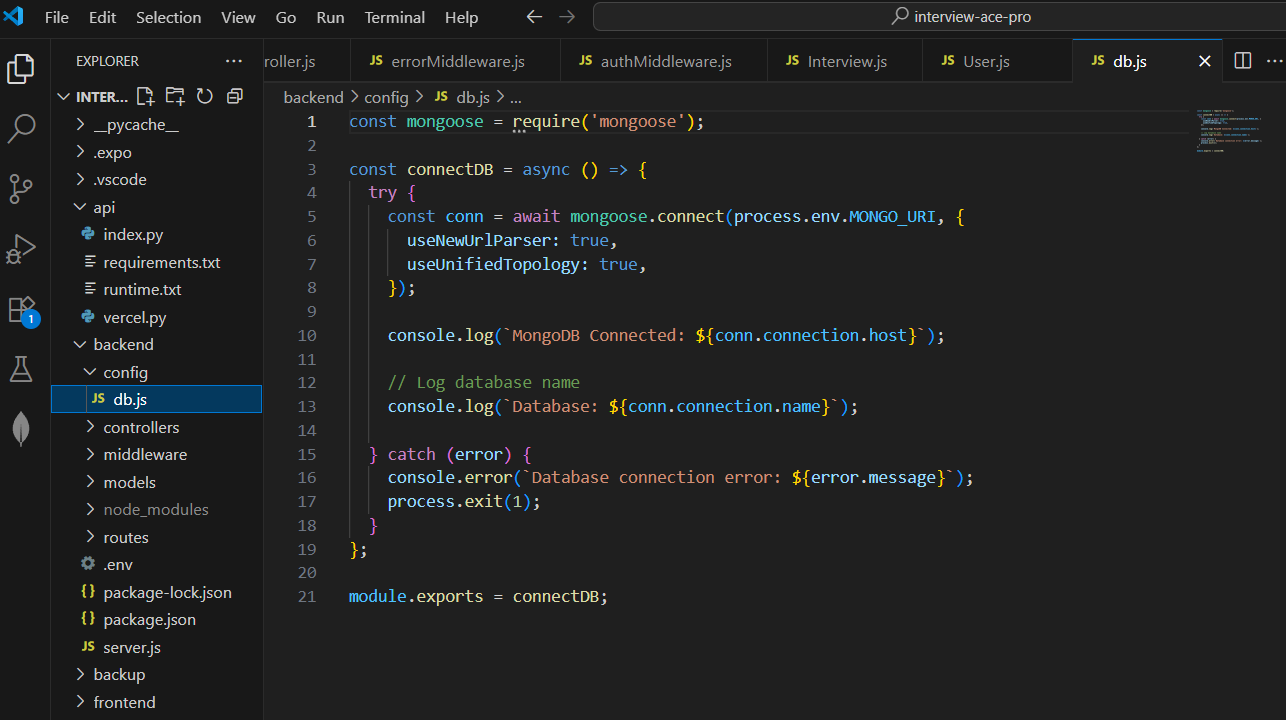
"title": "Explain closures in JavaScript.",

