Experiment 4

Aim: REST API Design with MongoDB + Mongoose Integration

Theory:

A **REST API (Representational State Transfer Application Programming Interface)** is an architectural style that enables communication between client and server over HTTP. REST APIs are based on resources, which are identified by URLs, and use standard HTTP methods such as GET, POST, PUT, and DELETE to perform operations on those resources. The design of REST APIs emphasizes statelessness, meaning each request from the client must contain all the information needed to process it, ensuring scalability and simplicity.

When building REST APIs, **MongoDB** is a popular choice of database due to its flexibility and document-oriented nature. MongoDB stores data in BSON (Binary JSON) format, making it a natural fit for REST APIs where data is transmitted as JSON. This allows seamless communication between the frontend (client) and backend (server) because there is no need for complex data transformation.

To work efficiently with MongoDB in a Node.js environment, developers commonly use **Mongoose**, an Object Data Modeling (ODM) library for MongoDB. Mongoose provides a schema-based solution to model application data, allowing developers to define the structure of documents, add validation rules, and use pre-built methods for database operations. This eliminates boilerplate code and enforces data consistency.

In a REST API with MongoDB and Mongoose integration, the general flow is:

1. **Client Request:** The frontend sends an HTTP request (e.g., POST /users with a JSON body containing user data).

2. **Express Server:** The backend built with Express receives the request, applies middleware for parsing JSON and handling CORS, and routes the request to the correct controller.

3. **Mongoose Operations:** The controller uses Mongoose models to perform database operations such as creating a new user document, fetching all users, updating existing documents, or deleting them.

4. **Response:** The server sends a JSON response back to the client, confirming the action or returning the requested data.

This architecture ensures modular and maintainable code by separating concerns into **routes** (defining API endpoints), **controllers** (business logic), and **models** (database schema). Using Mongoose enforces schema validation, reducing errors and maintaining data integrity.

Overall, REST API design with MongoDB and Mongoose integration provides a robust, scalable, and efficient approach for building real-world web applications. It allows seamless interaction between client and server, supports CRUD operations, and ensures a structured and maintainable backend.

1.1 Prerequisites (Hardware & Software)

Hardware Requirements:

- A laptop/PC with at least 4 GB RAM (8 GB recommended) and a dual-core CPU
- Stable internet connection (for MongoDB Atlas or local MongoDB)

Software Requirements:

- Modern web browser (Chrome/Firefox/Edge)
- Node.js LTS version (16.x or higher)
- MongoDB (local installation) or MongoDB Atlas (cloud)
- Postman or Thunder Client for API testing
- VS Code or any preferred code editor
- NPM packages: express, mongoose, cors, dotenv, nodemon

1.2 Core Concepts Used

- **Express.js Framework**: Route handling, middleware implementation, HTTP methods (GET, POST, PUT, DELETE)
- **MongoDB**: NoSQL database for storing movie documents with flexible schema
- **Mongoose ODM**: Schema definition, model creation, data validation, and query operations
- **REST Architecture**: Resource-based URLs, proper HTTP status codes, stateless communication
- **Middleware**: CORS handling, JSON parsing, request logging, error handling
- **Environment Variables**: Database connection strings, port configuration using dotenv
- **Async/Await**: Modern JavaScript for handling asynchronous database operations

