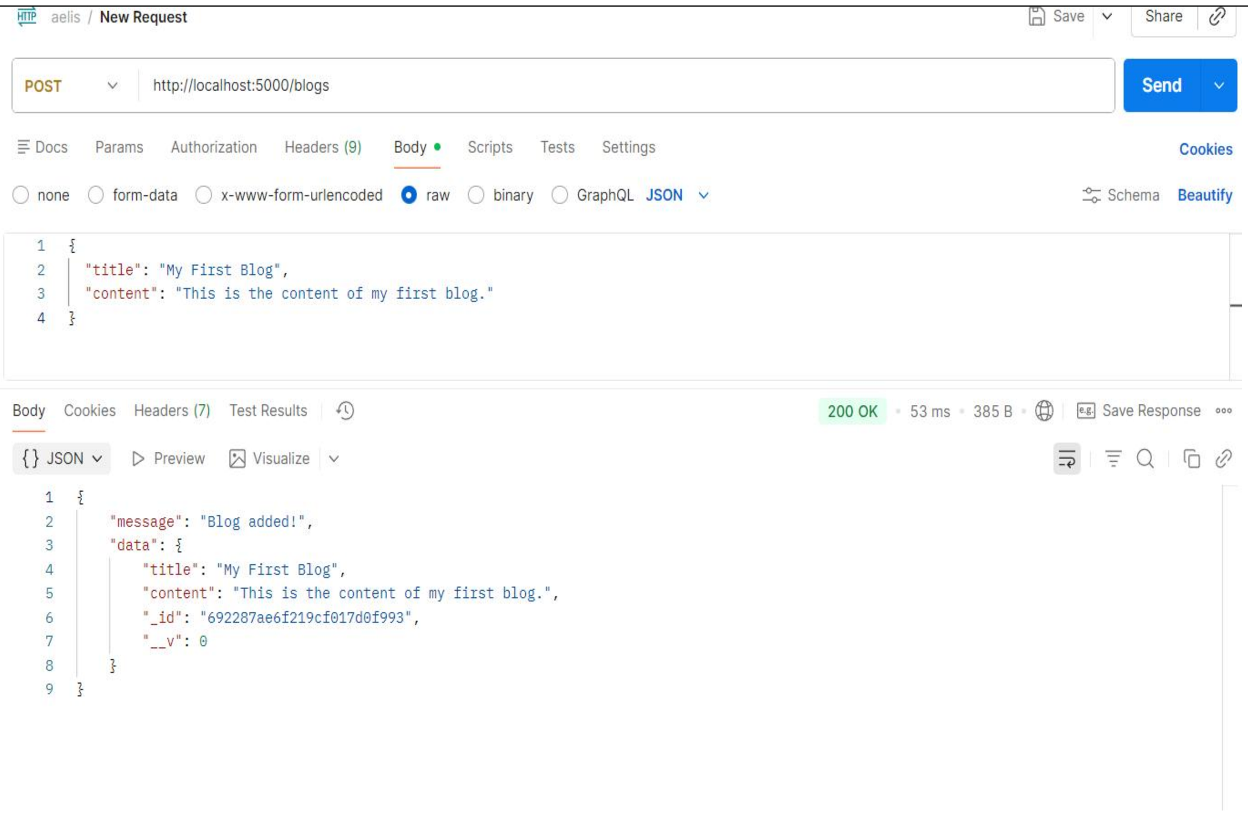
1	**While building a blogging application, users should be able to add new blog posts to the database. Sometimes, they might forget to include the title. Create a Mongoose model with validation to ensure that each blog post has both title and content. Additionally, configure your application to handle invalid submissions by returning a proper JSON error response.**

Code

```
const express = require("express");
const mongoose = require("mongoose");
const app = express();
app.use(express.json());
mongoose
  .connect("mongodb://127.0.0.1:27017/practical5")
  .then(() => console.log(" Connected to MongoDB"))
  .catch((err) => console.log(" MongoDB Error:", err.message));
const blogSchema = new mongoose.Schema({
  title: { type: String, required: [true, "Title is required"] },
  content: { type: String, required: [true, "Content is required"] },
});
const Blog = mongoose.model("Blog", blogSchema);
// Create blog post with validation
app.post("/blogs", async (req, res) => {
  try {
    const post = await Blog.create(req.body);
    res.json({ message: "Blog added!", data: post });
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});
const PORT = 5000;
app.listen(PORT, () => console.log(`Server running at http://localhost:${PORT}`));
```

Output



aelis / New Request

POST http://localhost:5000/blogs

```
{
  "title": "My First Blog",
  "content": "This is the content of my first blog."
}
```

200 OK · 53 ms · 385 B

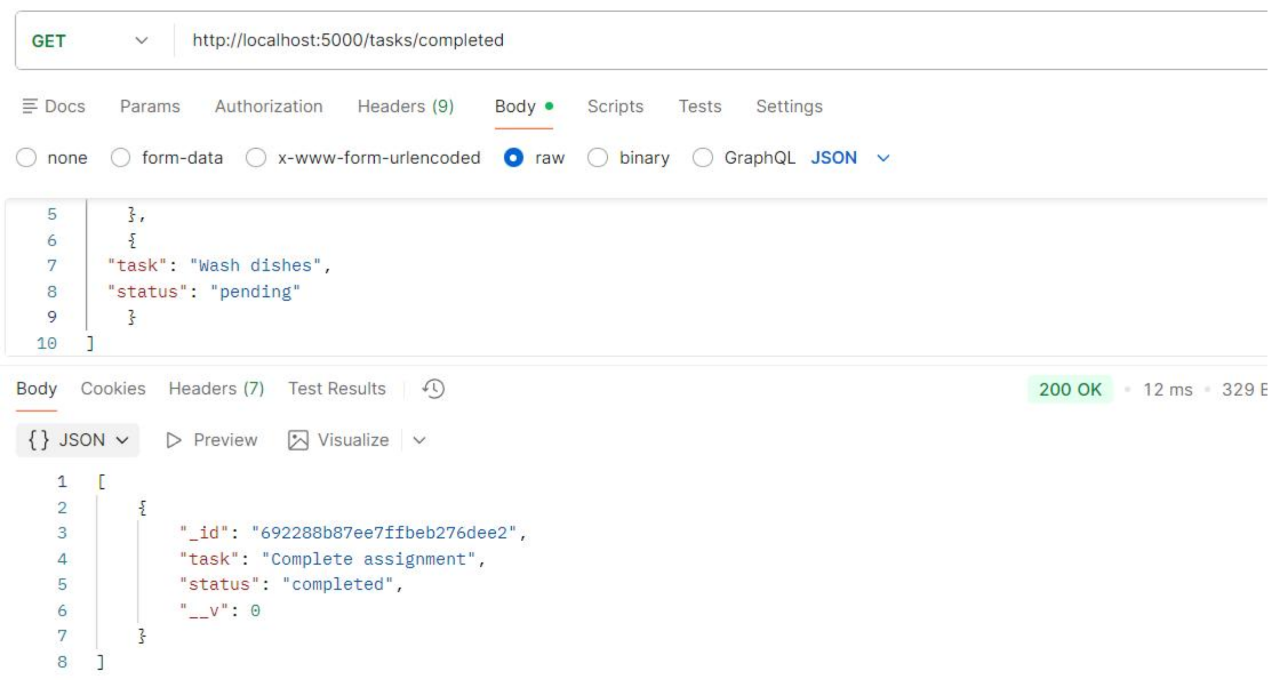
```
{
    "message": "Blog added!",
    "data": {
        "title": "My First Blog",
        "content": "This is the content of my first blog.",
        "_id": "692287ae6f219cf017d0f993",
        "__v": 0
    }
}
```

2

In a task management system, implement a Mongoose query to fetch and display all tasks where the status field is set to "completed"

