Here's a **Basic Crud full-stack setup** using **React.js**, **Express.js**, and **MongoDB** to help you get started. We'll break it into parts:

□ 1. Backend – Express + MongoDB

a. Setup

Create a new backend folder and initialize a Node.js project:

```
mkdir backend
cd backend
npm init -y
```

b. Install dependencies

```
npm install express mongoose cors
```

c. Create basic Express server (`index.js`)

```
// backend/index.js
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const app = express();
app.use(cors());
app.use(express.json());

// MongoDB connection
mongoose.connect('mongodb://localhost:27017/mydb', {
 useNewUrlParser: true,
 useUnifiedTopology: true,
})
.then(() => console.log('MongoDB connected'))
.catch(err => console.error(err));

// Simple schema

const UserSchema = new mongoose.Schema({
 name: String,
 email: String,
});
const User = mongoose.model('User', UserSchema);

// Routes

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

```
// Add user record

app.post('/api/users', async (req, res) => {
 const user = new User(req.body);
await user.save();
 res.json(user);
});

// Update user by ID

app.put('/api/users/:id', async (req, res) => {
  try {
    const updatedUser = await User.findByIdAndUpdate(
      req.params.id,
      req.body,
      { new: true } // return the updated document
    );
    res.json(updatedUser);
  } catch (err) {
    res.status(500).json({ error: 'Failed to update user' });
  }
});


// Delete user by ID

app.delete('/api/users/:id', async (req, res) => {
  try {
    await User.findByIdAndDelete(req.params.id);
    res.json({ message: 'User deleted successfully' });
  } catch (err) {
    res.status(500).json({ error: 'Failed to delete user' });
  }
});



// Start server

app.listen(5000, () => {

 console.log('Server running on http://localhost:5000');
});
```

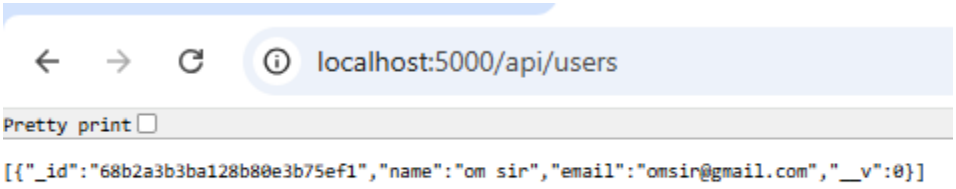
□ Final Steps

- Run backend:

```
node index.js
```

For Output Open Browser:-

http://localhost:5000/api/users



□ 2. Frontend – React

a. Setup React project

In a separate folder:

```
npx create-react-app frontend
cd frontend
npm install axios
```

b. Create a basic Crud (form and display list, edit , delete) (`App.js`)

```
// frontend/src/App.js

import React, { useEffect, useState } from 'react';

import axios from 'axios';


function App() {

  const [users, setUsers] = useState([]);

  const [form, setForm] = useState({ name: '', email: '' });

  const [editingUserId, setEditingUserId] = useState(null);


  const API_URL = 'http://localhost:5000/api/users';


  // Fetch users on mount

  useEffect(() => {

    fetchUsers();

  }, []);


  const fetchUsers = async () => {
```

```
  try {
    const res = await axios.get(API_URL);
    setUsers(res.data);
  } catch (err) {
    console.error('Error fetching users:', err);
  }
};

const handleInputChange = (e) => {
  const { name, value } = e.target;
  setForm(prev => ({ ...prev, [name]: value }));
};

const createUser = async () => {
  try {
    await axios.post(API_URL, form);
    fetchUsers();
  } catch (err) {
    console.error('Error creating user:', err);
  }
};

const updateUser = async () => {
  try {
    await axios.put(`${API_URL}/${editingUserId}`, form);
    fetchUsers();
```

```
    } catch (err) {
      console.error('Error updating user:', err);
    }
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    if (editingUserId) {
      await updateUser();
    } else {
      await createUser();
    }
    setForm({ name: '', email: '' });
    setEditingUserId(null);
  };

  const handleEdit = (user) => {
    setForm({ name: user.name, email: user.email });
    setEditingUserId(user._id);
  };

  const handleDelete = async (id) => {
  const confirmDelete = window.confirm('Are you sure you want to delete this
user?');
  if (!confirmDelete) return;
```

