**Web X.0 Lab**

**Experiment - 4**

Name: Shashwat Tripathi

Div: D15A

Roll no: 64

**Aim:** To study CRUD operations in MongoDB

**Theory:**

1. **Describe some of the features of MongoDB?**

MongoDB is a popular NoSQL database known for its flexibility and scalability. Some key features include:

Document-Oriented: MongoDB stores data in flexible, JSON-like documents, making it easy to model complex hierarchical data structures.

Schema-less: Unlike traditional relational databases, MongoDB doesn't require a predefined schema, allowing for dynamic and evolving data models.

Highly Scalable: MongoDB can scale horizontally across multiple servers using sharding, enabling it to handle large volumes of data and high throughput applications.

High Performance: With features like indexing, aggregation pipelines, and in-memory caching, MongoDB offers high performance for both read and write operations.

Flexible Query Language: MongoDB provides a powerful query language that supports rich document queries, text search, geospatial queries, and more.

Automatic Failover: MongoDB provides built-in support for automatic failover and replica sets, ensuring high availability and data redundancy.

Community and Enterprise Support: MongoDB offers both a free community edition and a commercial enterprise edition with additional features, support, and services.

1. **What are Documents and Collections in MongoDB?**

Documents: In MongoDB, a document is a JSON-like data structure that stores data in key-value pairs. Documents represent individual records in a collection and can contain nested fields and arrays, allowing for flexible data modeling.

Collections: A collection is a grouping of MongoDB documents. Collections are analogous to tables in relational databases, but unlike tables, collections don't enforce a schema. Collections can contain documents with different structures, making them highly flexible.

1. **When to use MongoDB?**

MongoDB is suitable for a wide range of use cases, including:

Scalable Web Applications: MongoDB's horizontal scalability and flexible data model make it well-suited for web applications with rapidly changing data requirements and high scalability needs.

Real-time Analytics: MongoDB's ability to handle large volumes of data and perform complex queries makes it suitable for real-time analytics and reporting applications.

Content Management Systems: MongoDB's document-oriented model is ideal for content management systems, allowing for easy storage and retrieval of various types of content.

Mobile and IoT Applications: MongoDB's support for mobile and IoT platforms, as well as its ability to synchronize data between devices and servers, makes it a good choice for mobile and IoT applications.

Prototyping and Agile Development: MongoDB's schema-less design and flexible data model make it well-suited for prototyping and agile development, where requirements may change frequently.

1. **What is Sharding in MongoDB?MongoDB can route queries to other available shards.**

Sharding is a technique used in MongoDB to horizontally partition data across multiple servers or shards. This allows MongoDB to distribute data and queries across a cluster of machines, enabling horizontal scalability and improved performance.

How it works: In MongoDB, sharding involves dividing a collection into smaller chunks called shards and distributing these shards across multiple servers in a cluster. Each shard contains a subset of the collection's data, and MongoDB automatically routes queries to the appropriate shards based on the shard key.

Benefits: Sharding allows MongoDB to scale out to handle large datasets and high throughput workloads. It improves read and write performance by distributing data and queries across multiple servers. Additionally, sharding provides fault tolerance and high availability by replicating data across shards using replica sets.

Routing Queries: MongoDB's query router (mongos) is responsible for routing queries to the appropriate shards based on the shard key. This allows MongoDB to distribute queries efficiently and balance the workload across the cluster.

Dynamic Scaling: MongoDB supports dynamic scaling, allowing you to add or remove shards from the cluster as needed without downtime. This makes it easy to scale MongoDB to accommodate changing data volumes and application requirements.

1. **What role does Mongoose play in building a RESTful API with MongoDB, and why is it commonly used?**

ORM for MongoDB: Mongoose is an Object-Document Mapper (ODM) for MongoDB, providing a schema-based solution for modeling application data.

Schema Definition: Mongoose allows developers to define schemas that enforce data consistency and structure, making it easier to work with MongoDB's flexible document model.

Validation: Mongoose provides built-in validation capabilities to ensure that data conforms to specified rules and constraints before it's saved to the database.

Query Building: Mongoose simplifies the process of building MongoDB queries by providing a fluent API for querying, filtering, updating, and deleting documents.

Middleware: Mongoose supports middleware functions that allow developers to execute custom logic before or after certain database operations, enabling tasks like data transformation, logging, or authentication.

Population: Mongoose supports population, allowing developers to reference documents from other collections and automatically populate them when querying data, simplifying data retrieval and aggregation.

Integration with Express: Mongoose seamlessly integrates with Express.js, a popular Node.js web framework, making it easy to build RESTful APIs with MongoDB.