Code

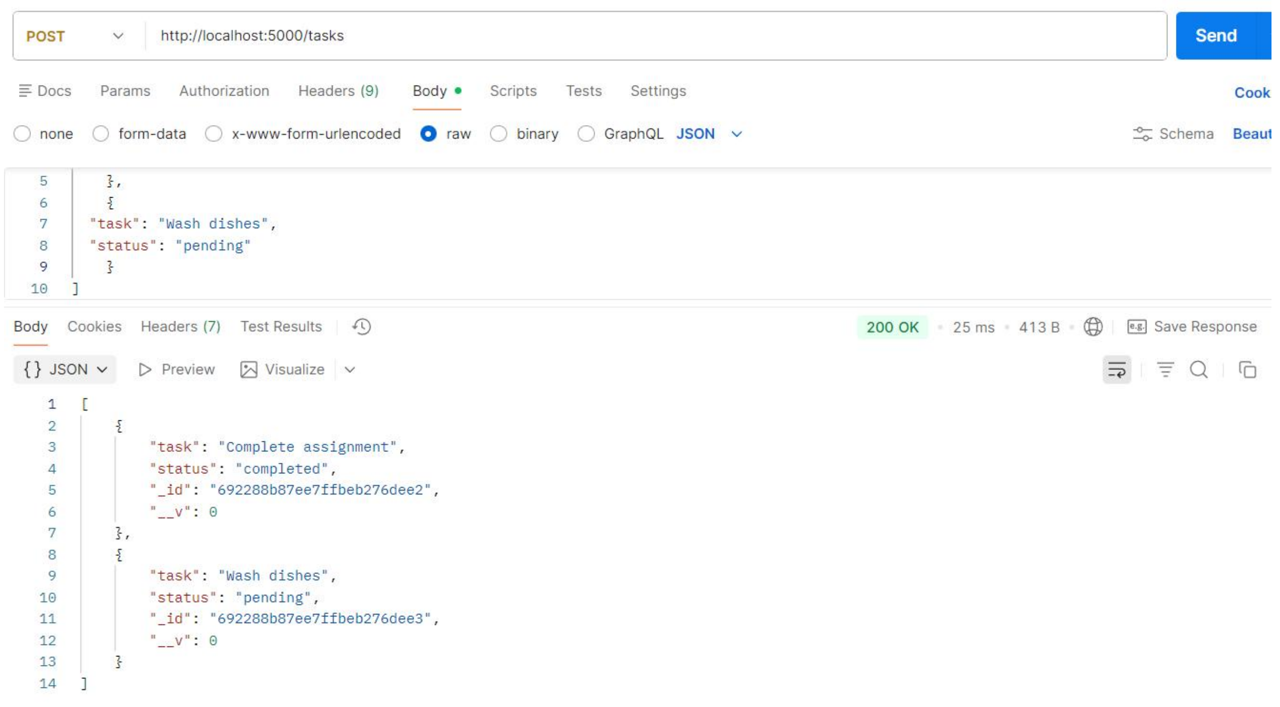
```
const taskSchema = new mongoose.Schema({
  task: String,
  status: String,
});

const Task = mongoose.model("Task", taskSchema);

app.get("/tasks/completed", async (req, res) => {
  const tasks = await Task.find({ status: "completed" });
  res.json(tasks);
});
```

Output	
	GET http://localhost:5000/tasks/completed
	Docs Params Authorization Headers (9) Body Scripts Tests Settings
	none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON
	`},`
	`{`
	`"task": "Wash dishes",`
	`"status": "pending"`
	`}`
	`]`
	Body Cookies Headers (7) Test Results 200 OK 12 ms 329 B
	{} JSON Preview Visualize
	`[`
	`{`
	`"_id": "692288b87ee7ffbeb276dee2",`
	`"task": "Complete assignment",`
	`"status": "completed",`
	`"__v": 0`
	`}`
	`]`

3	**You are working on a Task Management System where users can create tasks with different statuses, such as "completed", "in-progress", and "pending". You need to implement a Mongoose query to fetch and display all tasks that have the status field set to "completed".**

Code	**app.post("/tasks", async (req, res) => {**
	const task = await Task.create(req.body);
	res.json(task);
	});
	app.get("/tasks/completed", async (req, res) => {
	const tasks = await Task.find({ status: "completed" });
	res.json(tasks);
	});

Output


POST http://localhost:5000/tasks

```
  },
  {
"task": "Wash dishes",
"status": "pending"
  }
]
```

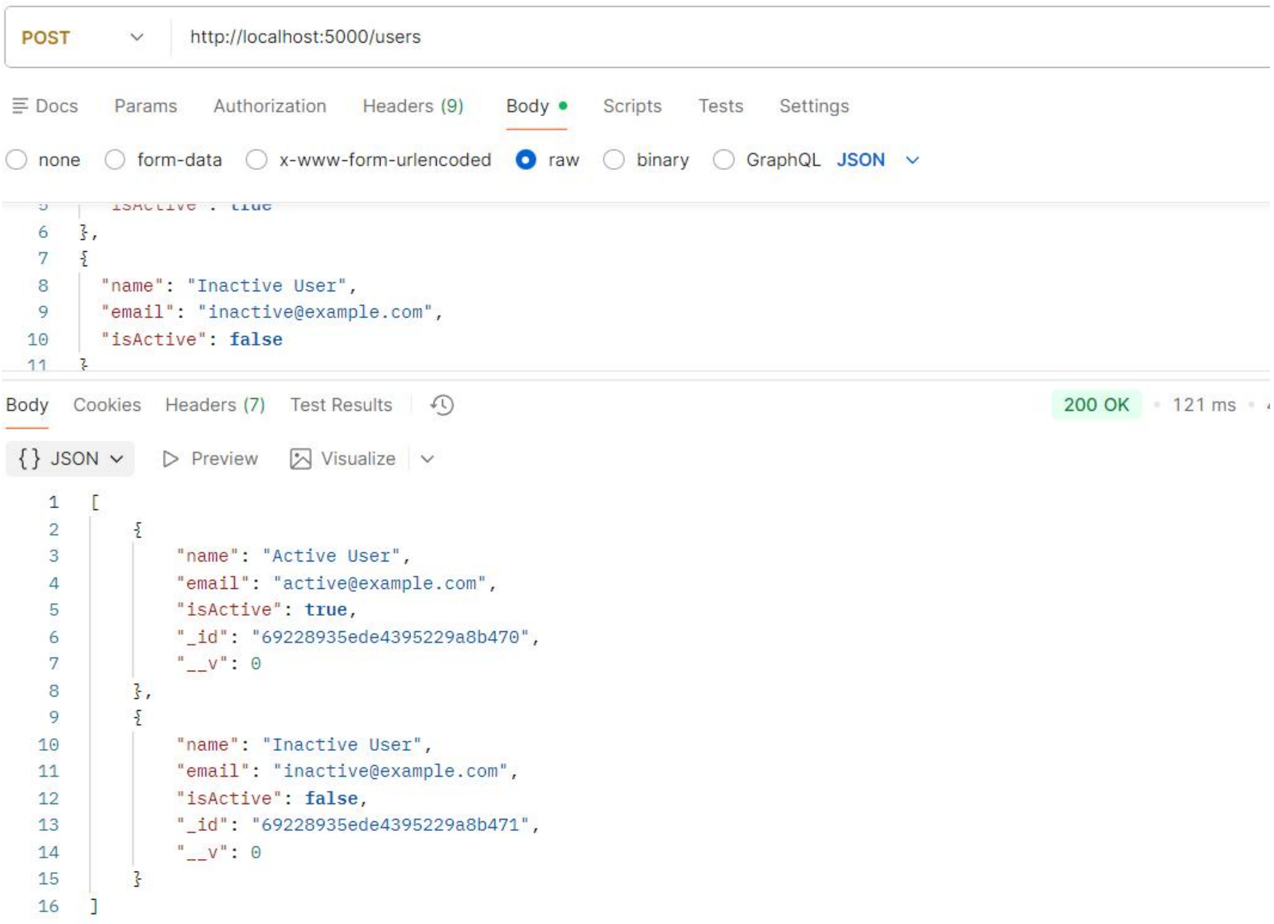
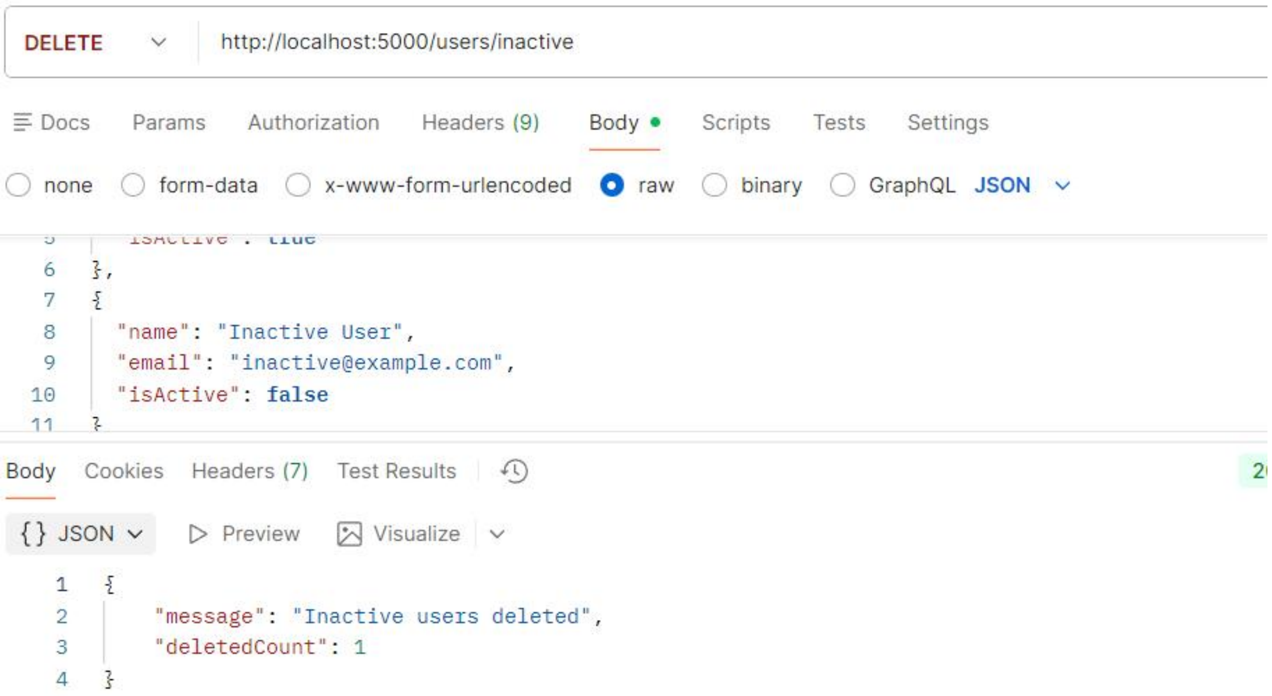
200 OK · 25 ms · 413 B

```
[
    {
        "task": "Complete assignment",
        "status": "completed",
        "_id": "692288b87ee7ffbeb276dee2",
        "__v": 0
    },
    {
        "task": "Wash dishes",
        "status": "pending",
        "_id": "692288b87ee7ffbeb276dee3",
        "__v": 0
    }
]
```