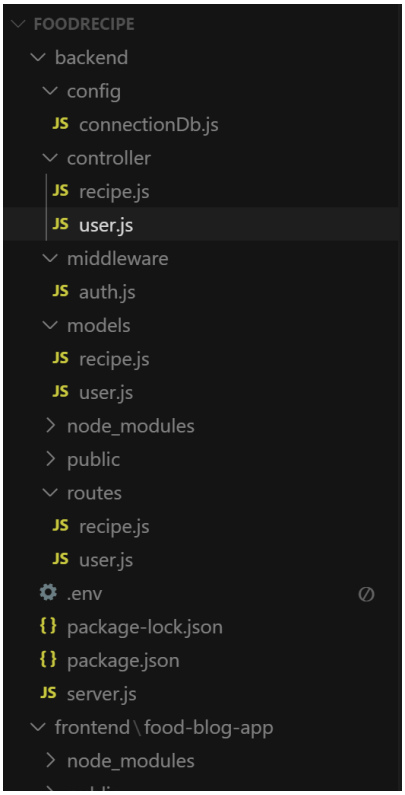
1.3 Project Workflow

1. **Setup Phase**: Initialize Node.js project → Install dependencies → Configure environment variables
2. **Database Connection**: Connect to MongoDB using Mongoose → Define movie schema and model
3. **API Routes**: Implement CRUD endpoints → Add validation and error handling
4. **Testing Phase**: Use Postman to test all endpoints → Verify data persistence in database
5. **Enhancement**: Add search functionality, pagination, and additional validation rules

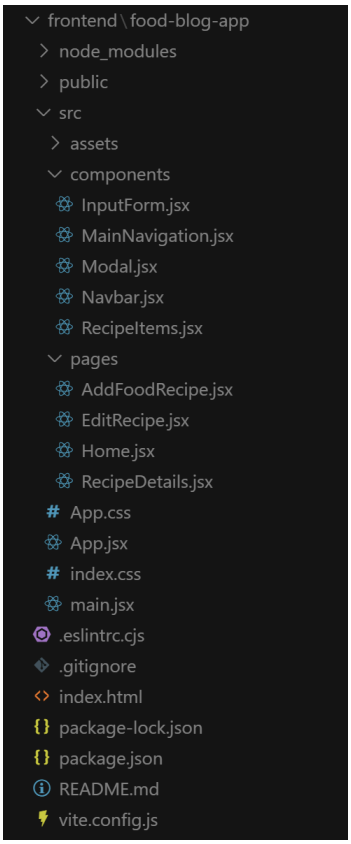
2. Code Implementation

Folder structure:

Backend



Frontend



2.1 backend/config/connectionDb.js

backend > config > connectionDb.js > <unknown>

```
1  const mongoose=require("mongoose")
2
3  const connectDb=async()=>{
4      await mongoose.connect(process.env.CONNECTION_STRING)
5      .then(()=>console.log("connected..."))
6  }
7
8  module.exports=connectDb
```

2.2 .env

backend > .env

Import to Postman

```
1  PORT=5000
2  CONNECTION_STRING="mongodb://localhost:27017/foodRecipe"
3  SECRET_KEY="TanuandVaishu"
```

2.3 models/recipe.js

backend > models > recipe.js > ...

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

const recipeSchema=mongoose.Schema({
    title:{
        type:String,
        required:true
    },
    ingredients:{
        type:Array,
        required:true
    },
    instructions:{
        type:String,
        required:true
    },
    time:{
        type:String,
    },
    coverImage:{
        type:String,
    },
    createdBy:{
        type:mongoose.Schema.Types.ObjectId,
        ref:"User"
    }

},{timestamps:true})

module.exports=mongoose.model("Recipes",recipeSchema)
```

2.4 models/user.js

backend > models > user.js > ...

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

const userSchema=mongoose.Schema({
    email:{
        type:String,
        required:true,
        unique:true
    },
    password:{
        type:String,
        required:true,
    }
},{timestamps:true})

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

2.5 routes/ recipe.js

backend > routes > recipe.js > ...

```
const express=require("express")
const { getRecipes,getRecipe,addRecipe,editRecipe,deleteRecipe,upload} = require("../controller/recipe")
const verifyToken = require("../middleware/auth")
const router=express.Router()

router.get("/",getRecipes) //Get all recipes
router.get("/:id",getRecipe) //Get recipe by id
router.post("/",upload.single('file'),verifyToken ,addRecipe) //add recipe
router.put("/:id",upload.single('file'),editRecipe) //Edit recipe
router.delete("/:id",deleteRecipe) //Delete recipe

module.exports=router
```

2.6 routes/user.js

backend > routes > user.js > <unknown>

```
const express=require("express")
const router=express.Router()
const {userLogin,userSignUp,getUser}=require("../controller/user")

router.post("/signUp",userSignUp)
router.post("/login",userLogin)
router.get("/user/:id",getUser)

module.exports=router
```

2.7 middleware/auth.js

backend > middleware > auth.js > ...