Why it's commonly used: Mongoose is commonly used in building RESTful APIs with MongoDB because it provides a convenient and expressive way to interact with MongoDB databases, simplifying data modeling, validation, querying, and middleware integration. Its rich feature set and seamless integration with Express.js make it a popular choice for Node.js developers working with MongoDB.

1. **List features of REST architectural style.**

Client-Server Architecture: Separation of concerns between client and server components, allowing them to evolve independently.

Statelessness: Each request from a client to the server must contain all the information necessary to understand and process the request, promoting scalability and reliability.

Uniform Interface: Resources are identified by URIs, and standard HTTP methods (GET, POST, PUT, DELETE) are used to manipulate resources, providing a consistent interface.

Cacheability: Responses can be cached to improve performance and reduce server load.

Layered System: Components are organized in layers, allowing for scalability, encapsulation, and security.

Code on Demand (optional): Servers can send executable code to clients to extend functionality dynamically (e.g., JavaScript in web browsers).

1. **What are the advantages of using APIs?**

Modularity and Reusability: APIs allow developers to modularize their code and expose specific functionality, making it reusable across different applications and platforms.

Interoperability: APIs provide a standardized way for different software systems to communicate and interact with each other, enabling integration and interoperability between diverse technologies.

Abstraction: APIs abstract away complex implementation details, allowing developers to interact with services and data through a simplified interface without needing to understand the underlying complexity.

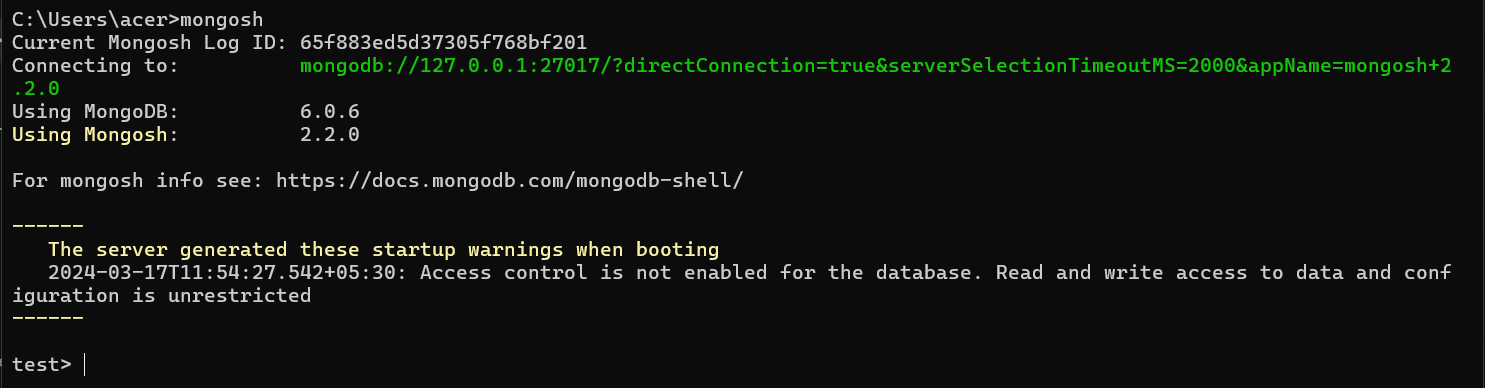
Scalability: APIs enable scalable architectures by allowing components to be distributed across different servers or microservices, facilitating horizontal scaling and improved performance.

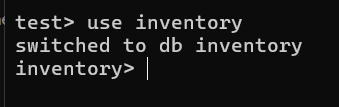
Flexibility: APIs allow for flexibility in software design and architecture, as they enable decoupling between different components, making it easier to make changes or updates without affecting other parts of the system.

Ecosystem Growth: APIs foster the growth of developer ecosystems by enabling third-party developers to build applications, extensions, or integrations that leverage existing services and platforms. This promotes innovation and collaboration within the developer community.

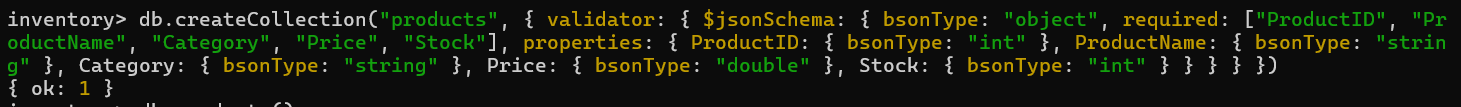
**Problem Statement:** Create a database, create a collection, insert data, query and manipulate data using various MongoDB operations.