4 **You are building an admin panel where admins can delete multiple inactive users from the database in one go. Write the Mongoose query to remove all documents with isActive: false and send a success response with the number of deleted documents.**

Code

```
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  isActive: Boolean,
});
const User = mongoose.model("User", userSchema);
app.post("/users", async (req, res) => {
  const user = await User.create(req.body);
  res.json(user);
});
app.delete("/users/inactive", async (req, res) => {
  const result = await User.deleteMany({ isActive: false });
  res.json({
    message: "Inactive users deleted",
    deletedCount: result.deletedCount,
  });
```

	});
Output	 
5	**While designing a blogging platform, each blog post should store its comments.**

POST http://localhost:5000/users

Docs Params Authorization Headers (9) Body Scripts Tests Settings

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
    "isActive": true
},
{
  "name": "Inactive User",
  "email": "inactive@example.com",
  "isActive": false
}
```

Body Cookies Headers (7) Test Results 200 OK 121 ms

{} JSON Preview Visualize

```
[
    {
        "name": "Active User",
        "email": "active@example.com",
        "isActive": true,
        "_id": "69228935ede4395229a8b470",
        "__v": 0
    },
    {
        "name": "Inactive User",
        "email": "inactive@example.com",
        "isActive": false,
        "_id": "69228935ede4395229a8b471",
        "__v": 0
    }
]
```

DELETE http://localhost:5000/users/inactive

Docs Params Authorization Headers (9) Body Scripts Tests Settings

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
    "isActive": true
},
{
  "name": "Inactive User",
  "email": "inactive@example.com",
  "isActive": false
}
```

Body Cookies Headers (7) Test Results

{} JSON Preview Visualize

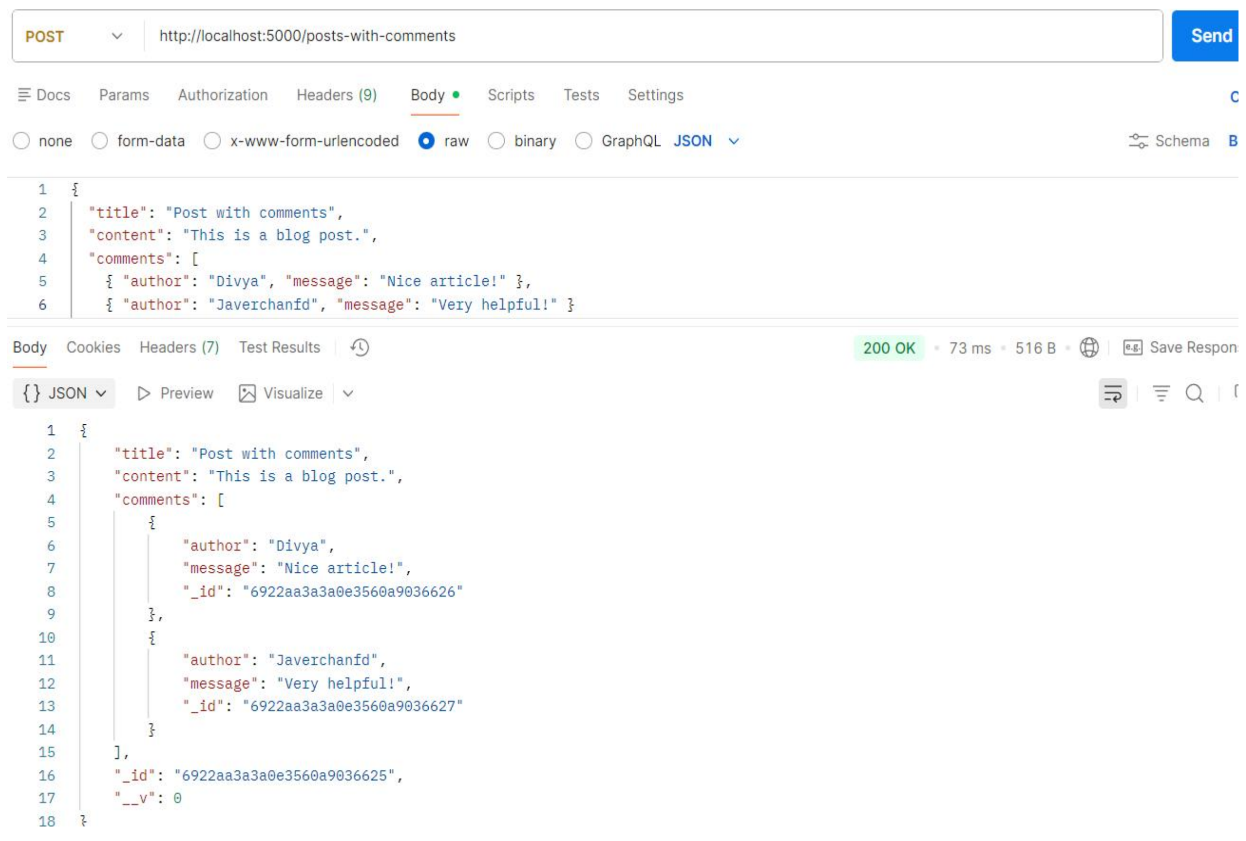
```
{
    "message": "Inactive users deleted",
    "deletedCount": 1
}
```