```
  try {
    await axios.delete(`${API_URL}/${id}`);
    fetchUsers();
  } catch (err) {
    console.error('Error deleting user:', err);
  }
};



  return (
    <div style={{ padding: 30 }}>
      <h2>Users CRUD (React + Axios)</h2>

      <form onSubmit={handleSubmit}>
        <input
          type="text"
          name="name"
          placeholder="Name"
          value={form.name}
          onChange={handleInputChange}
          required
        />
        <input
          type="email"
          name="email"
          placeholder="Email"
```

```
          value={form.email}
          onChange={handleInputChange}
          required
        />
        <button type="submit">
          {editingUserId ? 'Update' : 'Add'} User
        </button>
      </form>

      <hr />

      <ul>
        {users.map(user => (
          <li key={user._id}>
            <strong>{user.name}</strong> ({user.email})
            <button onClick={() => handleEdit(user)}>Edit</button>
            <button onClick={() => handleDelete(user._id)}>Delete</button>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

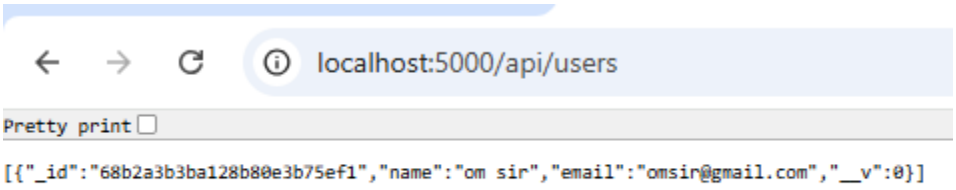
□ Final Steps

- Run backend:

  ```
  node index.js
  ```

For Output Open Browser:-

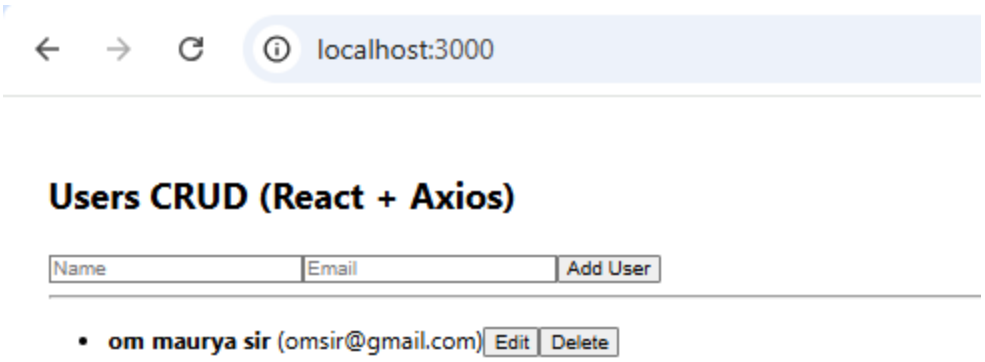
http://localhost:5000/api/users



- Run frontend:

  ```
  npm start
  ```

  output:-



Make sure MongoDB is running locally on port `27017`.

□ Result

- A simple React frontend to **create and list users**
- A Node/Express backend connected to **MongoDB**
- Communication via REST API

Absolutely! Let's break down **each line of code** from both the **backend (Express + MongoDB)** and **frontend (React)** so you can fully understand what's happening.

Here's a detailed line-by-line explanation of your **`backend/index.js`** code for an Express + MongoDB-based API:

□ 1. Import Required Packages

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

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

- `express`: Web framework for Node.js. Helps define routes and handle HTTP requests/responses.
- `mongoose`: ODM (Object Data Modeling) library for MongoDB. It provides schema-based modeling for MongoDB data.
- `cors`: Middleware to allow Cross-Origin Resource Sharing. Necessary if your frontend runs on a different port (e.g., React on `localhost:3000` and backend on `localhost:5000`).

□ 2. Initialize Express App

```
const app = express();
```

Creates an Express application instance so you can define routes, middleware, etc.

□ 3. Middleware Setup

```
app.use(cors());
app.use(express.json());
```

- `app.use(cors())`: Allows requests from other domains (like React frontend).
- `app.use(express.json())`: Parses incoming JSON request bodies. Required for reading `req.body` in POST/PUT requests.