1. Create a database named "inventory".

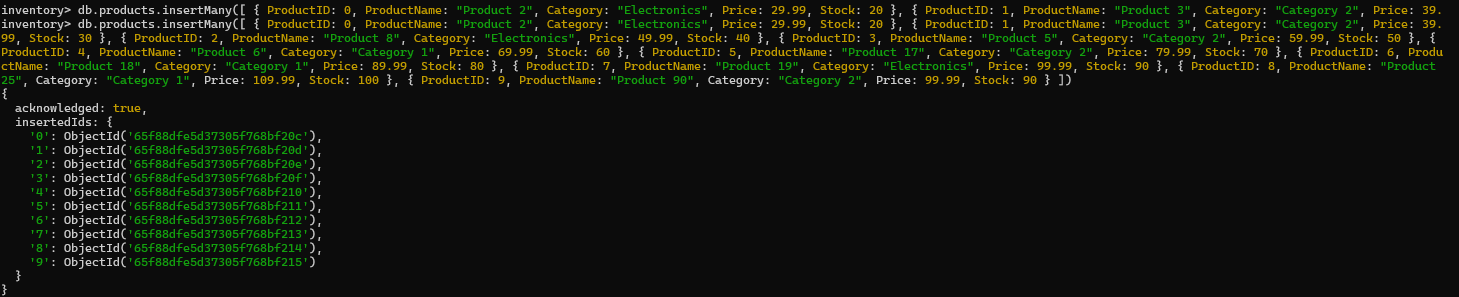




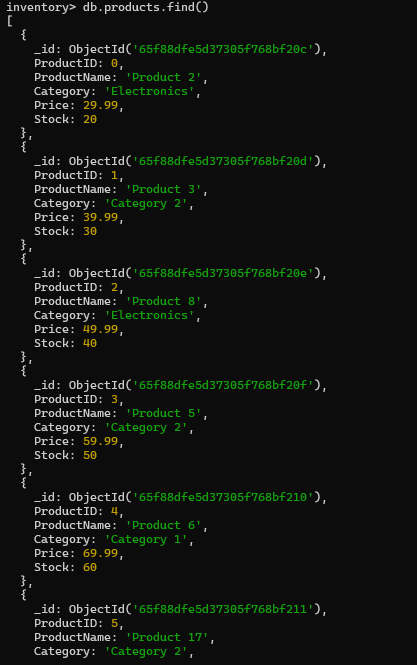
1. Create a collection named "products" with the fields: (ProductID, ProductName, Category, Price, Stock).