	Create a Mongoose schema that embeds comments inside the post document. Each comment should have author and message

Code

```
const commentSchema = new mongoose.Schema({
  author: String,
  message: String,
});
app.post("/posts-with-comments", async (req, res) => {
  const post = await Post.create(req.body);
  res.json(post);
});
const postSchema = new mongoose.Schema({
  title: String,
  content: String,
  comments: [commentSchema],
});
const Post = mongoose.model("PostWithComments", postSchema);
```

Output



http://localhost:5000/posts-with-comments

raw JSON

```
{
  "title": "Post with comments",
  "content": "This is a blog post.",
  "comments": [
    { "author": "Divya", "message": "Nice article!" },
    { "author": "Javerchanfd", "message": "Very helpful!" }
```

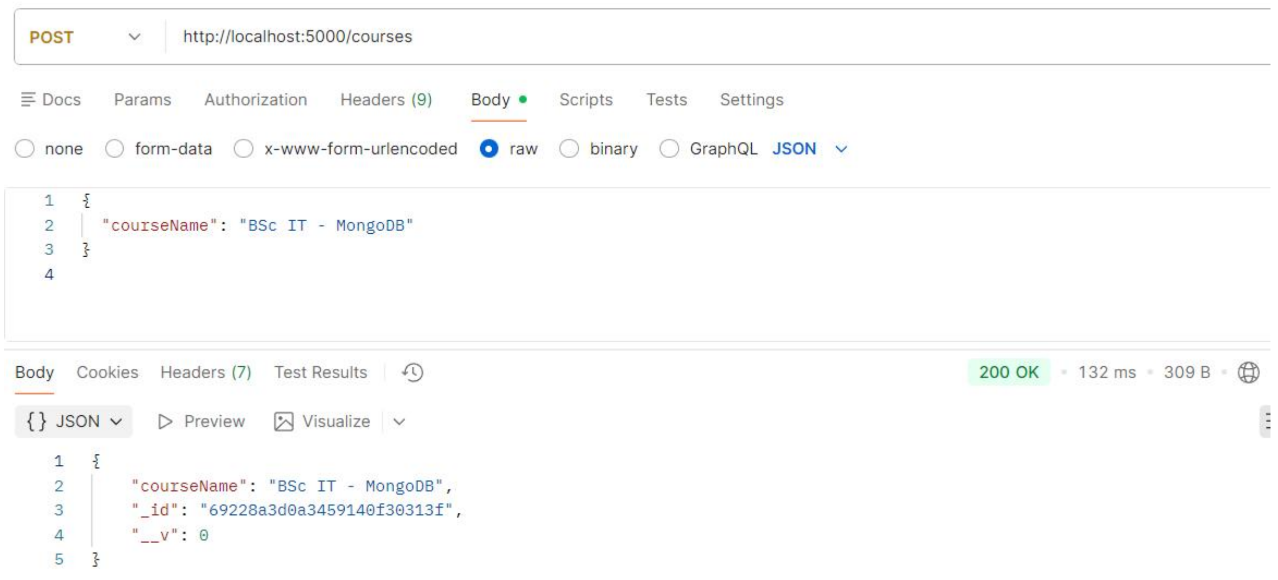
Body — 200 OK · 73 ms · 516 B

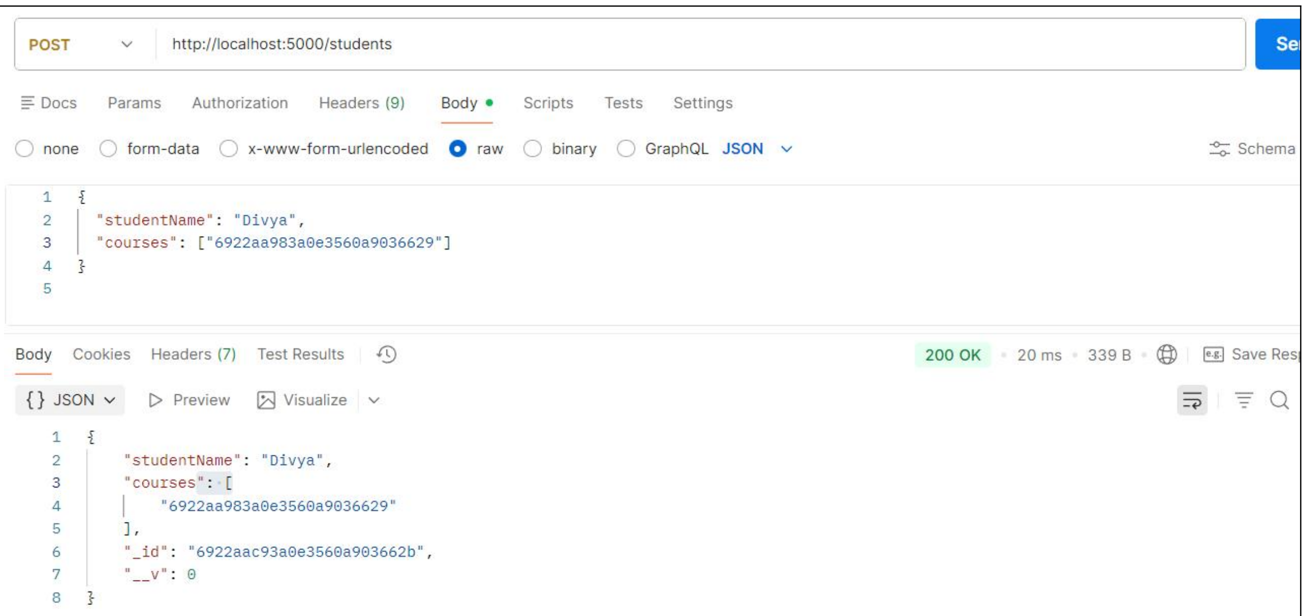
{} JSON

```
{
    "title": "Post with comments",
    "content": "This is a blog post.",
    "comments": [
        {
            "author": "Divya",
            "message": "Nice article!",
            "_id": "6922aa3a3a0e3560a9036626"
        },
        {
            "author": "Javerchanfd",
            "message": "Very helpful!",
            "_id": "6922aa3a3a0e3560a9036627"
        }
    ],
    "_id": "6922aa3a3a0e3560a9036625",
    "__v": 0
}
```

6	**In a student management application, each student can enroll in multiple courses. Design a normalized schema where student documents reference course documents.**

Code	
	```
	const courseSchema = new mongoose.Schema({
	  courseName: String,
	});
	const studentSchema = new mongoose.Schema({
	  studentName: String,
	  courses: [{ type: mongoose.Schema.Types.ObjectId, ref: "Course" }],
	});
	const Course = mongoose.model("Course", courseSchema);
	const Student = mongoose.model("Student", studentSchema);

	app.post("/courses", async (req, res) => {
	  const course = await Course.create(req.body);
	  res.json(course);
	});
	app.post("/students", async (req, res) => {
	  const student = await Student.create(req.body);
	  res.json(student);
	});
	```

Output	 POST http://localhost:5000/courses
	Body: raw JSON
	```
	{
	  "courseName": "BSc IT - MongoDB"
	}
	```
	200 OK · 132 ms · 309 B
	```
	{
	    "courseName": "BSc IT - MongoDB",
	    "_id": "69228a3d0a3459140f30313f",
	    "__v": 0
	}
	```



POST http://localhost:5000/students

```
{
  "studentName": "Divya",
  "courses": ["6922aa983a0e3560a9036629"]
}
```