"difficulty": "Medium",

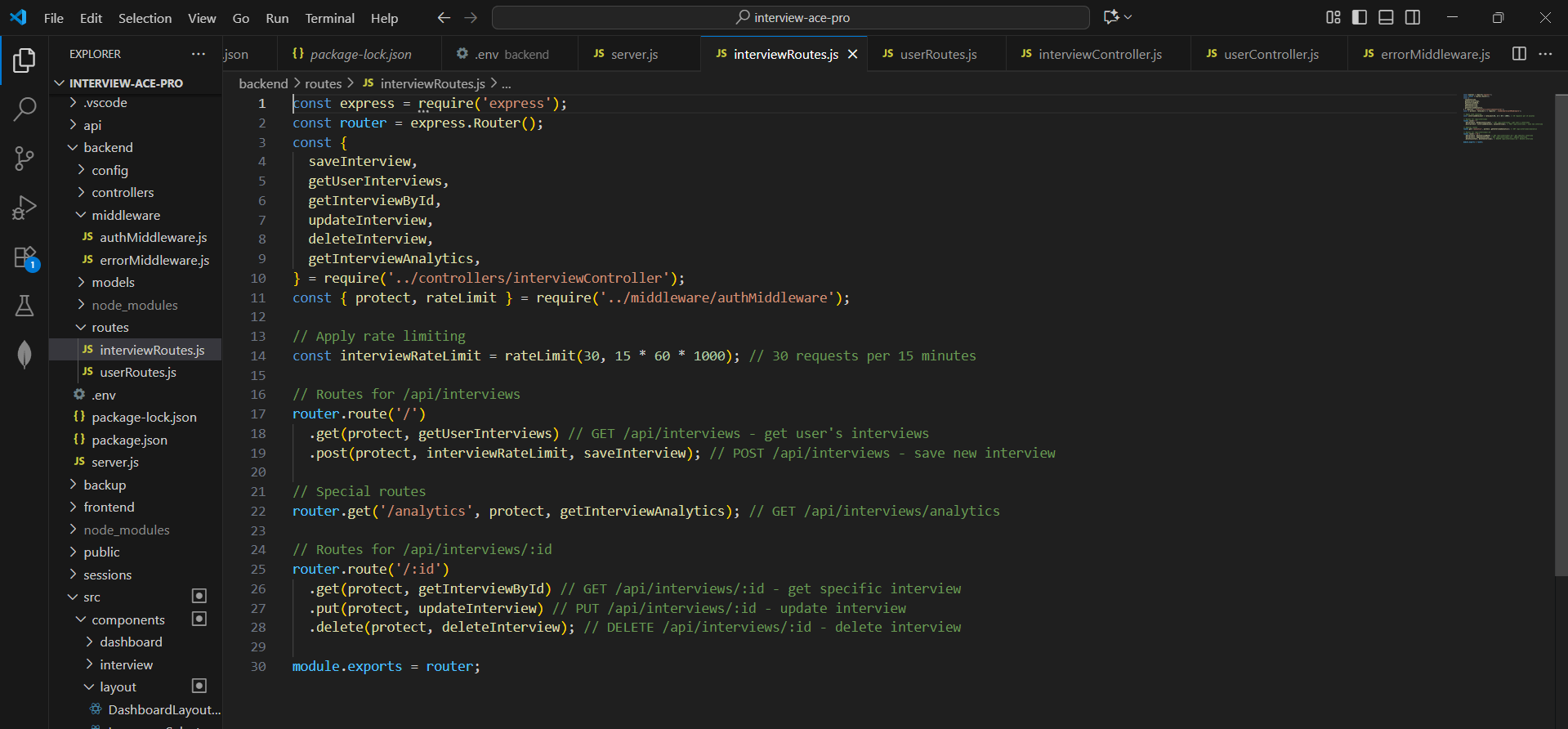
"category": "JavaScript"

