

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

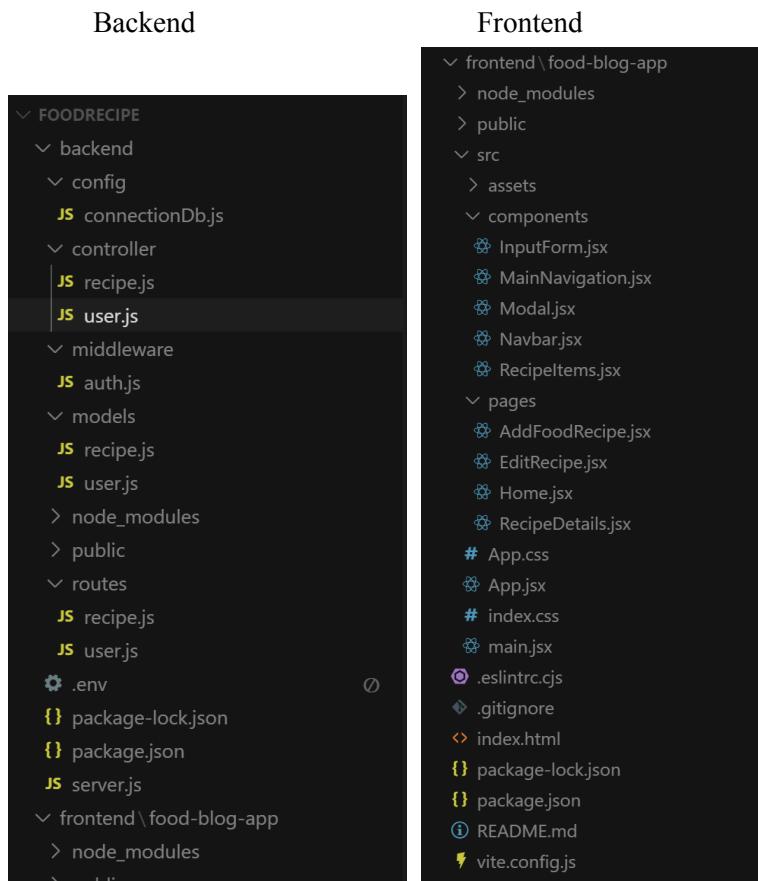
- **Express.js Framework:** Route handling, middleware implementation, HTTP methods (GET, POST, PUT, DELETE)
- **MongoDB:** NoSQL database for storing movie documents with flexible schema
- **Mongoose ODM:** Schema definition, model creation, data validation, and query operations
- **REST Architecture:** 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:



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 > JS recipe.js > ...
1  const mongoose=require("mongoose")
2
3  const recipeSchema=mongoose.Schema({
4      title:{
5          type:String,
6          required:true
7      },
8      ingredients:{
9          type:Array,
10         required:true
11     },
12     instructions:{
13         type:String,
14         required:true
15     },
16     time:{
17         type:String,
18     },
19     coverImage:{
20         type:String,
21     },
22     createdBy:{
23         type:mongoose.Schema.Types.ObjectId,
24         ref:"User"
25     }
26
27 },{timestamps:true})
28
29 module.exports=mongoose.model("Recipes",recipeSchema)
```

2.4 models/user.js

```
backend > models > JS user.js > ...
1  const mongoose=require("mongoose")
2
3  const userSchema=mongoose.Schema({
4      email:{
5          type:String,
6          required:true,
7          unique:true
8      },
9      password:{
10         type:String,
11         required:true,
12     }
13 },{timestamps:true})
14
15 module.exports=mongoose.model("User",userSchema)
```

2.5 routes/ recipe.js

```
backend > routes > JS recipe.js > ...
1 const express=require("express")
2 const { getRecipes,getRecipe,addRecipe,editRecipe,deleteRecipe,upload} = require("../controller/recipe")
3 const verifyToken = require("../middleware/auth")
4 const router=express.Router()
5
6 router.get("/",getRecipes) //Get all recipes
7 router.get("/:id",getRecipe) //Get recipe by id
8 router.post("/",upload.single('file'),verifyToken ,addRecipe) //add recipe
9 router.put("/:id",upload.single('file'),editRecipe) //Edit recipe
10 router.delete("/:id",deleteRecipe) //Delete recipe
11
12 module.exports=router
```

2.6 routes/user.js