200 OK • 20 ms • 339 B

```
{
    "studentName": "Divya",
    "courses": [
        "6922aa983a0e3560a9036629"
    ],
    "_id": "6922aac93a0e3560a903662b",
    "__v": 0
}
```

7 **You're building a social media app where users can like posts and interact with each other's profiles. Your task is to design the database schema using Mongoose (a MongoDB ODM for Node.js) with a hybrid data model, where:**

1. Likes are stored directly in the post document (embedded data).

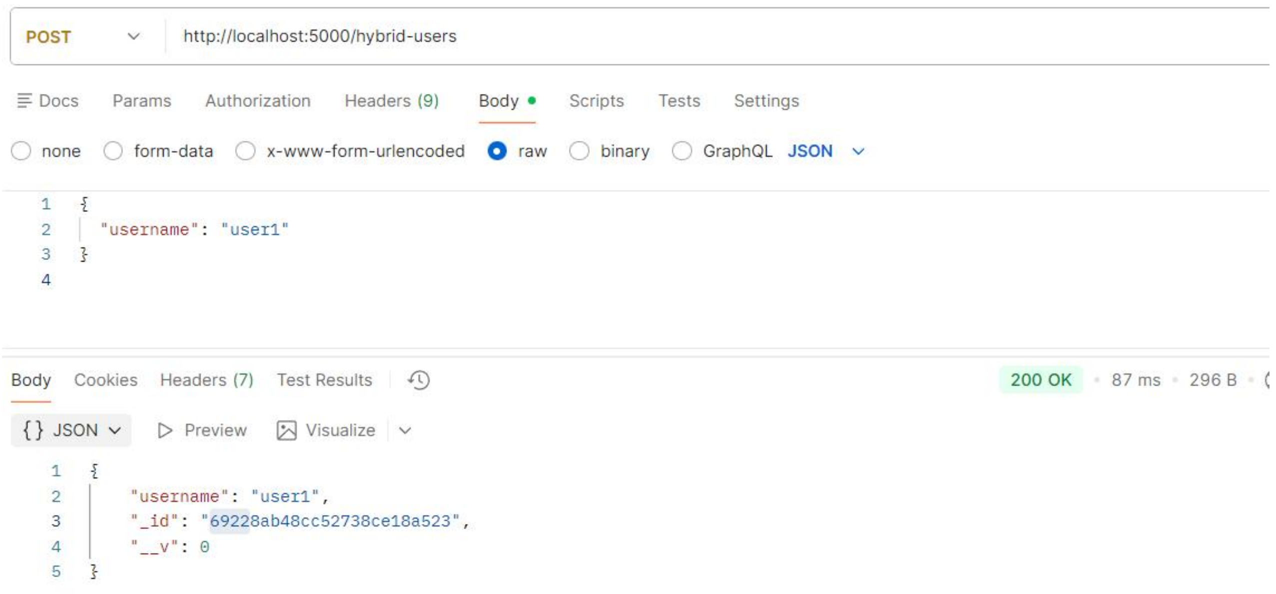
2. User profiles are stored as references in the database, linking users to the posts they like (referenced data).

Code

```
const user7Schema = new mongoose.Schema({
  username: String,
});

const likeSchema = new mongoose.Schema({
  user: { type: mongoose.Schema.Types.ObjectId, ref: "HybridUser" },
});

const hybridPostSchema = new mongoose.Schema({
  text: String,
  likes: [likeSchema],
});

const HybridUser = mongoose.model("HybridUser", user7Schema);
const HybridPost = mongoose.model("HybridPost", hybridPostSchema);
```

```
app.post("/hybrid-users", async (req, res) => {
  const user = await HybridUser.create(req.body);
  res.json(user);
});


app.post("/hybrid-posts", async (req, res) => {
  const post = await HybridPost.create(req.body);
  res.json(post);
});
```

Output

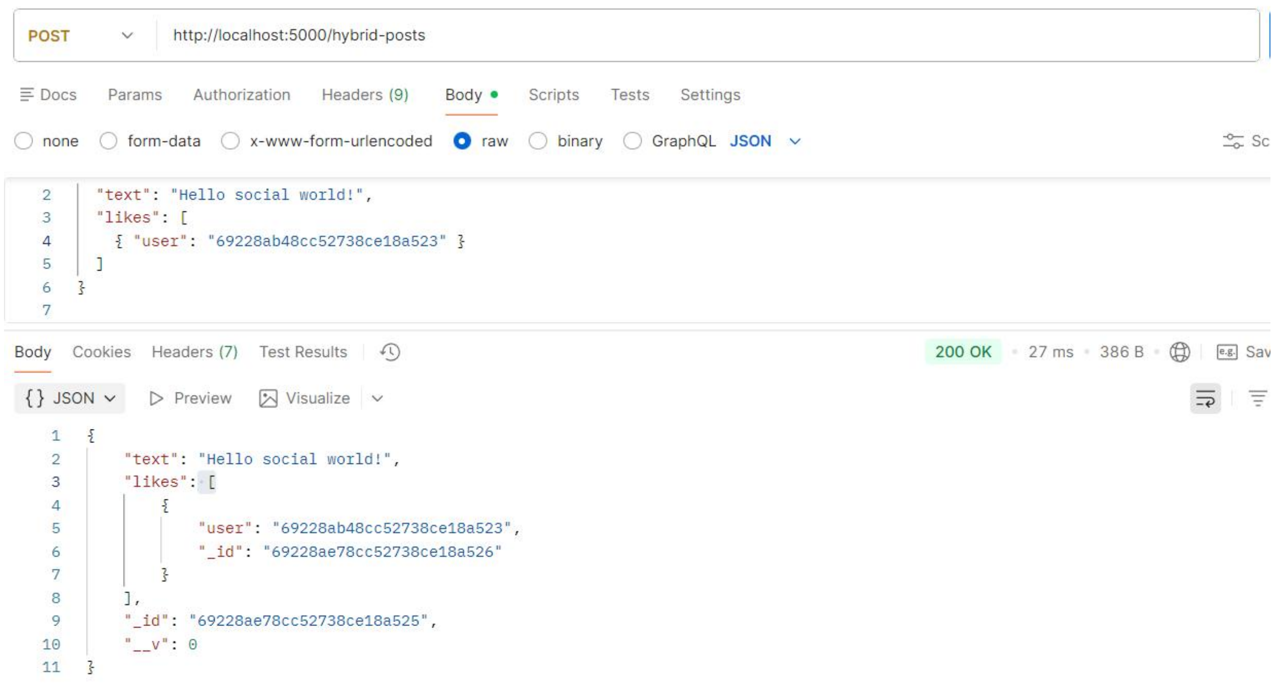


http://localhost:5000/hybrid-users

```
{
  "username": "user1"
}
```

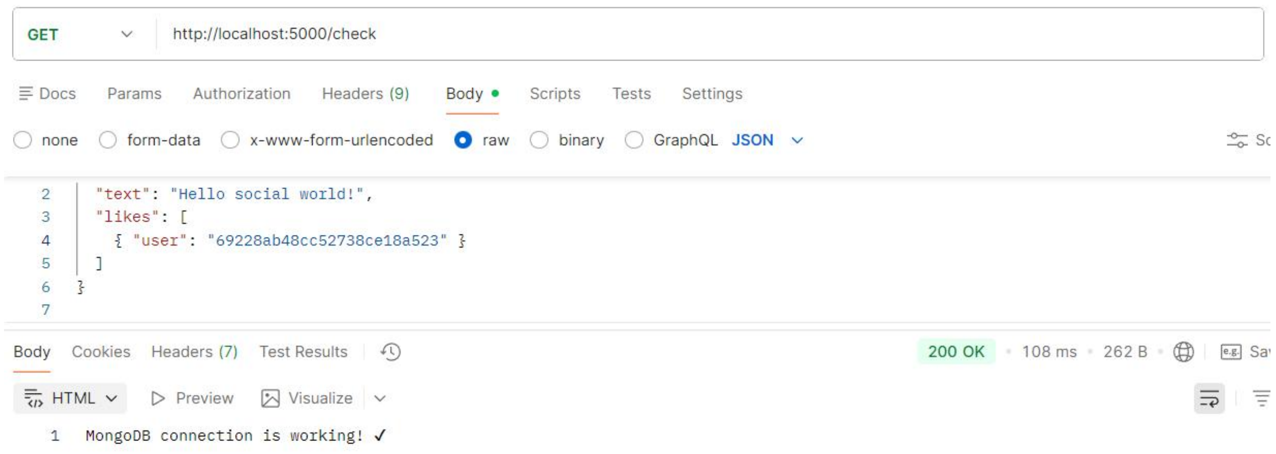
200 OK · 87 ms · 296 B

```
{
    "username": "user1",
    "_id": "69228ab48cc52738ce18a523",
    "__v": 0
}
```

