MONGODB & JWT LAB MANUAL

Objective

- Connect to MongoDB
- To learn usage of JWT

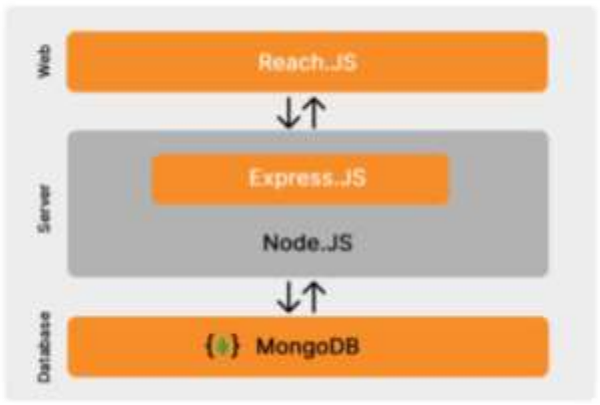
Software

- VS Code
- Mongodb Atlas

Theory

MongoDB is a source-available, cross-platform, document-oriented database program. Classified as a NoSQL database product, MongoDB utilizes JSON-like documents with optional schemas.

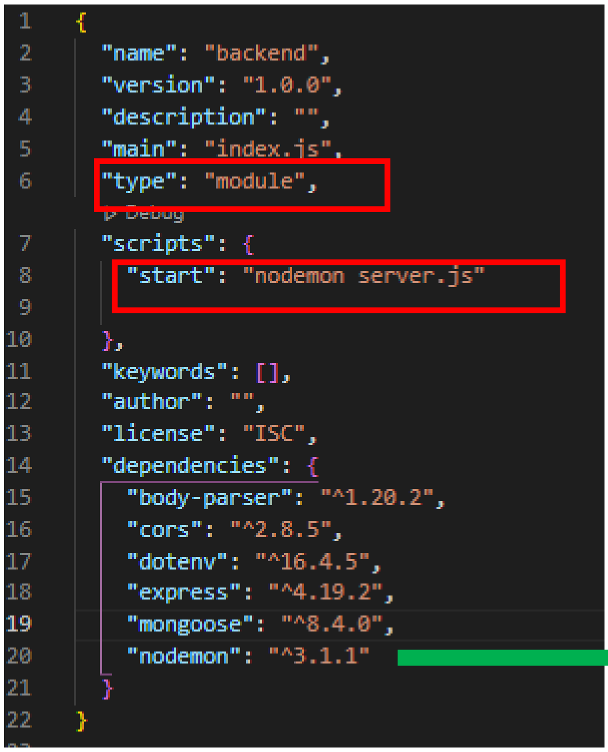
MySQL	MongoDB
Table	Collection
Row	Document
Column	Field
Joins	Embedded documents, linking



USING MONGODB ATLAS:

Step 1: Set Up Backend with Node.js and Express

1. **Initialize a Node.js Project**:
   Create folder
   mkdir foldername
   cd foldername
   npm init -y
2. **Install Required Packages**:
   npm install express mongoose cors



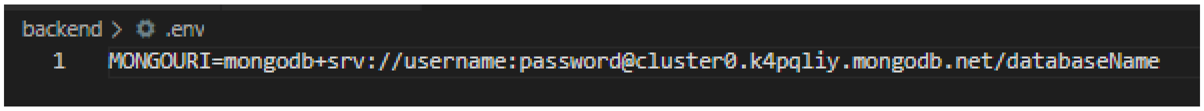
```
{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  ▷ Debug
  "scripts": {
    "start": "nodemon server.js"

  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.20.2",
    "cors": "^2.8.5",
    "dotenv": "^16.4.5",
    "express": "^4.19.2",
    "mongoose": "^8.4.0",
    "nodemon": "^3.1.1"
  }
}
```

npm install dotenv

npm install -g nodemon

To automatically restart your server whenever you make changes to your code, you can use a tool like **nodemon**. **nodemon** is a utility that monitors changes in your source files and automatically restarts your server.