```
const jwt=require("jsonwebtoken")

const verifyToken=async(req,res,next)=>{
    let token=req.headers["authorization"]

    if(token){
        token=token.split(" ")[1]
        jwt.verify(token,process.env.SECRET_KEY,(err,decoded)=>{
            if(err){
                return res.status(400).json({message:"Invalid token"})
            }
            else{
                console.log(decoded)
                req.user=decoded
            }
        })
        next()
    }
    else{
        return res.status(400).json({message:"Invalid token"})
    }
}
module.exports=verifyToken
```

2.8 server.js

backend > JS server.js > ...

```
const express=require("express")
const app=express()
const dotenv=require("dotenv").config()
const connectDb=require("./config/connectionDb")
const cors=require("cors")

const PORT=process.env.PORT || 3000
connectDb()

app.use(express.json())
app.use(cors())
app.use(express.static("public"))

app.use("/",require("./routes/user"))
app.use("/recipe",require("./routes/recipe"))

app.listen(PORT,(err)=>{
    console.log(`app is listening on port ${PORT}`)
})
```

3. Output Screenshots

3.1 Server Startup - MongoDB Connected

Connected to MongoDB successfully
Server running on http://localhost:5000

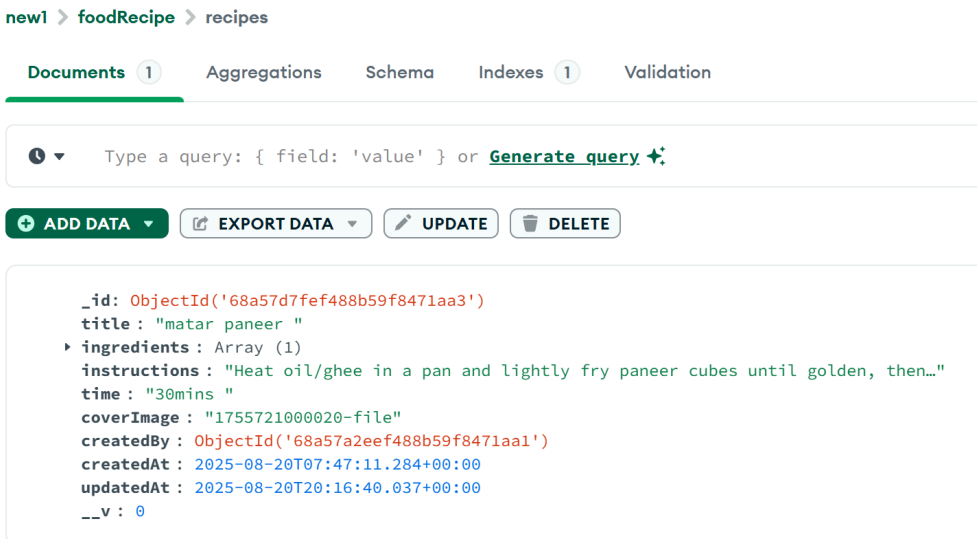
Backend