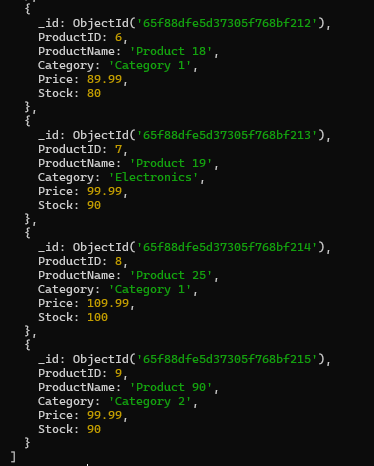


1. Insert 10 documents into the "products" collection.



1. Display all the documents in the "products" collection.

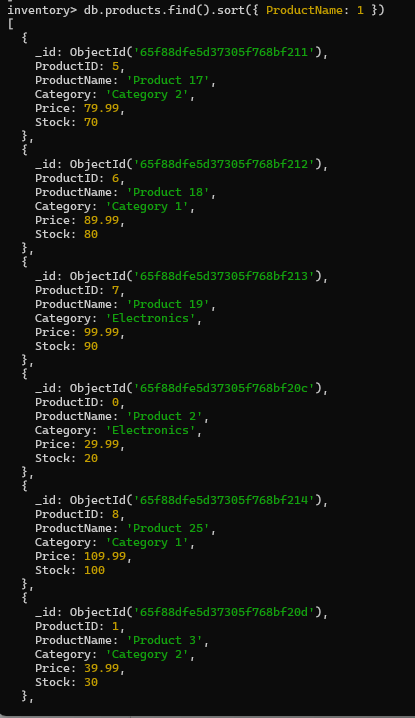


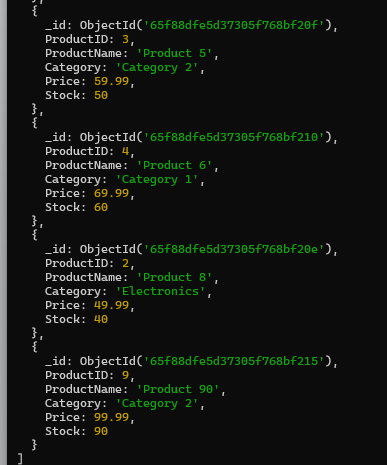


1. Display all the products in the "Electronics" category.



1. Display all the products in ascending order of their names.





1. Display the details of the first 5 products.



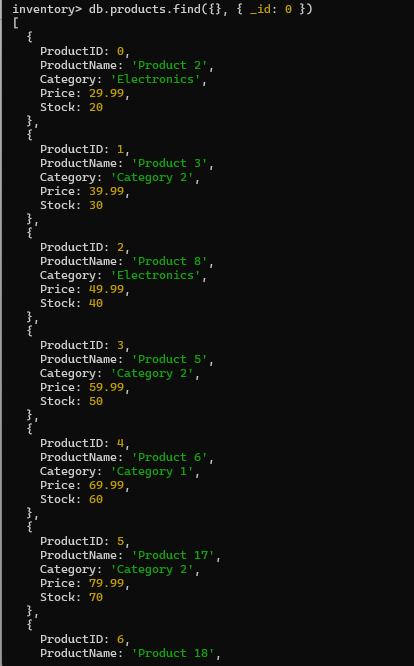
1. Display the categories of products with a specific name.



1. Display the number of products in the "Electronics" category.



1. Display all the products without showing the "\_id" field.

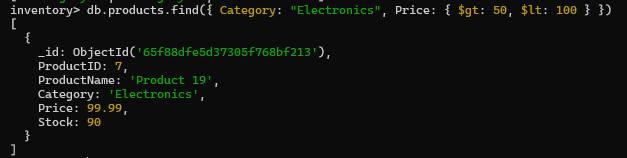




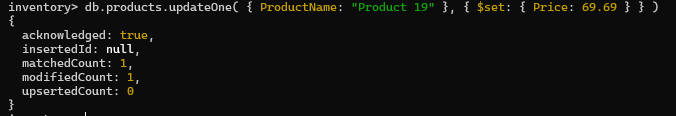
1. Display all the distinct categories of products.



1. Display products in the "Electronics" category with prices greater than 50 but less than 100.



1. Change the price of a product.



1. Delete a particular product entry.

****

**B) Create a set of RESTful endpoints using Node.js, Express, and Mongoose for handling student data operations.**

The endpoints should support:

* Retrieve a list of all students.
* Retrieve details of an individual student by ID.
* Add a new student to the database.
* Update details of an existing student by ID.
* Delete a student from the database by ID.

Connect the server to MongoDB using Mongoose, and store student data with attributes: name, age, and grade.

**Code:**

**index.js (main code runs from here)**

const express = require("express");

const mongoose = require("mongoose");

const app = express();

app.use(express.json());

app.listen(3000, () => {

console.log(`Server Started at ${3000}`);

});

require("dotenv").config();

const mongoString = process.env.DATABASE\_URL;

mongoose.connect(mongoString);

const database = mongoose.connection;

database.on("error", (error) => {

console.log(error);

});

database.once("connected", () => {

console.log("Database Connected");

});

const routes = require("./routes/routes");

app.use("/api", routes);

—-----------------------------------------------------------------------------------------------

**models.js (to create a basic structure in db)**

const mongoose = require("mongoose");

const dataSchema = new mongoose.Schema({

name: {

required: true,

type: String,

},

age: {

required: true,

type: Number,

},

marks: {

required: true,

type: Number,

},

ID: {

required: true,

type: Number,

},

});

module.exports = mongoose.model("Data", dataSchema);

—-------------------------------------------------------------------------------------------------

**routes.js (All the API endpoints are written here)**

const express = require("express");

const router = express.Router();

//Post Method

router.post("/post", async (req, res) => {

const data = new Model({

name: req.body.name,

age: req.body.age,

marks: req.body.marks,

ID: req.body.ID,

});

try {

const dataToSave = await data.save();

res.status(200).json(dataToSave);

} catch (error) {

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

}

});

//Get all Method

router.get("/getAll", async (req, res) => {

try {

const data = await Model.find();

res.json(data);

} catch (error) {

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

}

});

router.get("/getOne/:id", async (req, res) => {

try {

const data = await Model.findById(req.params.id); // Fetch the document by ID

if (!data) {

return res.status(404).json({ message: "Idea not found" }); // Return 404 if the idea is not found

}

res.json(data); // Return the entire document

} catch (error) {

res.status(500).json({ message: error.message }); // Return 500 if there's an error

}

});

router.patch("/update/:id", async (req, res) => {

try {

const { id } = req.params;

const newData = req.body; // New data to update the document

const updatedData = await Model.findByIdAndUpdate(id, newData, {

new: true,

}); // Find and update the document

if (!updatedData) {

return res.status(404).json({ message: "Idea not found" }); // Return 404 if the idea is not found

}

res.json(updatedData); // Return the updated document

} catch (error) {

res.status(500).json({ message: error.message }); // Return 500 if there's an error