```
backend > routes > JS user.js > [?] <unknown>
1 const express=require("express")
2 const router=express.Router()
3 const {userLogin,userSignUp,getUser}=require("../controller/user")
4
5 router.post("/signUp",userSignUp)
6 router.post("/login",userLogin)
7 router.get("/user/:id",getUser)
8
9 module.exports=router
```

2.7 middleware/auth.js

```
backend > middleware > JS auth.js > ...
1 const jwt=require("jsonwebtoken")
2
3 const verifyToken=async(req,res,next)=>{
4     let token=req.headers["authorization"]
5
6     if(token){
7         token=token.split(" ")[1]
8         jwt.verify(token,process.env.SECRET_KEY,(err,decoded)=>{
9             if(err){
10                 return res.status(400).json({message:"Invalid token"})
11             }
12             else{
13                 console.log(decoded)
14                 req.user=decoded
15             }
16         })
17         next()
18     }
19     else{
20         return res.status(400).json({message:"Invalid token"})
21     }
22 }
23 module.exports=verifyToken
```

2.8 server.js

```
backend > js server.js > ...
1 const express=require("express")
2 const app=express()
3 const dotenv=require("dotenv").config()
4 const connectDb=require("./config/connectionDb")
5 const cors=require("cors")
6
7 const PORT=process.env.PORT || 3000
8 connectDb()
9
10 app.use(express.json())
11 app.use(cors())
12 app.use(express.static("public"))
13
14 app.use("/",require("./routes/user"))
15 app.use("/recipe",require("./routes/recipe"))
16
17 app.listen(PORT,(err)=>{
18   |   console.log(`app is listening on port ${PORT}`)
19 })
```

3. Output Screenshots

3.1 Server Startup - MongoDB Connected

Connected to MongoDB successfully

Server running on http://localhost:5000

Backend

```
[node:0] Starting node server.js
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

new1 > foodRecipe > recipes

Documents 1 Aggregations Schema Indexes 1 Validation

Type a query: { field: 'value' } or [Generate query](#)

[ADD DATA](#) [EXPORT DATA](#) [UPDATE](#) [DELETE](#)

```
_id: ObjectId('68a57d7fef488b59f8471aa3')
title: "matar paneer"
ingredients: Array (1)
instructions: "Heat oil/ghee in a pan and lightly fry paneer cubes until golden, then..."
time: "30mins"
coverImage: "1755721000020-file"
createdBy: ObjectId('68a57a2eef488b59f8471aa1')
createdAt: 2025-08-20T07:47:11.284+00:00
updatedAt: 2025-08-20T20:16:40.037+00:00
__v: 0
```