```
app is listening on port 5000
connected...
```

frontend

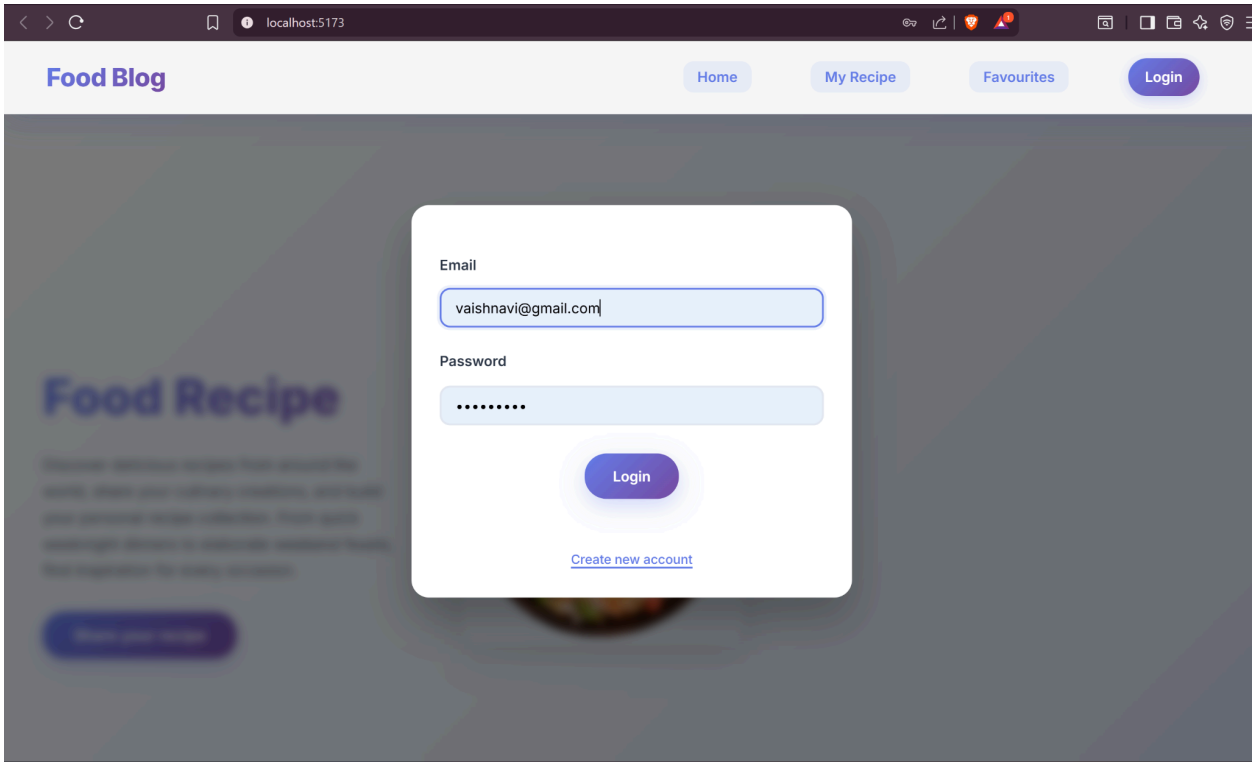
```
VITE v5.4.19  ready in 960 ms