}

});

router.delete("/delete/:id", async (req, res) => {

try {

const id = req.params.id;

const data = await Model.findByIdAndDelete(id);

if (!data) {

return res.status(404).json({ message: "Document not found" }); // Return 404 if the document is not found

}

res.send(`Document with ID ${id} has been deleted.`);

} catch (error) {

res.status(500).json({ message: error.message }); // Return 500 if there's an error

}

});

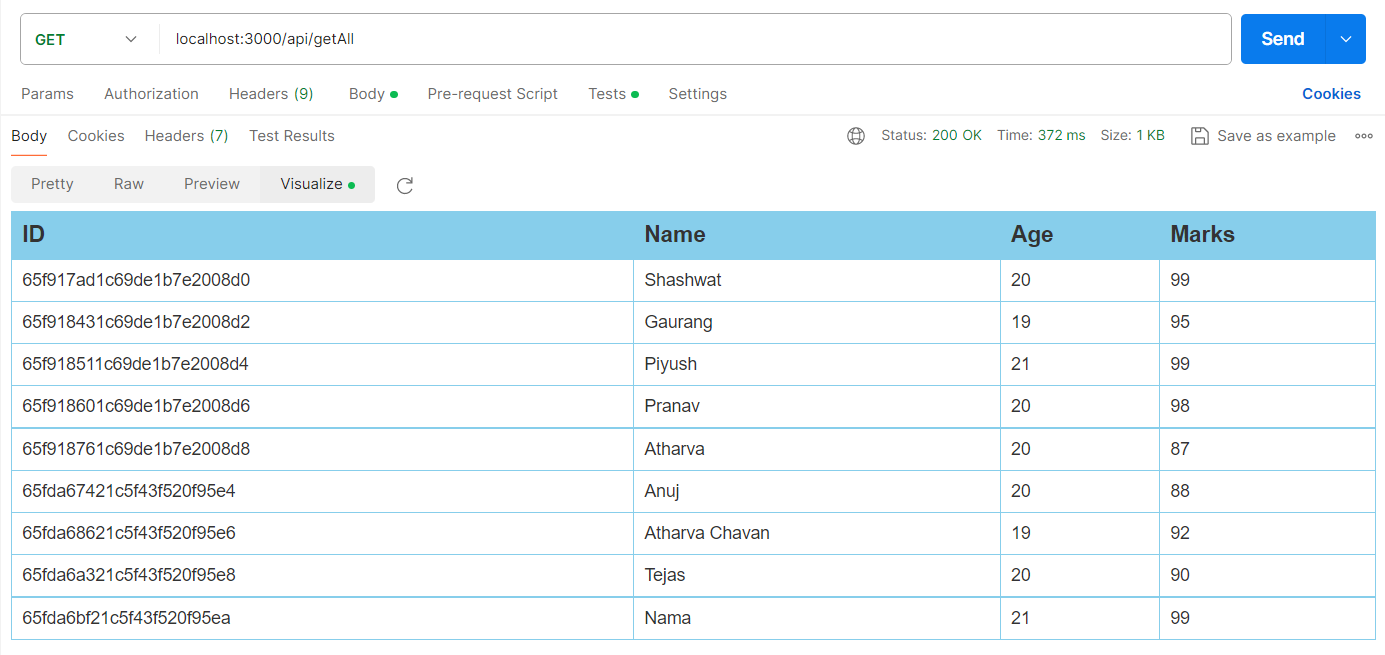
module.exports = router;

const Model = require("../model/model");

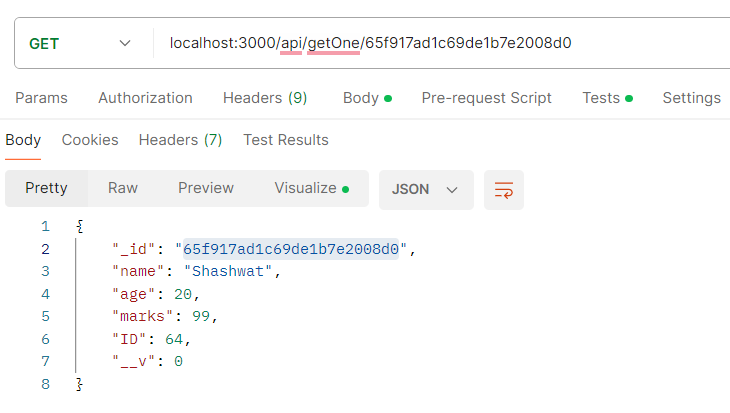
**All the API endpoints are called using Postman.**