	
	POST http://localhost:5000/hybrid-posts
	Docs Params Authorization Headers (9) Body ● Scripts Tests Settings
	none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON
	`"text": "Hello social world!",`
	`"likes": [`
	`{ "user": "69228ab48cc52738ce18a523" }`
	`]`
	`}`
	Body Cookies Headers (7) Test Results 200 OK · 27 ms · 386 B
	{} JSON Preview Visualize
	`{`
	`"text": "Hello social world!",`
	`"likes": [`
	`{`
	`"user": "69228ab48cc52738ce18a523",`
	`"_id": "69228ae78cc52738ce18a526"`
	`}`
	`],`
	`"_id": "69228ae78cc52738ce18a525",`
	`"__v": 0`
	`}`

8	**While developing a Node.js application, connect it to a local MongoDB instance using Mongoose. Write the code to establish the connection and log " Connected to MongoDB" on success, or an error message otherwise.**

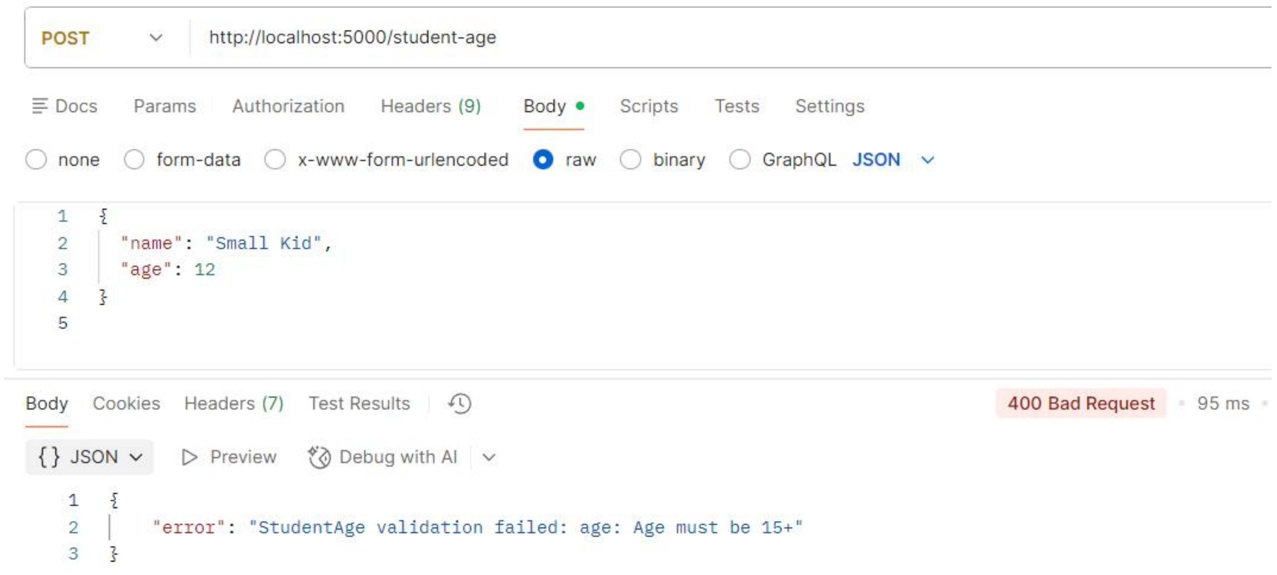
Code	**app.get("/check", (req, res) => {**
	res.send("MongoDB connection is working! ✔");
	});

Output	
	GET http://localhost:5000/check
	Docs Params Authorization Headers (9) Body ● Scripts Tests Settings
	none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON
	`"text": "Hello social world!",`
	`"likes": [`
	`{ "user": "69228ab48cc52738ce18a523" }`
	`]`
	`}`
	Body Cookies Headers (7) Test Results 200 OK · 108 ms · 262 B
	HTML Preview Visualize
	`MongoDB connection is working! ✓`

9	**While developing a school app, ensure that students younger than 15 cannot be**

	saved to the database. Implement this using Mongoose schema validation.

Code	
	```
	const studentAgeSchema = new mongoose.Schema({
	  name: String,
	  age: { type: Number, min: [15, "Age must be 15+"] },
	});
	const StudentAge = mongoose.model("StudentAge", studentAgeSchema);
	app.post("/student-age", async (req, res) => {
	  try {
	    const s = await StudentAge.create(req.body);
	    res.json(s);
	  } catch (err) {
	    res.status(400).json({ error: err.message });
	  }
	});
	```

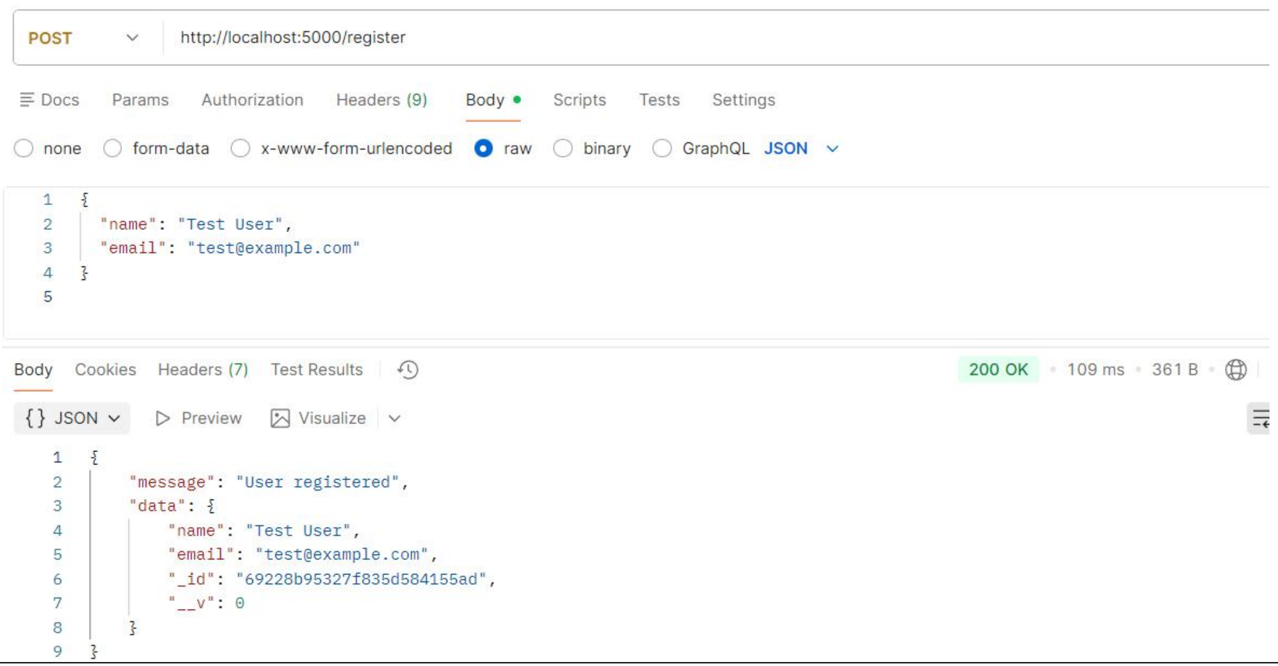
Output	 POST http://localhost:5000/student-age
	Body: raw JSON
	```
	{
	  "name": "Small Kid",
	  "age": 12
	}
	```
	400 Bad Request · 95 ms
	```
	{
	    "error": "StudentAge validation failed: age: Age must be 15+"
	}
	```