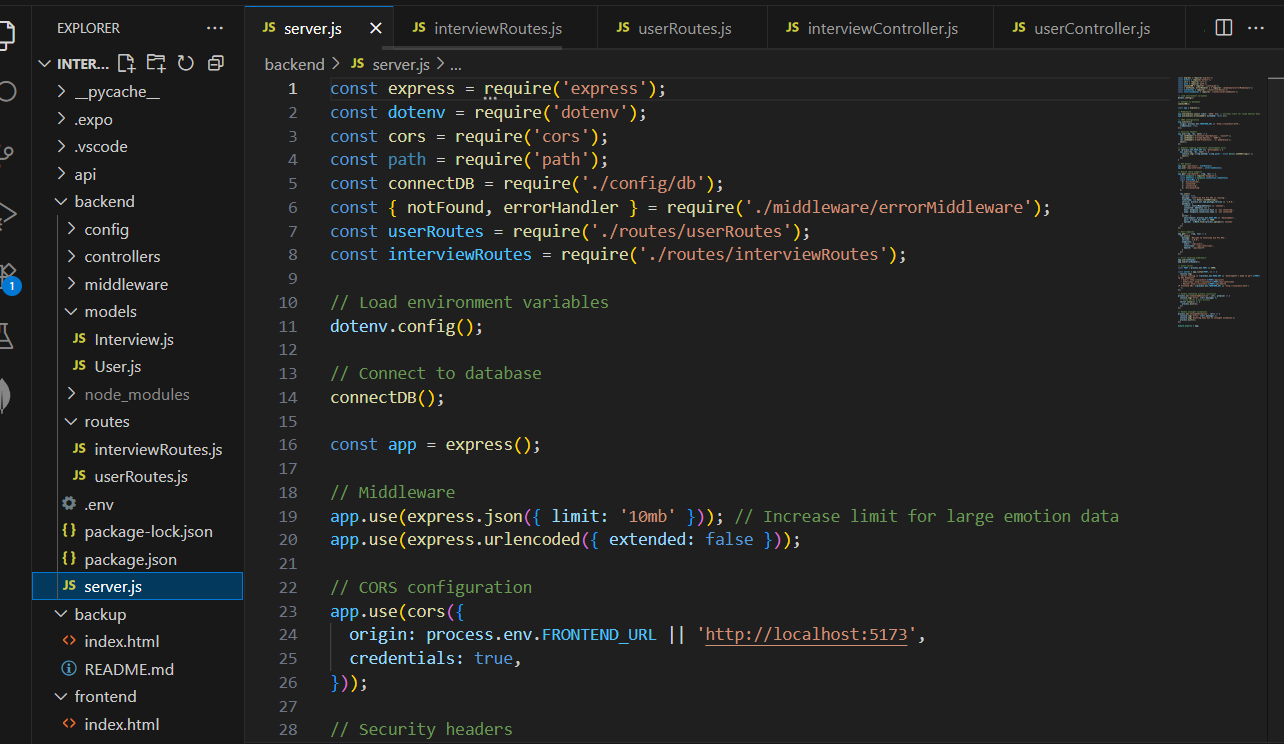
}

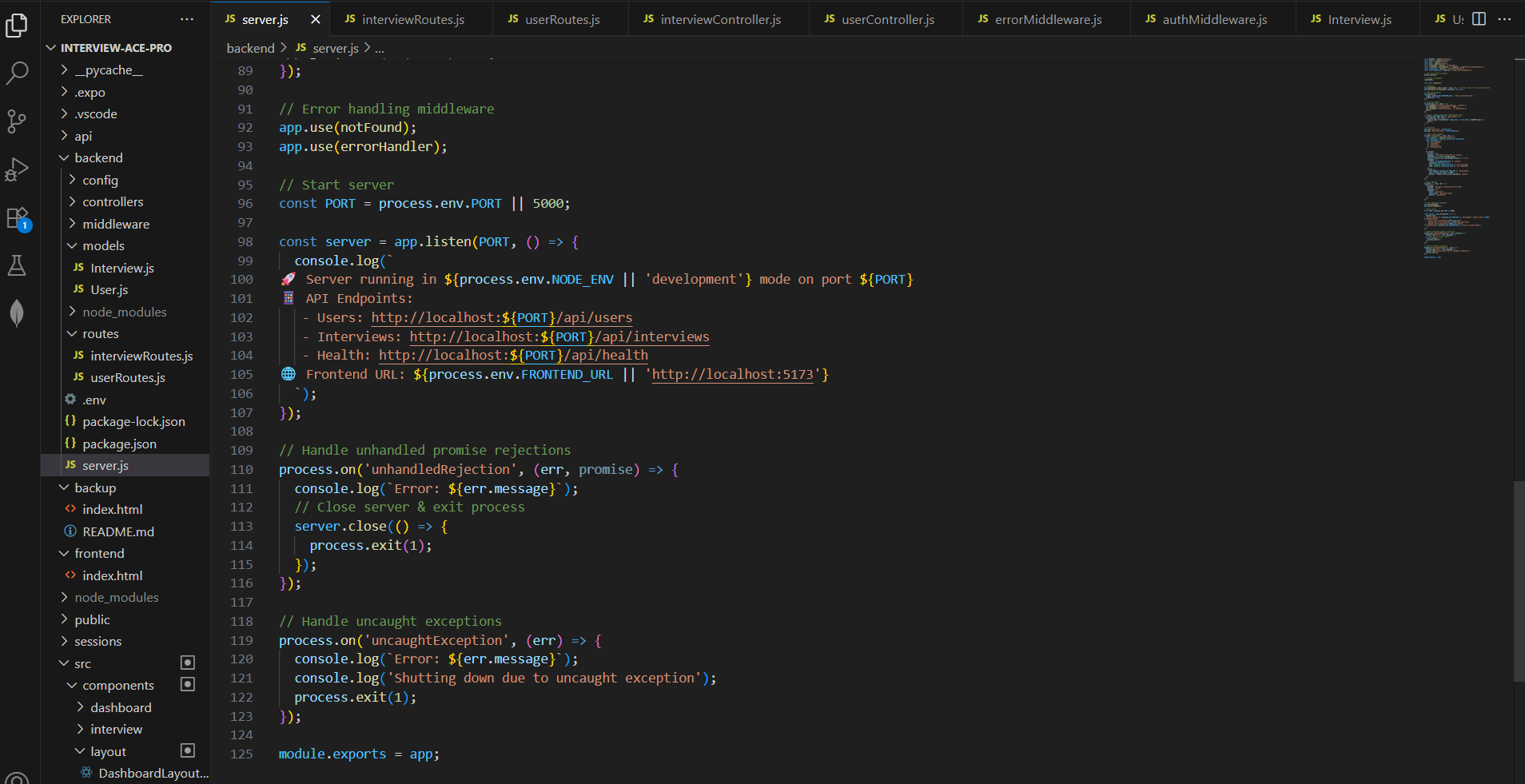
Code

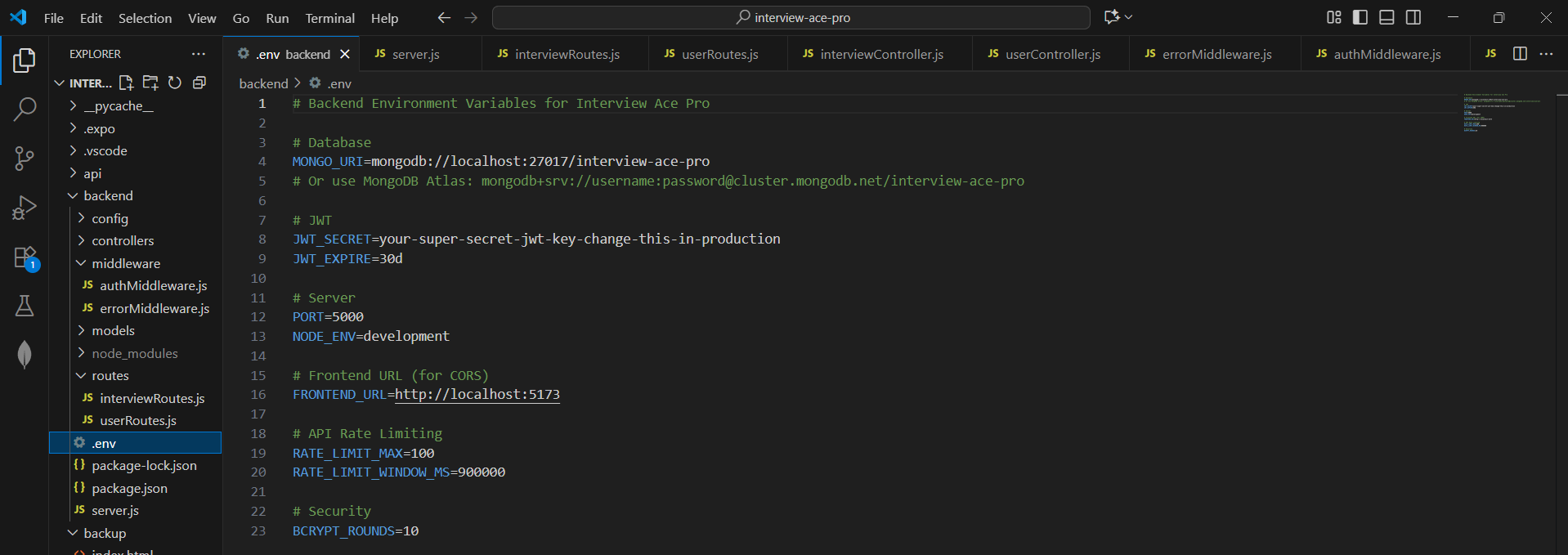


### 

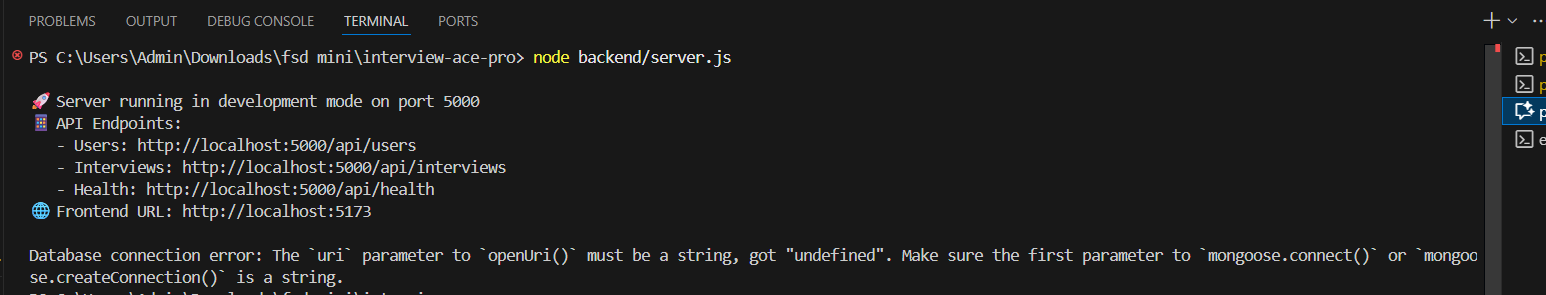


Main server .js : -

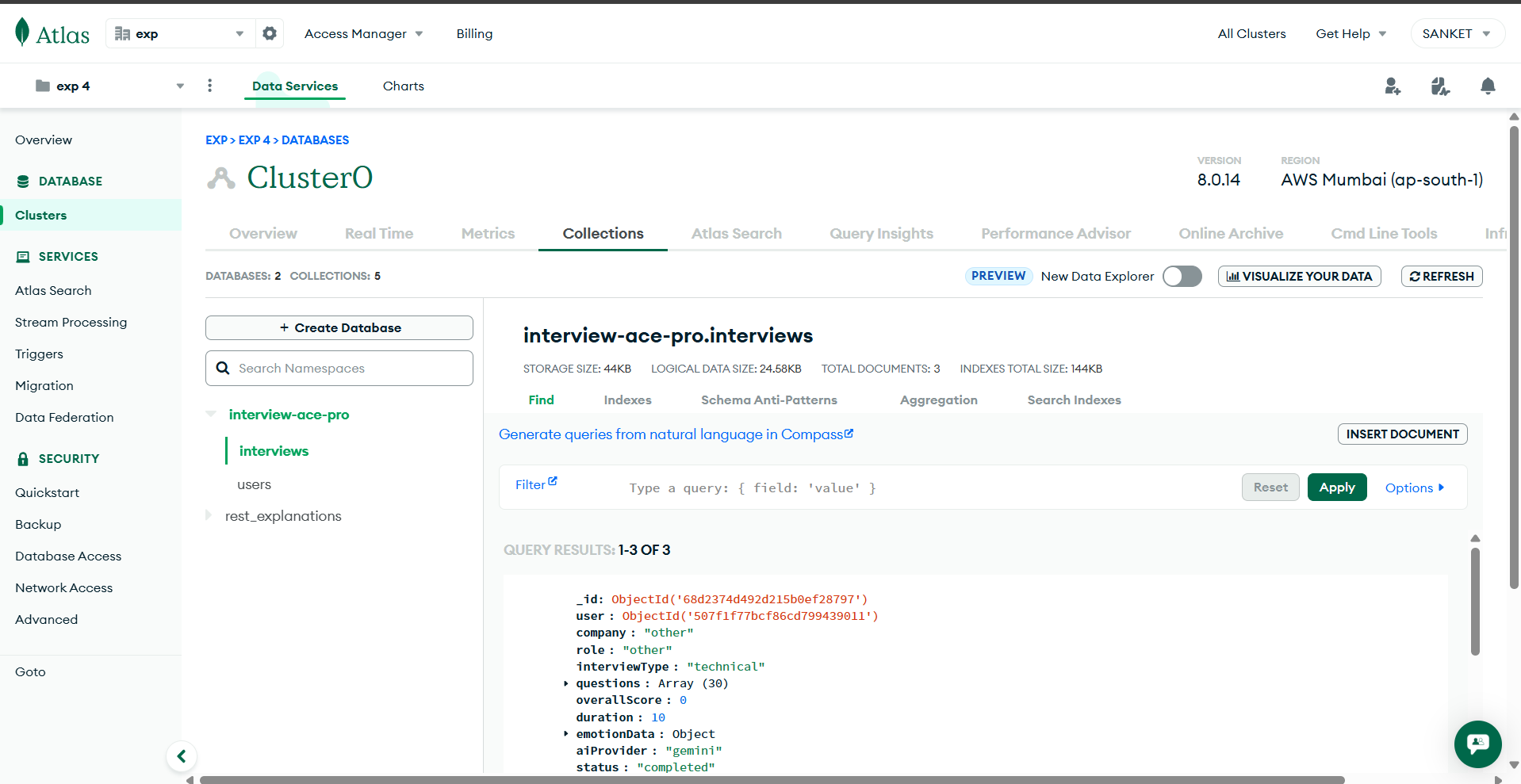


Env File