**Output:**

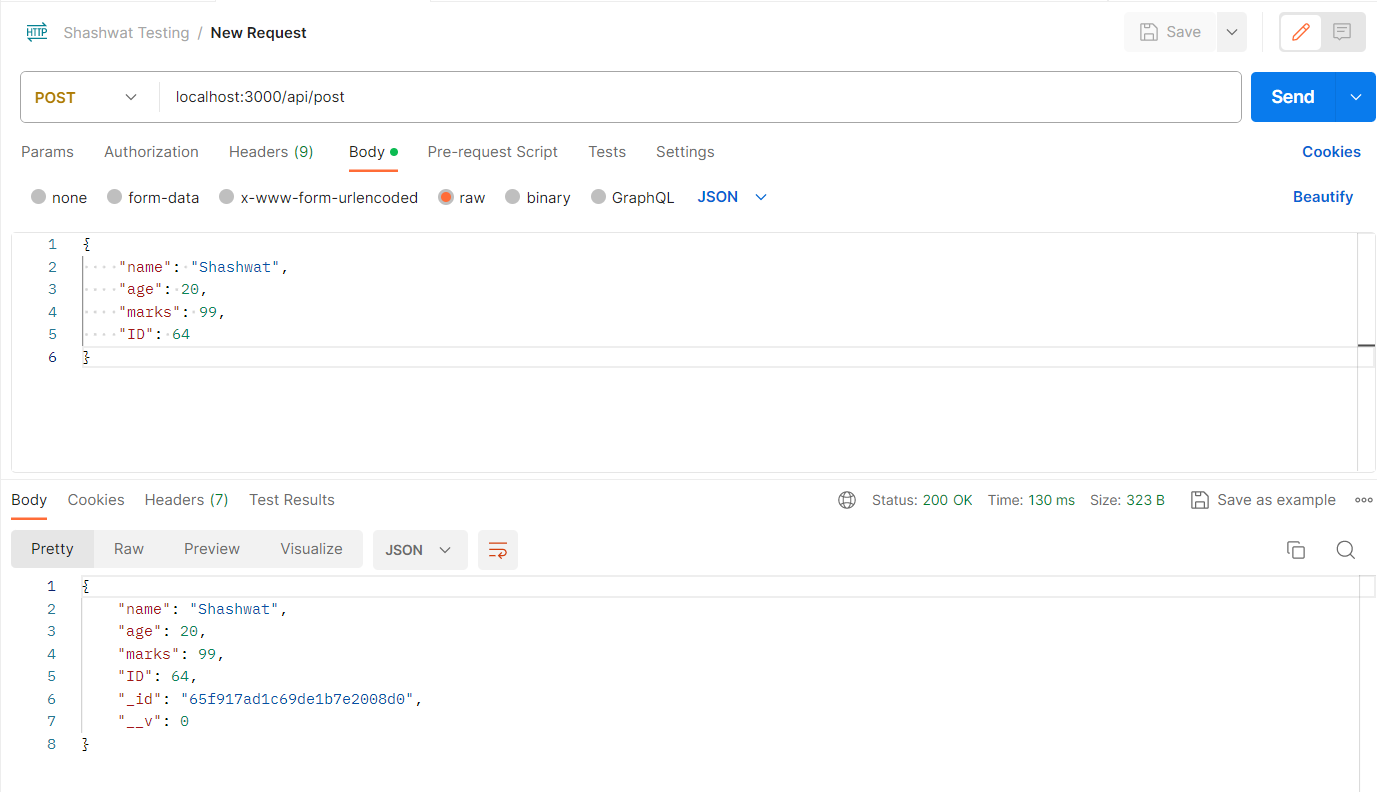
**Retrieve a list of all students.**

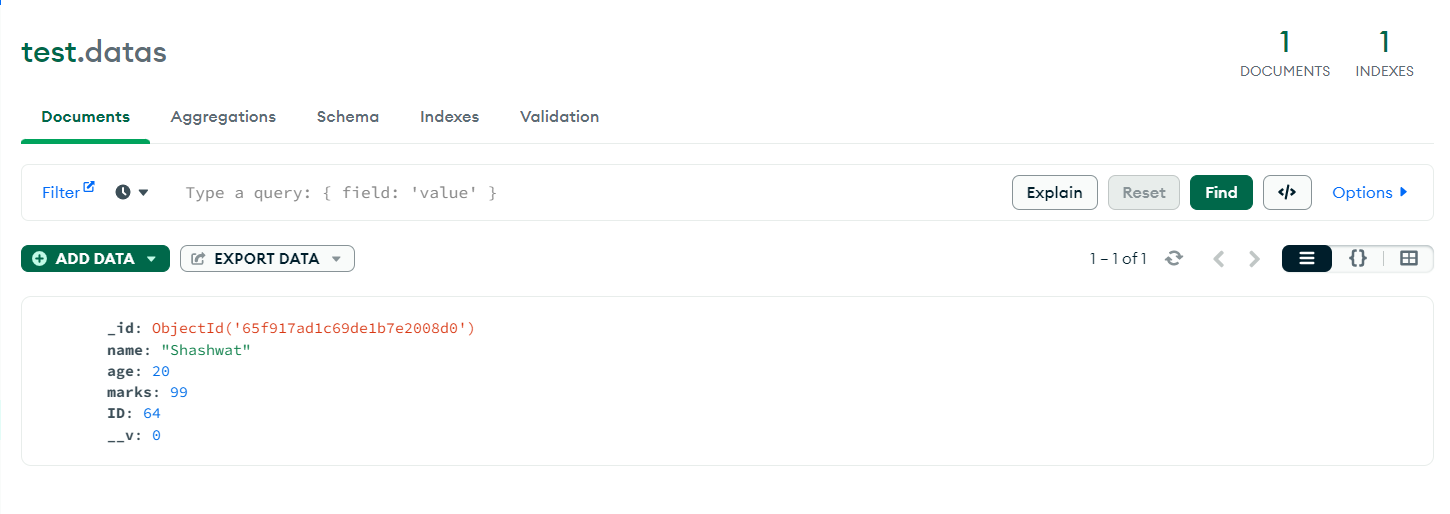


**Retrieve details of an individual student by ID.**



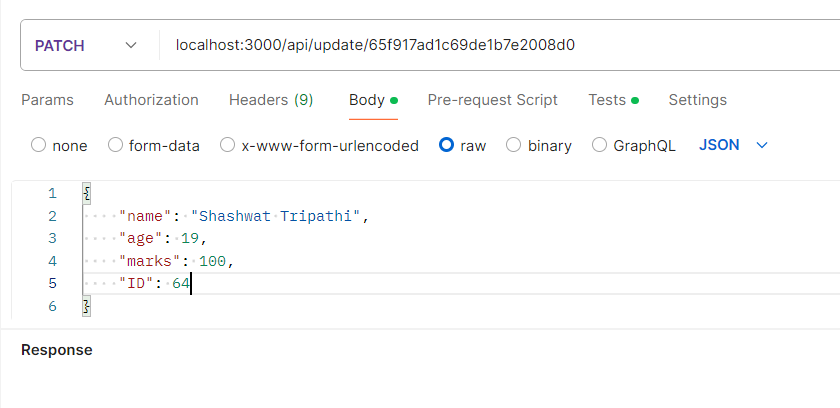