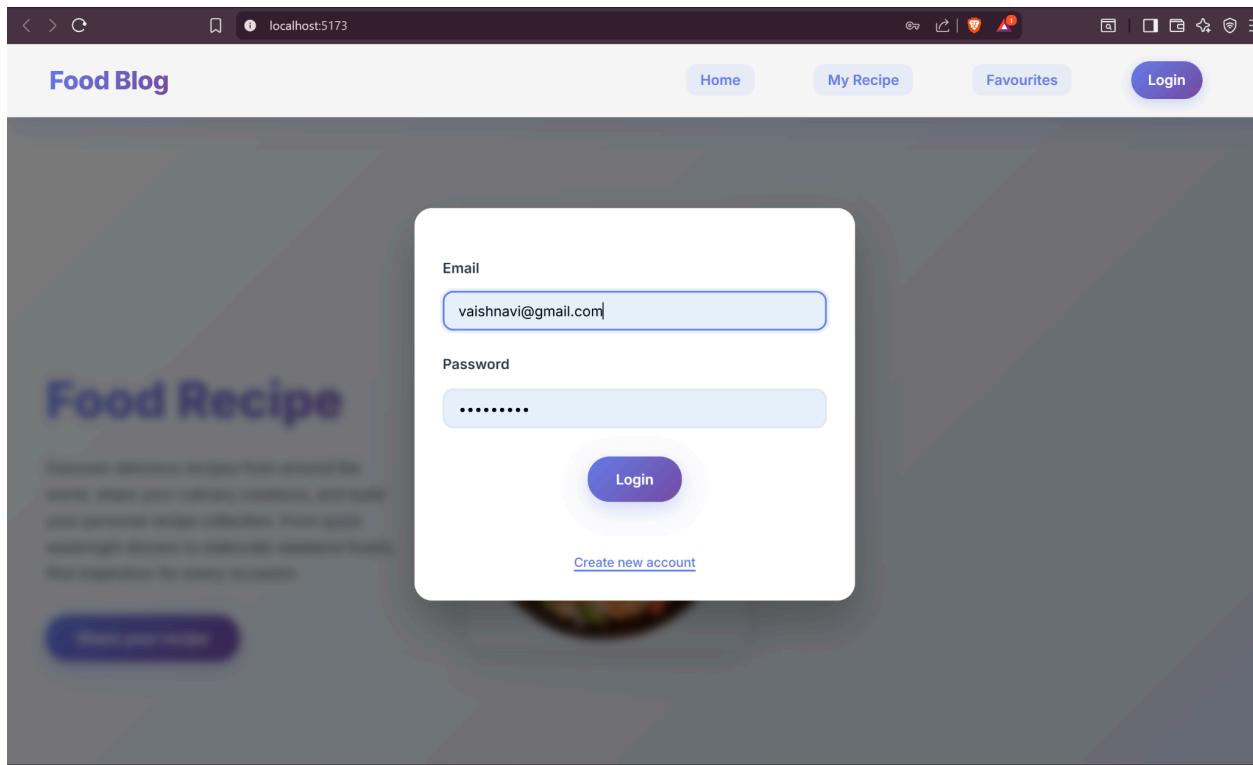
3.3 users database in MongoDB compass

```
_id: ObjectId('68a578d4ef488b59f8471a96')
email: "tanaya@gmail.com"
password: "$2b$10$NL/i5vsAoIvgsKTqkacTcewBnTIzg.vMgMxf5p0qK6f./wZqesLC6"
createdAt: 2025-08-20T07:27:16.919+00:00
updatedAt: 2025-08-20T07:27:16.919+00:00
__v: 0
```

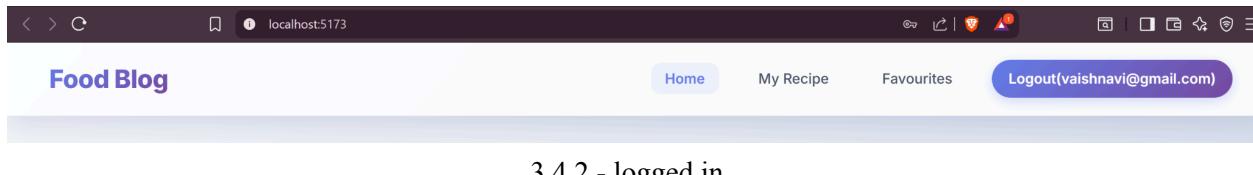
```
_id: ObjectId('68a57a2eef488b59f8471aa1')
email: "vaishnavi@gmail.com"
password: "$2b$10$gkBbd1voFyisEk92Eg5Ev.Wj0LaAHfS2POYmtnjUHjGAqZ1WBGGXC"
createdAt: 2025-08-20T07:33:02.686+00:00
updatedAt: 2025-08-20T07:33:02.686+00:00
__v: 0
```

```
_id: ObjectId('68a61e63829bfadec68ef2df')
email: "newuser@gmail.com"
password: "$2b$10$yfHA3TdBMorxEfLjjGO/K.WftYCvCtay8RNicgX3c0Uqg/Jrf89Du"
createdAt: 2025-08-20T19:13:39.818+00:00
updatedAt: 2025-08-20T19:13:39.818+00:00
__v: 0
```