10	**You are building a user registration API for a web application. When a new user tries to register, you need to ensure that the email address they provide is unique. Implement the registration logic so that if a user attempts to register with an email that already exists in the database, the API should respond with a JSON error**

Code	
	```
	const regUserSchema = new mongoose.Schema({
	  name: String,
	```

```
  email: { type: String, unique: true },
});

const RegUser = mongoose.model("RegUser", regUserSchema);

app.post("/register", async (req, res) => {
  try {
    const user = await RegUser.create(req.body);
    res.json({ message: "User registered", data: user });
  } catch (err) {
    if (err.code === 11000) {
      return res.status(400).json({ error: "Email already exists" });
    }
    res.status(500).json({ error: err.message });
  }
});
```

Output

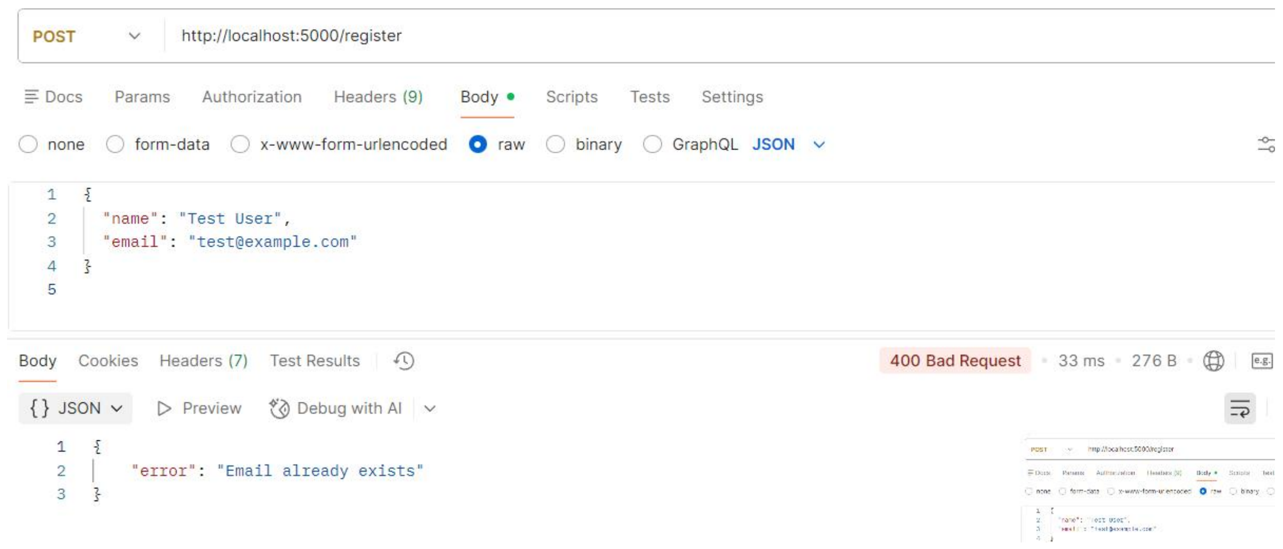


POST http://localhost:5000/register

```
{
  "name": "Test User",
  "email": "test@example.com"
}
```

Body — 200 OK • 109 ms • 361 B — JSON

```
{
    "message": "User registered",
    "data": {
        "name": "Test User",
        "email": "test@example.com",
        "_id": "69228b95327f835d584155ad",
        "__v": 0
    }
}
```



POST http://localhost:5000/register

```
{
  "name": "Test User",
  "email": "test@example.com"
}
```

Body — 400 Bad Request · 33 ms · 276 B — {} JSON

```
{
    "error": "Email already exists"
}
```

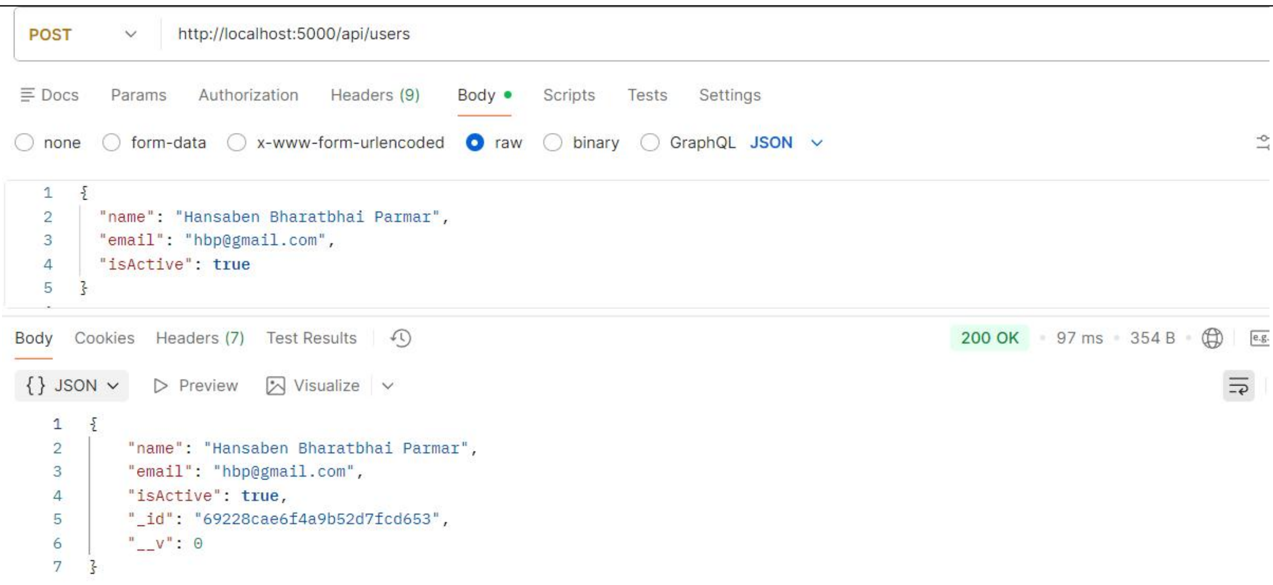
11 **You are building a simple user management system using Express.js and MongoDB. Perform the following tasks:**

- **Connect to MongoDB using Mongoose in your Express.js application.**
- **Create a User model with fields: name, email, and password.**
- **Implement routes to:**
  - **Add a new user**
  - **View all users**
  - **View a user by ID**
  - **Update a user by ID**
  - **Delete a user by ID**
- **Test all routes using Postman or any API client**

Code

```
app.post("/api/users", async (req, res) => {
  const user = await User.create(req.body);
  res.json(user);
});


app.get("/api/users", async (req, res) => {
  const users = await User.find();
  res.json(users);
});


app.get("/api/users/:id", async (req, res) => {
  const user = await User.findById(req.params.id);
  res.json(user);
});

app.put("/api/users/:id", async (req, res) => {
  const updated = await User.findByIdAndUpdate(req.params.id, req.body, {
    new: true,
  });
  res.json(updated);
});

app.delete("/api/users/:id", async (req, res) => {
  await User.findByIdAndDelete(req.params.id);
  res.json({ message: "User deleted" });
});
```