**Add a new student to the database.**



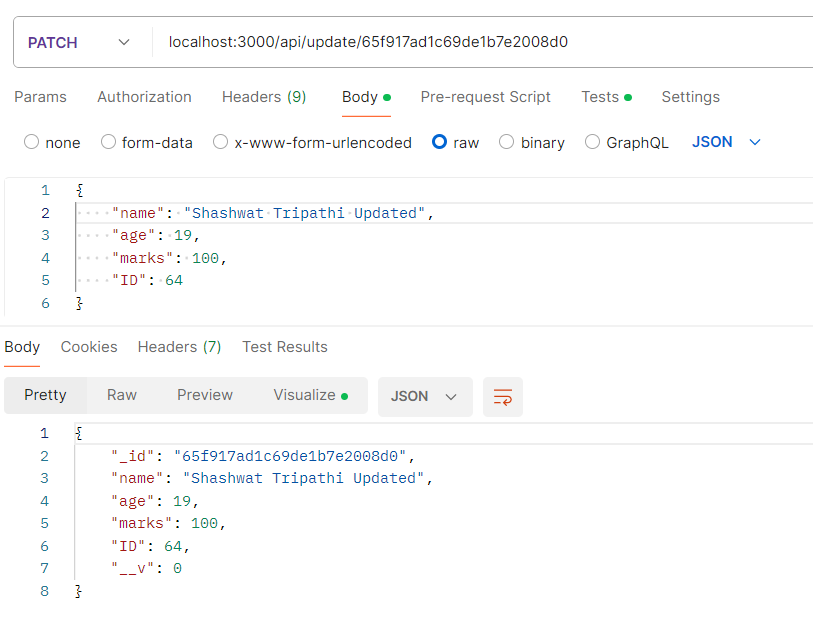


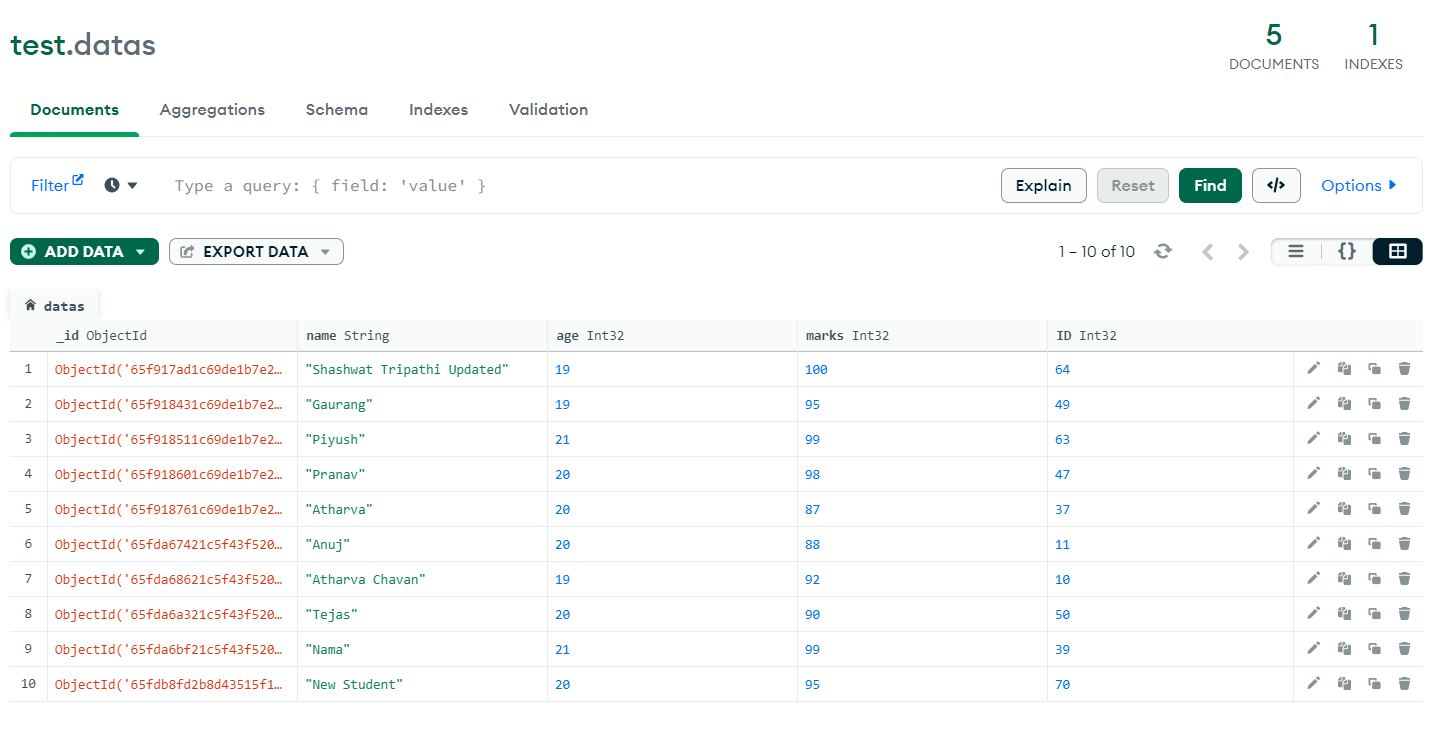
**Update details of an existing student by ID.**

Before updating the details of a student:

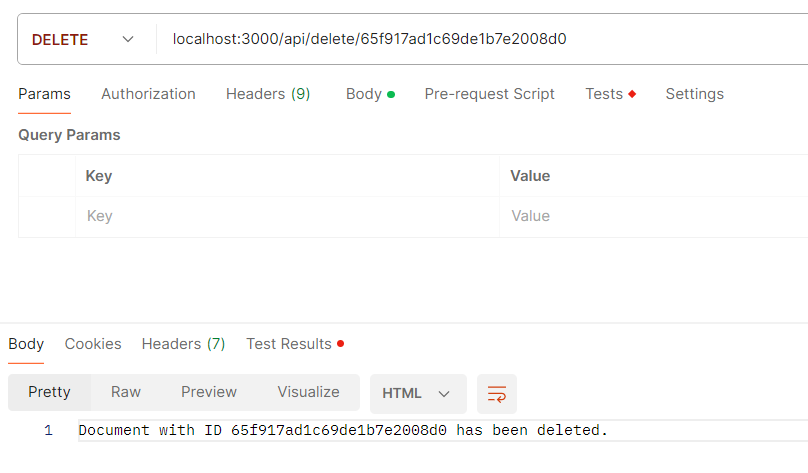


After updating the details:



In DB:  


**Delete a student from the database by ID.**

****

The student is deleted from the database:  


**Conclusion: Thus, we have studied and implemented CRUD operations in MongoDB.**