3.4 Frontend- user login using the GET method

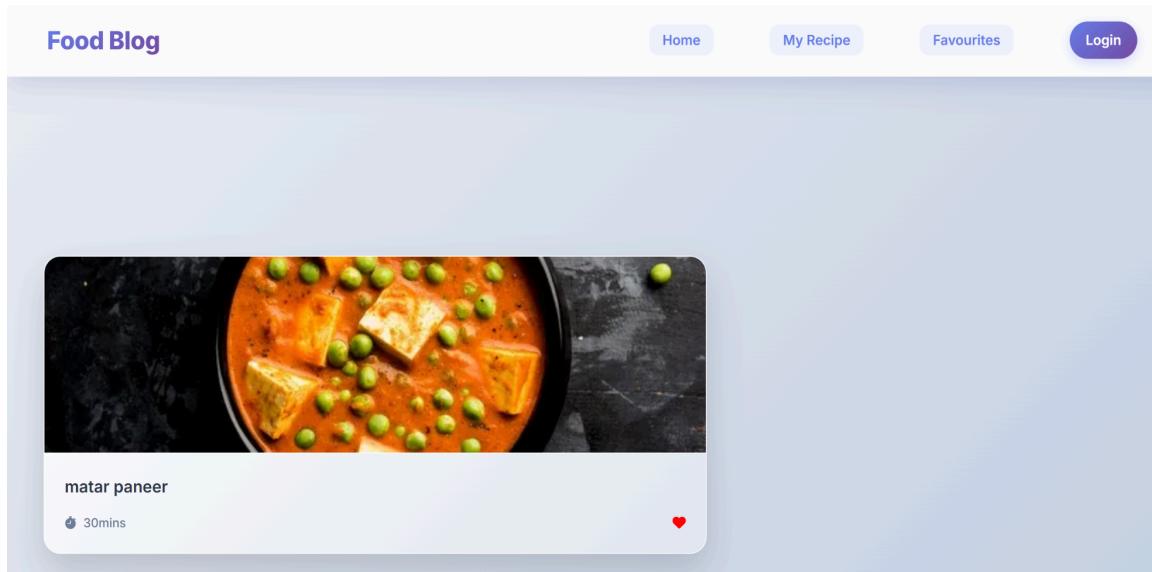


3.4.1- logging in



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

A screenshot of a web browser window showing the "Food Blog" application's edit recipe page. The URL in the address bar is "localhost:5173/editRecipe/68a57d7fe488b59f8471aa3". The page has a header with "Food Blog", "Home", "My Recipe", "Favourites", and "Logout(vaishnavi@gmail.com)". The main content area shows a recipe for "matar paneer". It includes a "30mins" cooking time, a "Ingredients" section with a list of ingredients, and an "Instructions" section with a detailed cooking process. There is also a "Recipe Image" field with a file input and a "Edit Recipe" button at the bottom.

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 > js recipe.js > [e] editRecipe
  1 const Recipes=require("../models/recipe")
  2 const multer = require('multer')
  3
  4 const storage = multer.diskStorage({
  5   destination: function (req, file, cb) {
  6     cb(null, './public/images')
  7   },
  8   filename: function (req, file, cb) {
  9     const filename = Date.now() + '-' + file.fieldname
 10     cb(null, filename)
 11   }
 12 })
 13
 14 const upload = multer({ storage: storage })
 15
 16 const getRecipes=async(req,res)=>{
 17   const recipes=await Recipes.find()
 18   return res.json(recipes)
 19 }
 20
 21 const getRecipe=async(req,res)=>{
 22   const recipe=await Recipes.findById(req.params.id)
 23   res.json(recipe)
 24 }
 25
 26 const addRecipe=async(req,res)=>{
 27   console.log(req.user)
 28   const {title,ingredients,instructions,time}=req.body
 29
 30   if(!title || !ingredients || !instructions)
 31   {
 32     res.json({message:"Required fields can't be empty"})
 33   }
 34
 35   const newRecipe=await Recipes.create({
 36     title,ingredients,instructions,time,coverImage:req.file.filename,
 37     createdBy:req.user.id
 38   })
 39   return res.json(newRecipe)
 40 }
 41
 42 const editRecipe=async(req,res)=>{
 43   const {title,ingredients,instructions,time}=req.body
 44   let recipe=await Recipes.findById(req.params.id)
 45
 46   try{
 47     if(recipe){
 48       let coverImage=req.file?.filename ? req.file?.filename : recipe.coverImage
 49       await Recipes.findByIdAndUpdate(req.params.id,...req.body,coverImage,{new:true})
 50       res.json({title,ingredients,instructions,time})
 51     }
 52   }
 53   catch(err){
 54     return res.status(404).json({message:err})
 55   }
 56 }
 57
 58 const deleteRecipe=async(req,res)=>{
 59   try{
 60     await Recipes.deleteOne({_id:req.params.id})
 61     res.json({status:"ok"})
 62   }
 63   catch(err){
 64     return res.status(400).json({message:"error"})
 65   }
 66 }
 67
 68 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 > ...
1 const mongoose = require('mongoose');
2
3 const ItemSchema = new mongoose.Schema({
4   name: { type: String, required: true },
5   quantity: { type: Number, default: 1 },
6   dateAdded: { type: Date, default: Date.now }
7 });
8
9 module.exports = mongoose.model('Item', ItemSchema);
10 | Ctrl+L to chat, Ctrl+K to generate
```

4.4 server.js