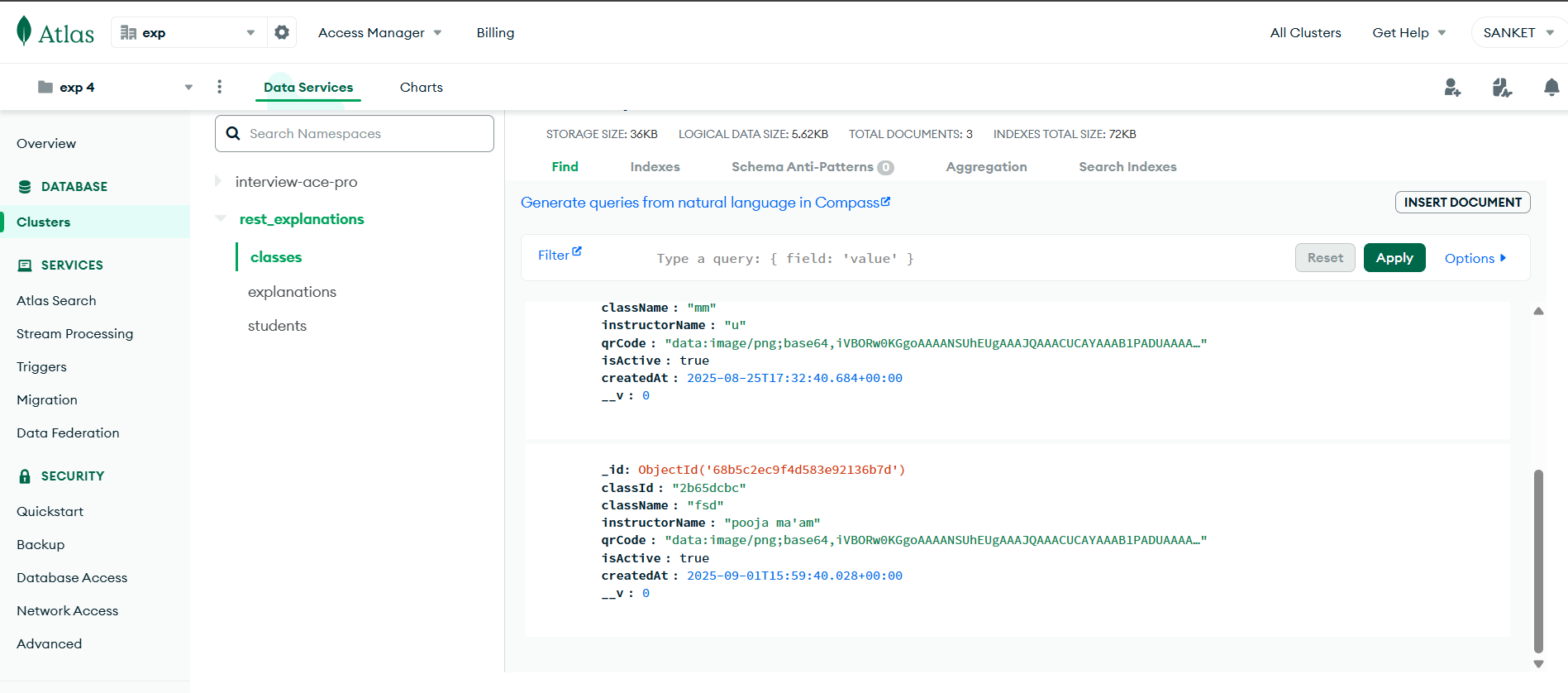
Server starting



Atlas Server



Stored database



### **Conclusion**

This experiment successfully demonstrated how to design and implement a **RESTful API** connected to **MongoDB** using **Mongoose**.  
 Through Express routes, controllers, and models, CRUD operations were performed in a structured and modular way.

In the **Interview Simulator Website**, this approach enables:

* Dynamic storage and retrieval of coding questions.
* Separation of business logic (controllers) and data models.
* Secure, scalable communication between frontend and backend.

Overall, integrating MongoDB with Mongoose provides a robust backend foundation, ensuring **data integrity**, **flexibility**, and **maintainability** for modern full-stack applications.