→ Local:   http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help
```

3.2 recipes database in MongoDB

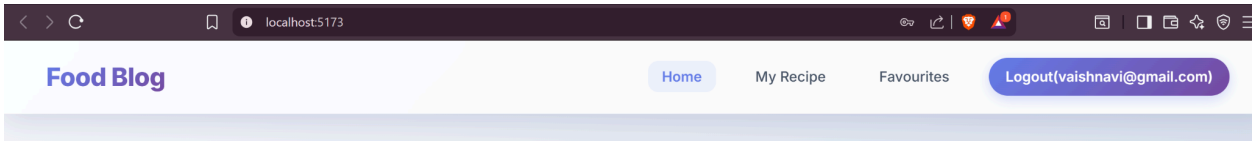


3.3 users database in MongoDB compass

3.4 Frontend- user login using the GET method

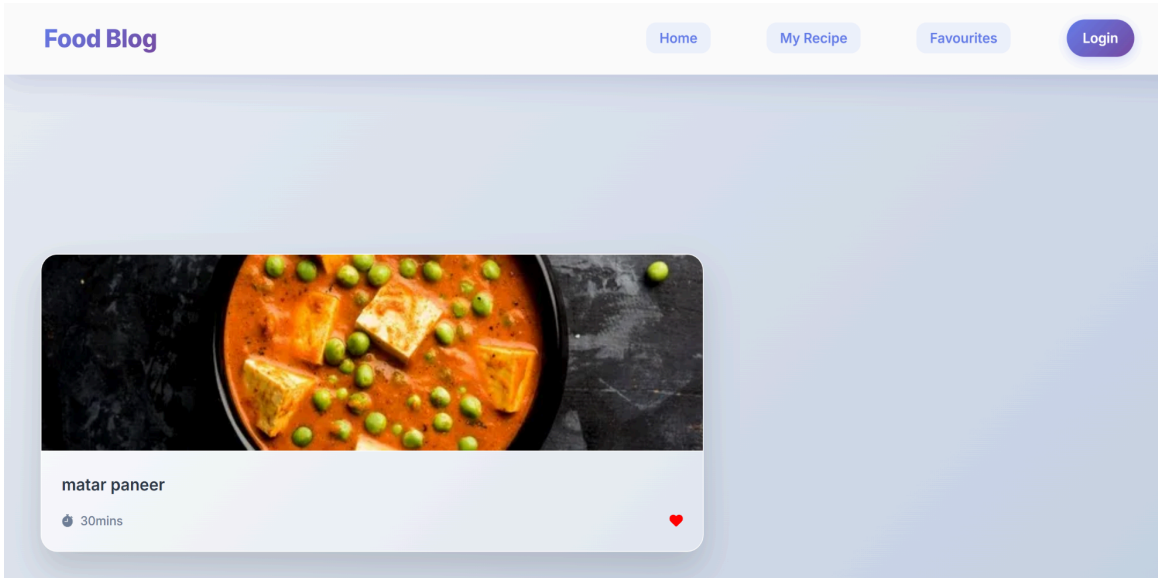


3.4.1- logging in



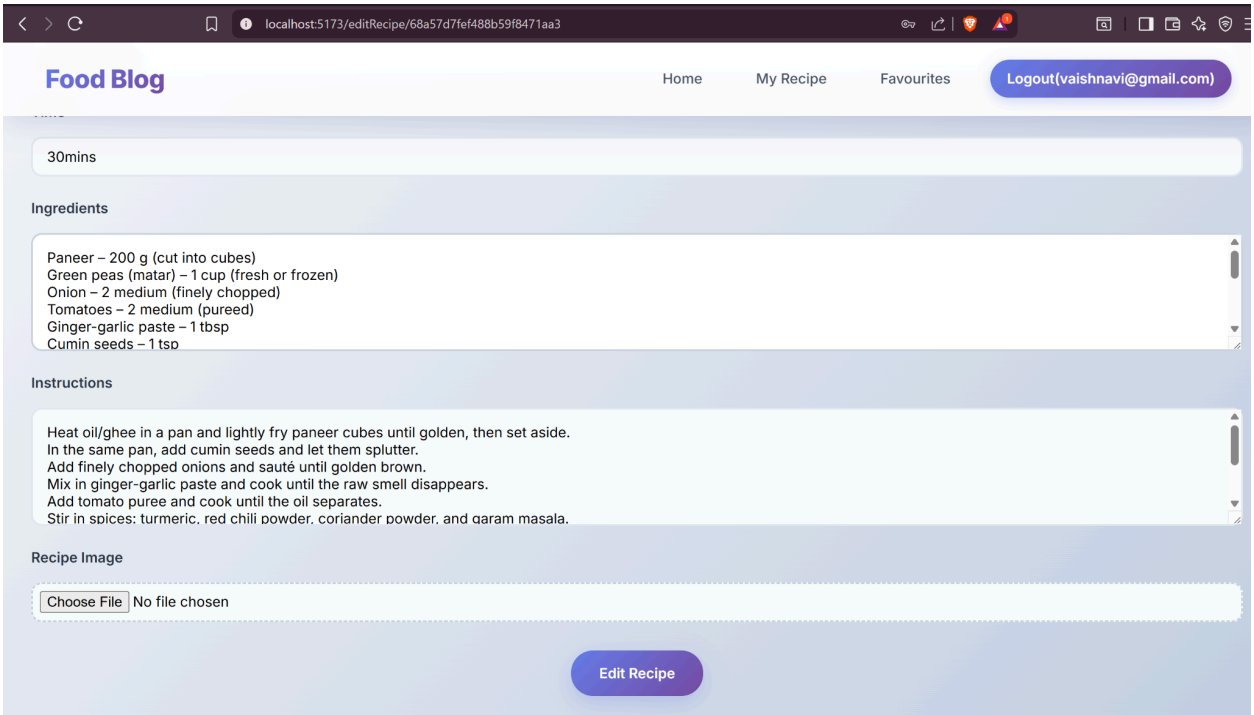
3.4.2 - logged in

3.5 Recipe stored in the database using the POST method, is seen on the frontend of the website using the GET method



3.6 favourite recipe using React usestate:

3.7 Editing the posted recipe using the PUT method



4. Extra Feature (~30%): MongoDB Atlas - the cloud database, password hashing using Bcryptjs and tokenizing the authentication using jwt.

4.1 controller/ user.js

backend > controller > recipe.js > editRecipe