```
JS server.js > ...
1 const express = require('express');
2 const mongoose = require('mongoose');
3 const bodyParser = require('body-parser');
4
5 const app = express();
6 app.use(bodyParser.json());
7
8 // MongoDB Atlas Connection
9 mongoose.connect(
10   "mongodb+srv://tanayaitis:<db_password>@cluster0.7ckjy1y.mongodb.net/?retryWri
11   {
12     useNewUrlParser: true,
13     useUnifiedTopology: true
14   }
15 )
16 .then(() => console.log("✅ MongoDB Atlas Connected"))
17 .catch(err => console.error("✗ Connection Error:", err));
18
19 // Routes
20 const itemRoutes = require('./routes/items');
21 app.use('/api/items', itemRoutes);
22
23 // Start server
24 app.listen(5000, () => console.log('🚀 Server running on port 5000'));
```

4.5 routes/items.js

```
routes > js items.js > ...
1  const express = require('express');
2  const router = express.Router();
3  const Item = require('../models/Item');
4
5  // CREATE
6  router.post('/', async (req, res) => {
7      try {
8          const newItem = new Item(req.body);
9          const savedItem = await newItem.save();
10         res.status(201).json(savedItem);
11     } catch (err) {
12         res.status(400).json({ message: err.message });
13     }
14 });
15 // READ all
16 router.get('/', async (req, res) => {
17     try {
18         const items = await Item.find();
19         res.json(items);
20     } catch (err) {
21         res.status(500).json({ message: err.message });
22     }
23 });
24 // READ one
25 router.get('/:id', async (req, res) => {
26     try {
27         const item = await Item.findById(req.params.id);
28         res.json(item);
29     } catch (err) {
30         res.status(404).json({ message: 'Item not found' });
31     }
32 });
33 // UPDATE
34 router.put('/:id', async (req, res) => {
35     try {
36         const updatedItem = await Item.findByIdAndUpdate(
37             req.params.id,
38             req.body,
39             { new: true }
40         );
41         res.json(updatedItem);
42     } catch (err) {
43         res.status(400).json({ message: err.message });
44     }
45 });
46 // DELETE
47 router.delete('/:id', async (req, res) => {
48     try {
49         await Item.findByIdAndDelete(req.params.id);
50         res.json({ message: 'Item deleted' });
51     } catch (err) {
52         res.status(404).json({ message: 'Item not found' });
53     }
54 });
55
56 module.exports = router;
57 | Ctrl+U to chat Ctrl+K to generate
```

4.6 MongoDB atlas database for storing items data for billing

The screenshot shows the MongoDB Compass interface. On the left sidebar, under the 'test' database, the 'items' collection is selected. The main panel displays the 'test.items' collection with the following details: STORAGE SIZE: 36KB, LOGICAL DATA SIZE: 81B, TOTAL DOCUMENTS: 1, INDEXES TOTAL SIZE: 20KB. Below these details are tabs for 'Find', 'Indexes', 'Schema Anti-Patterns' (0), 'Aggregation', and 'Search Indexes'. A search bar at the top says 'Generate queries from natural language in Compass'. At the bottom of the main panel, there is a 'QUERY RESULTS: 1-1 OF 1' section containing a single document:

```
_id: ObjectId('68a0c96bab9207239559408f')
name : "Apples"
quantity : 20
dateAdded : 2025-08-16T18:09:47.509+00:00
__v : 0
```

Below the results are buttons for 'Filter', 'Type a query: { field: 'value' }', 'Reset', 'Apply', and 'Options'.

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