Output



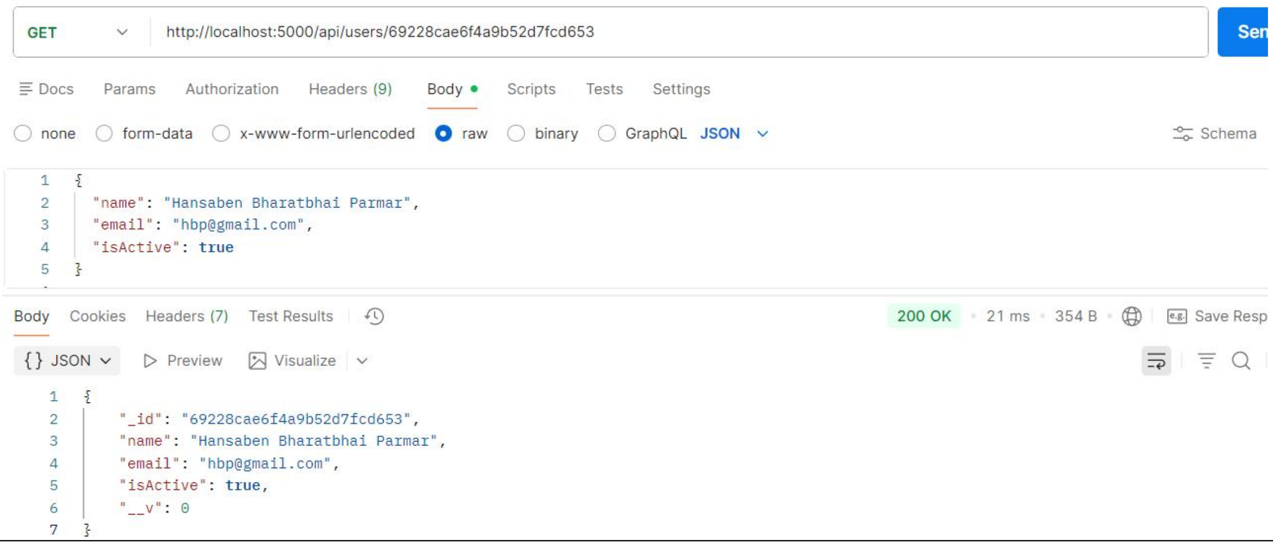
POST http://localhost:5000/api/users

none | form-data | x-www-form-urlencoded | [✓ raw] | binary | GraphQL | JSON

```
{
  "name": "Hansaben Bharatbhai Parmar",
  "email": "hbp@gmail.com",
  "isActive": true
}
```

200 OK • 97 ms • 354 B

```
{
    "name": "Hansaben Bharatbhai Parmar",
    "email": "hbp@gmail.com",
    "isActive": true,
    "_id": "69228cae6f4a9b52d7fcd653",
    "__v": 0
}
```



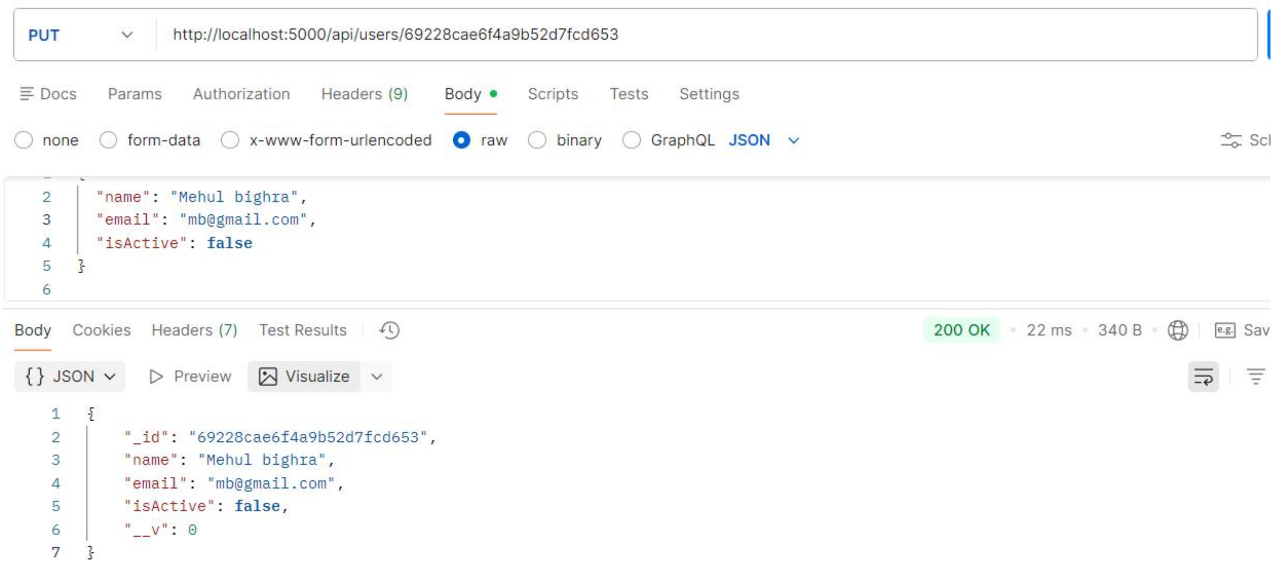
GET http://localhost:5000/api/users/69228cae6f4a9b52d7fcd653

none | form-data | x-www-form-urlencoded | [✓ raw] | binary | GraphQL | JSON

```
{
  "name": "Hansaben Bharatbhai Parmar",
  "email": "hbp@gmail.com",
  "isActive": true
}
```

200 OK • 21 ms • 354 B

```
{
    "_id": "69228cae6f4a9b52d7fcd653",
    "name": "Hansaben Bharatbhai Parmar",
    "email": "hbp@gmail.com",
    "isActive": true,
    "__v": 0
}
```



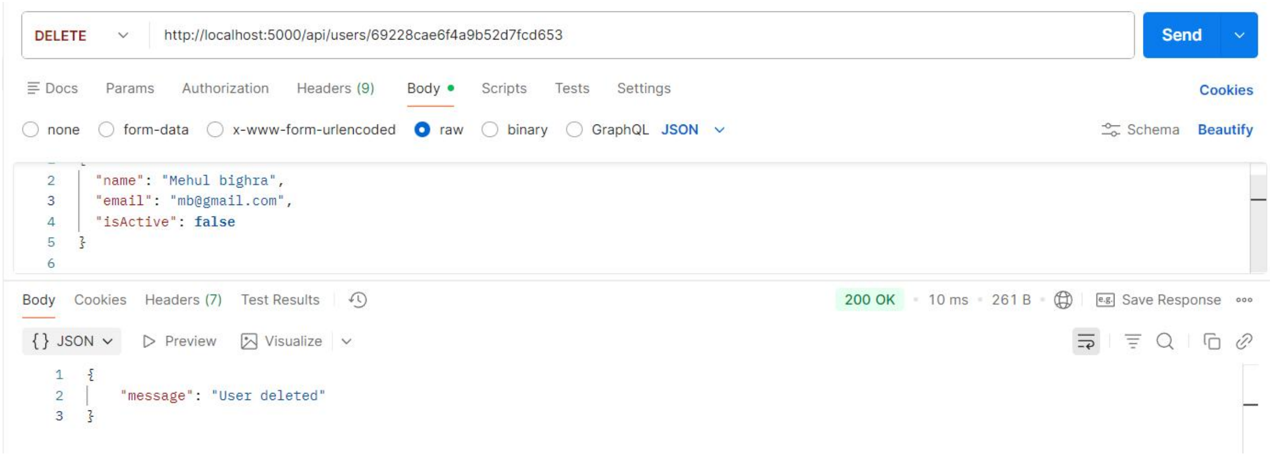
PUT http://localhost:5000/api/users/69228cae6f4a9b52d7fcd653

Body: raw JSON

```
{
  "name": "Mehul bighra",
  "email": "mb@gmail.com",
  "isActive": false
}
```

200 OK · 22 ms · 340 B

```
{
    "_id": "69228cae6f4a9b52d7fcd653",
    "name": "Mehul bighra",
    "email": "mb@gmail.com",
    "isActive": false,
    "__v": 0
}
```



DELETE http://localhost:5000/api/users/69228cae6f4a9b52d7fcd653

Body: raw JSON

```
{
  "name": "Mehul bighra",
  "email": "mb@gmail.com",
  "isActive": false
}
```

200 OK · 10 ms · 261 B

```
{
    "message": "User deleted"
}
```