□ 4. Connect to MongoDB

```
mongoose.connect('mongodb://localhost:27017/mydb', {
 useNewUrlParser: true,
 useUnifiedTopology: true,
})
.then(() => console.log('MongoDB connected'))
.catch(err => console.error(err));
```

- Connects to a MongoDB instance running locally (`mongodb://localhost:27017/mydb`).

- `useNewUrlParser` and `useUnifiedTopology` are options to avoid warnings in Mongoose.
- `.then()` logs success; `.catch()` logs errors.

□ 5. Define a Mongoose Schema & Model

```
const UserSchema = new mongoose.Schema({
 name: String,
 email: String,
});
```

- Defines a schema (`UserSchema`) that tells MongoDB what kind of structure a user document will have.
- Here, a user has a `name` and `email`, both of type `String`.

```
const User = mongoose.model('User', UserSchema);
```

- Creates a Mongoose model (`User`) for interacting with the `users` collection in MongoDB using the defined schema.

□ 6. GET Route – Fetch All Users

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

- `GET /api/users`: Retrieves all users from the database.
- `User.find()` fetches all documents in the `users` collection.
- `res.json(users)`: Sends them back to the client as JSON.

□ 7. POST Route – Create a New User

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

- `POST /api/users`: Creates a new user.

- `req.body` contains the JSON data sent from the client (e.g., `{ name: "Alice", email: "alice@example.com" }`).
- `new User(...)`: Creates a new user document.
- `user.save()`: Saves it to MongoDB.
- `res.json(user)`: Sends back the newly created user.

□ 8. PUT Route – Update an Existing User

```
app.put('/api/users/:id', async (req, res) => {
  try {
    const updatedUser = await User.findByIdAndUpdate(
      req.params.id,      // The user's ID from the URL
      req.body,           // Updated fields (name/email)
      { new: true }       // Option: return the updated document
    );
    res.json(updatedUser);
  } catch (err) {
    res.status(500).json({ error: 'Failed to update user' });
  }
});
```

- `PUT /api/users/:id`: Updates a user with the given `id`.
- `User.findByIdAndUpdate(...)`: Finds and updates the document.
- `{ new: true }`: Ensures the updated document is returned.
- On success, it sends the updated user back; on error, returns a 500 status.

□ 9. DELETE Route – Remove a User

```
app.delete('/api/users/:id', async (req, res) => {
  try {
    await User.findByIdAndDelete(req.params.id);
    res.json({ message: 'User deleted successfully' });
  } catch (err) {
    res.status(500).json({ error: 'Failed to delete user' });
  }
});
```

- `DELETE /api/users/:id`: Deletes a user by ID.
- `User.findByIdAndDelete(...)`: Finds and removes the user.
- On success, returns a confirmation message.

□ 10. Start the Server

```
app.listen(5000, () => {
 console.log('Server running on http://localhost:5000');
});
```

- Starts the Express server on **port 5000**.
- Logs a message to the console once it's running.

□ Summary of Routes

HTTP Method	Route	Purpose
GET	`/api/users`	Fetch all users
POST	`/api/users`	Add new user
PUT	`/api/users/:id`	Update existing user
DELETE	`/api/users/:id`	Delete user

Front End React Js Code Explanation:-

Here's a **line-by-line detailed explanation** of your React code that implements a full **CRUD interface** (Create, Read, Update, Delete) using **Axios** to connect to a backend API (`http://localhost:5000/api/users`).

□ IMPORT STATEMENTS

```
import React, { useEffect, useState } from 'react';
```

- `React`: Core React library.
- `useState`: React Hook for creating state variables.
- `useEffect`: React Hook to run side effects (like fetching data when component loads).

```
import axios from 'axios';
```

- Imports `axios`, a promise-based HTTP client to communicate with your Express backend.

□ FUNCTION COMPONENT

```
function App() {
```

- Defines a functional React component named `App`.

□ STATE VARIABLES

```
  const [users, setUsers] = useState([]);
```

- `users`: Stores an array of all user objects from the database.
- `setUsers`: Function to update the `users` state.

```
  const [form, setForm] = useState({ name: '', email: '' });
```

- `form`: Stores the form input values for creating/updating a user.
- `setForm`: Updates the `form` state.

```
  const [editingUserId, setEditingUserId] = useState(null);
```

- `editingUserId`: Holds the ID of the user being edited.
- If `null`, the form is in "Create" mode; if set, it's in "Update" mode.