```
const Recipes=require("../models/recipe")
const multer  = require('multer')

const storage = multer.diskStorage({
    destination: function (req, file, cb) {
      cb(null, './public/images')
    },
    filename: function (req, file, cb) {
      const filename = Date.now() + '-' + file.fieldname
      cb(null, filename)
    }
  })

  const upload = multer({ storage: storage })

const getRecipes=async(req,res)=>{
    const recipes=await Recipes.find()
    return res.json(recipes)
}

const getRecipe=async(req,res)=>{
    const recipe=await Recipes.findById(req.params.id)
    res.json(recipe)
}

const addRecipe=async(req,res)=>{
    console.log(req.user)
    const {title,ingredients,instructions,time}=req.body

    if(!title || !ingredients || !instructions)
    {
        res.json({message:"Required fields can't be empty"})
    }

    const newRecipe=await Recipes.create({
        title,ingredients,instructions,time,coverImage:req.file.filename,
        createdBy:req.user.id
    })
   return res.json(newRecipe)
}

const editRecipe=async(req,res)=>{
    const {title,ingredients,instructions,time}=req.body
    let recipe=await Recipes.findById(req.params.id)

    try{
        if(recipe){
            let coverImage=req.file?.filename ? req.file?.filename : recipe.coverImage
            await Recipes.findByIdAndUpdate(req.params.id,{...req.body,coverImage},{new:true})
            res.json({title,ingredients,instructions,time})
        }
    }
    catch(err){
        return res.status(404).json({message:err})
    }

}
const deleteRecipe=async(req,res)=>{
    try{
        await Recipes.deleteOne({_id:req.params.id})
        res.json({status:"ok"})
    }
    catch(err){
        return res.status(400).json({message:"error"})
    }
}

module.exports={getRecipes,getRecipe,addRecipe,editRecipe,deleteRecipe,upload}
```

4.2 hashed password in mongodb database

```
email : "vaishnavi@gmail.com"
password : "$2b$10$gkBBd1voFyisEk92Eg5Ev.Wj0LaAHfS2POYmtnjUHjGAqZ1WBGGXC"
createdAt : 2025-08-20T07:33:02.686+00:00
updatedAt : 2025-08-20T07:33:02.686+00:00
```

4.3 models/item.js

models > JS Item.js > ...

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

const ItemSchema = new mongoose.Schema({
    name: { type: String, required: true },
    quantity: { type: Number, default: 1 },
    dateAdded: { type: Date, default: Date.now }
});

module.exports = mongoose.model('Item', ItemSchema);
Ctrl+L to chat, Ctrl+K to generate
```

4.4 server.js

JS server.js > ...

```
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');

const app = express();
app.use(bodyParser.json());

// MongoDB Atlas Connection
mongoose.connect(
  "mongodb+srv://tanayaitis:<db_password>@cluster0.7ckjy1y.mongodb.net/?retryWri
  {
    useNewUrlParser: true,
    useUnifiedTopology: true
  }
)
.then(() => console.log("✅ MongoDB Atlas Connected"))
.catch(err => console.error("❌ Connection Error:", err));

// Routes
const itemRoutes = require('./routes/items');
app.use('/api/items', itemRoutes);

// Start server
app.listen(5000, () => console.log('🚀 Server running on port 5000'));
```