□ API Endpoint

```
  const API_URL = 'http://localhost:5000/api/users';
```

- Base URL to communicate with your backend Express server's `/api/users` endpoint.

□ FETCH USERS ON COMPONENT MOUNT

```
useEffect(() => {
  fetchUsers();
}, []);
```

- `useEffect(...)`: Runs when the component mounts (only once due to the empty dependency array `[]`).
- Calls `fetchUsers()` to load all users from the backend.

□ GET USERS FROM API

```
const fetchUsers = async () => {
  try {
    const res = await axios.get(API_URL);
    setUsers(res.data);
  } catch (err) {
    console.error('Error fetching users:', err);
  }
};
```

- Sends a GET request to fetch all users.
- On success, updates the `users` state.
- On error, logs the error to the console.

□ HANDLE FORM INPUTS

```
const handleInputChange = (e) => {
  const { name, value } = e.target;
  setForm(prev => ({ ...prev, [name]: value }));
};
```

- Handles changes in the form input fields (`name`, `email`).
- Uses dynamic property names to update either field in the `form` object.

□ CREATE USER

```
const createUser = async () => {
  try {
    await axios.post(API_URL, form);
    fetchUsers();
  } catch (err) {
    console.error('Error creating user:', err);
  }
```

```
};
```

- Sends a POST request to create a new user with data from the form.
- Refreshes the user list after successful creation.

□ UPDATE USER

```
const updateUser = async () => {
  try {
    await axios.put(`${API_URL}/${editingUserId}`, form);
    fetchUsers();
  } catch (err) {
    console.error('Error updating user:', err);
  }
};
```

- Sends a PUT request to update an existing user (by ID).
- Refreshes the user list after the update.

□ HANDLE FORM SUBMISSION

```
const handleSubmit = async (e) => {
  e.preventDefault();
  if (editingUserId) {
    await updateUser();
  } else {
    await createUser();
  }
  setForm({ name: '', email: '' });
  setEditingUserId(null);
};
```

- Handles the form submit event.
- Prevents page refresh with `e.preventDefault()`.
- Decides whether to create or update a user based on `editingUserId`.
- Resets the form and editing state after submission.

✏□ HANDLE EDIT BUTTON

```
const handleEdit = (user) => {
  setForm({ name: user.name, email: user.email });
  setEditingUserId(user._id);
};
```

- Fills the form with existing user data when "Edit" is clicked.
- Sets `editingUserId` so the form goes into "Update" mode.

□ HANDLE DELETE BUTTON WITH CONFIRMATION

```
  const handleDelete = async (id) => {
    const confirmDelete = window.confirm('Are you sure you want to delete
this user?');
    if (!confirmDelete) return;

    try {
      await axios.delete(`${API_URL}/${id}`);
      fetchUsers();
    } catch (err) {
      console.error('Error deleting user:', err);
    }
  };
```

- Asks the user for confirmation before deleting.
- If confirmed, sends DELETE request.
- Refreshes user list afterward.

□ RENDER JSX UI

```
  return (
    <div style={{ padding: 30 }}>
      <h2>Users CRUD (React + Axios)</h2>
```

- Starts rendering the UI.
- Adds padding and heading.

□ Form Component

```
      <form onSubmit={handleSubmit}>
        <input
          type="text"
          name="name"
          placeholder="Name"
          value={form.name}
          onChange={handleInputChange}
          required
        />
        <input
          type="email"
          name="email"
```

```
        placeholder="Email"
        value={form.email}
        onChange={handleInputChange}
        required
      />
      <button type="submit">
        {editingUserId ? 'Update' : 'Add'} User
      </button>
    </form>
```

- Shows two inputs and a button.
- Populates form with state values.
- Dynamically shows `Add User` or `Update User` depending on whether editing.

□ Display User List

```
    <hr />
    <ul>
      {users.map(user => (
        <li key={user._id}>
          <strong>{user.name}</strong> ({user.email})
          <button onClick={() => handleEdit(user)}>Edit</button>
          <button onClick={() => handleDelete(user._id)}>Delete</button>
        </li>
      ))}
    </ul>
```

- Loops through `users` array and displays each user.
- Shows **Edit** and **Delete** buttons for each.
- Buttons call corresponding handlers with user data.