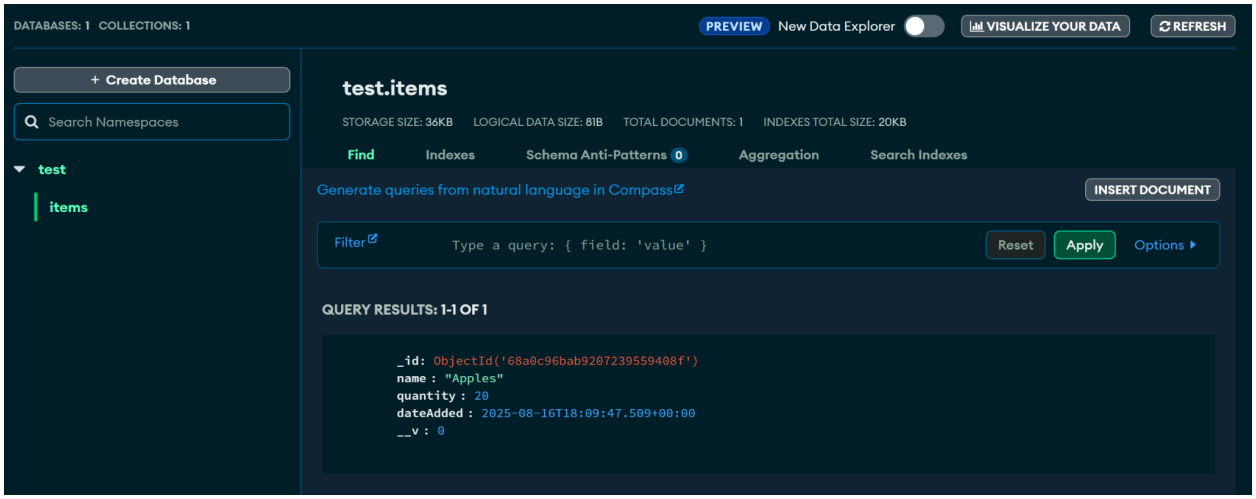
4.5 routes/items.js

routes > JS items.js > ...

```
const express = require('express');
const router = express.Router();
const Item = require('../models/Item');

// CREATE
router.post('/', async (req, res) => {
    try {
        const newItem = new Item(req.body);
        const savedItem = await newItem.save();
        res.status(201).json(savedItem);
    } catch (err) {
        res.status(400).json({ message: err.message });
    }
});
// READ all
router.get('/', async (req, res) => {
    try {
        const items = await Item.find();
        res.json(items);
    } catch (err) {
        res.status(500).json({ message: err.message });
    }
});
// READ one
router.get('/:id', async (req, res) => {
    try {
        const item = await Item.findById(req.params.id);
        res.json(item);
    } catch (err) {
        res.status(404).json({ message: 'Item not found' });
    }
});
// UPDATE
router.put('/:id', async (req, res) => {
    try {
        const updatedItem = await Item.findByIdAndUpdate(
            req.params.id,
            req.body,
            { new: true }
        );
        res.json(updatedItem);
    } catch (err) {
        res.status(400).json({ message: err.message });
    }
});
// DELETE
router.delete('/:id', async (req, res) => {
    try {
        await Item.findByIdAndDelete(req.params.id);
        res.json({ message: 'Item deleted' });
    } catch (err) {
        res.status(404).json({ message: 'Item not found' });
    }
});

module.exports = router;
```

Ctrl+I to chat, Ctrl+K to generate

4.6 MongoDB atlas database for storing items data for billing



5. Conclusion

This project demonstrates comprehensive REST API development using modern Node.js ecosystem tools. The implementation showcases proper API design patterns including resource-based routing, HTTP status codes, error handling, and data validation. The MongoDB integration with Mongoose provides robust data modeling with schema validation and query optimization. The search and filtering features illustrate advanced database querying techniques while maintaining API performance. The project structure is production-ready and follows industry best practices for scalable backend development, making it suitable for educational purposes and real-world applications.

Key Learning Outcomes:

- RESTful API architecture and design principles
- MongoDB document database operations and indexing
- Mongoose ODM for schema definition and validation
- Express.js middleware and route handling
- Error handling and API response standardization
- Database query optimization and pagination
- API testing methodologies using Postman