□ Export the Component

```
export default App;
```

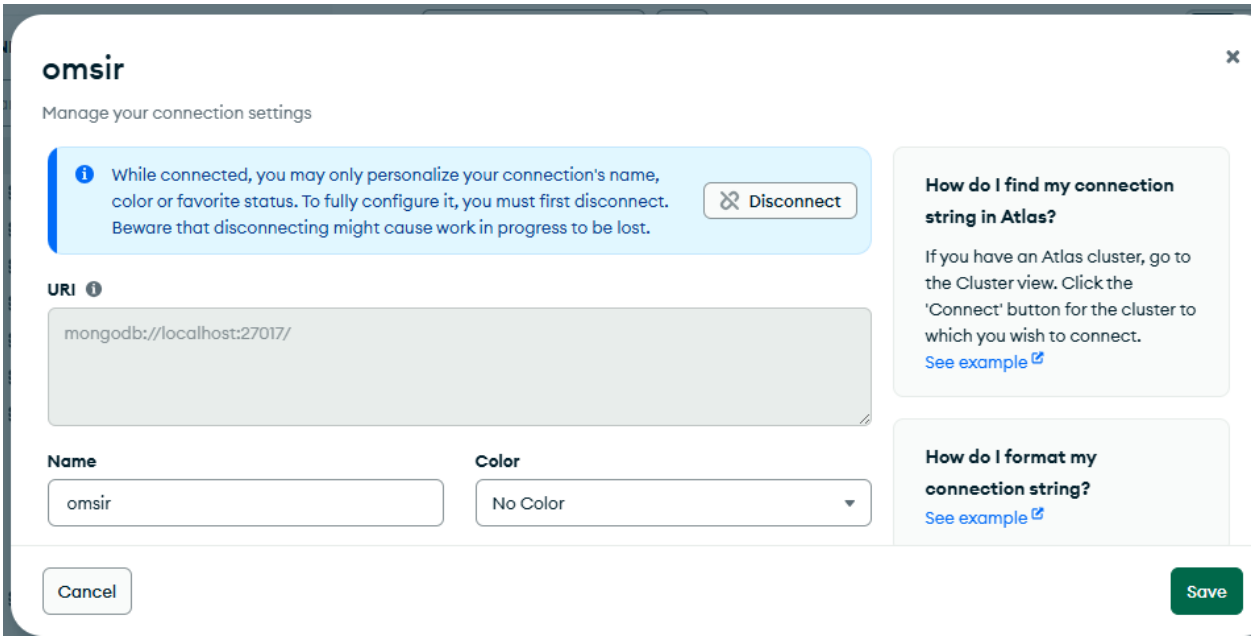
- Exports the component so it can be imported in `index.js` or other files.

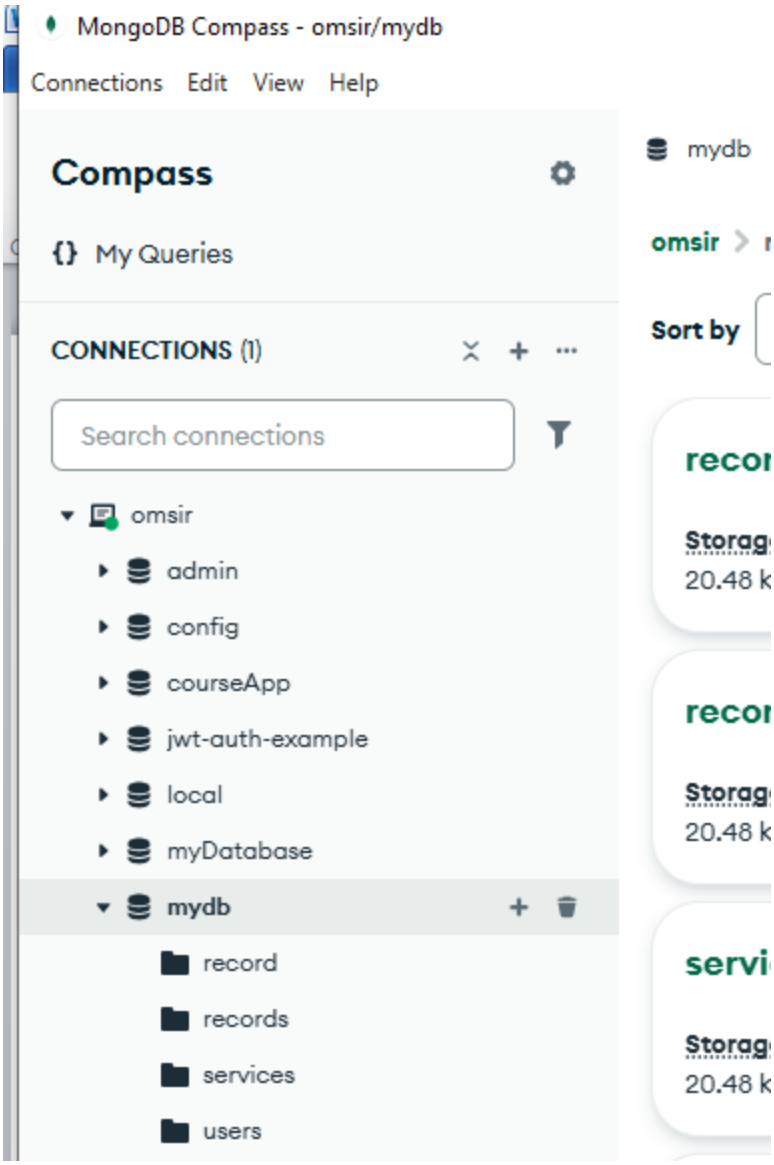
□ Summary

Feature	Code Used	Purpose
Fetch users	`fetchUsers()`	GET all users from backend
Add user	`createUser()`	POST new user
Edit user	`updateUser()`	PUT request
Delete user	`handleDelete()`	DELETE request
Edit form	`handleEdit()`	Load user data into form
Form input	`handleInputChange()`	Sync form state with input
Submit form	`handleSubmit()`	Call create or update depending on context

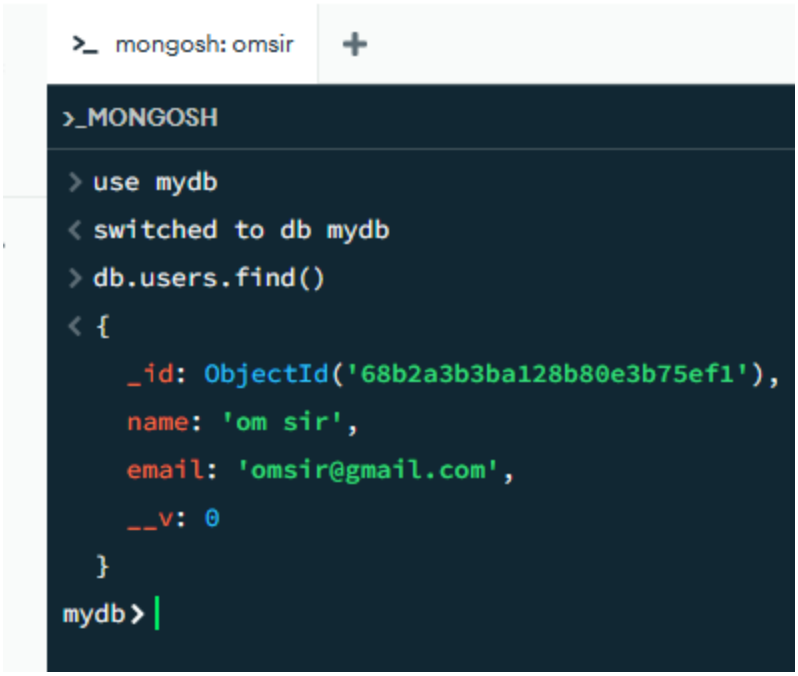
MongoDB :-

Connection settings:-





And Display data using mongoDB shell :-


mongosh: omsir
>_MONGOSH
> use mydb
< switched to db mydb
> db.users.find()
< {
_id: ObjectId('68b2a3b3ba128b80e3b75ef1'),
name: 'om sir',
email: 'omsir@gmail.com',
__v: 0
}
mydb>