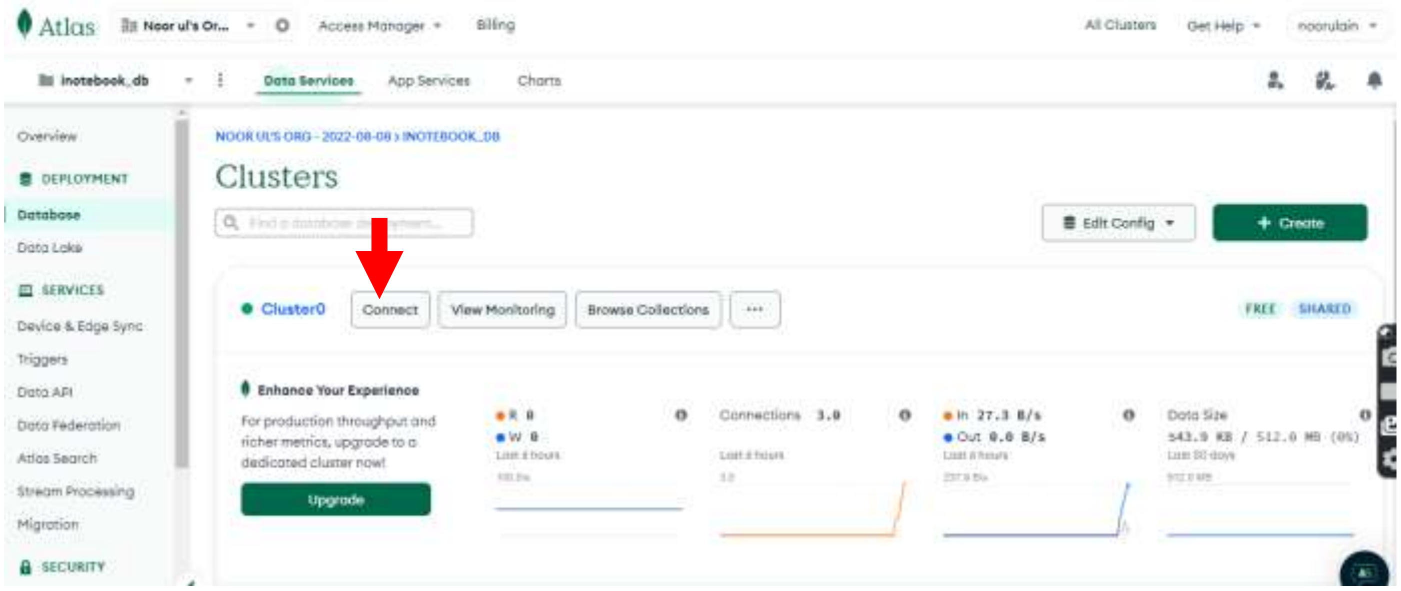
Using **dotenv** is a great way to manage sensitive information like database credentials by storing them in an environment file rather than hardcoding them in your source code.



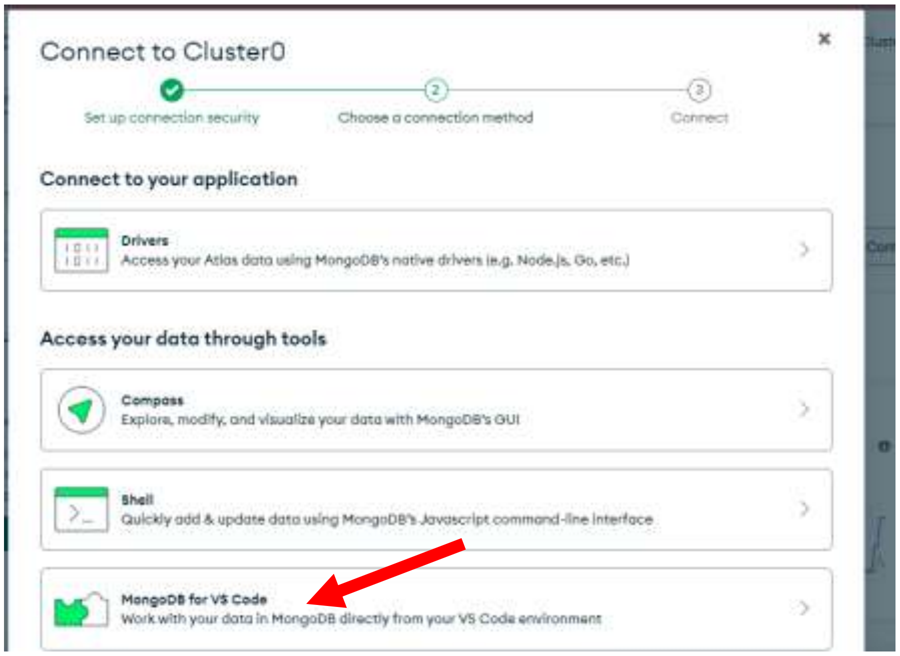
backend > .env

```
MONGOURI=mongodb+srv://username:password@cluster0.k4pqliy.mongodb.net/databaseName
```

How to get a connection string?



For production throughput and richer metrics, upgrade to a dedicated cluster now!



Connect to your application

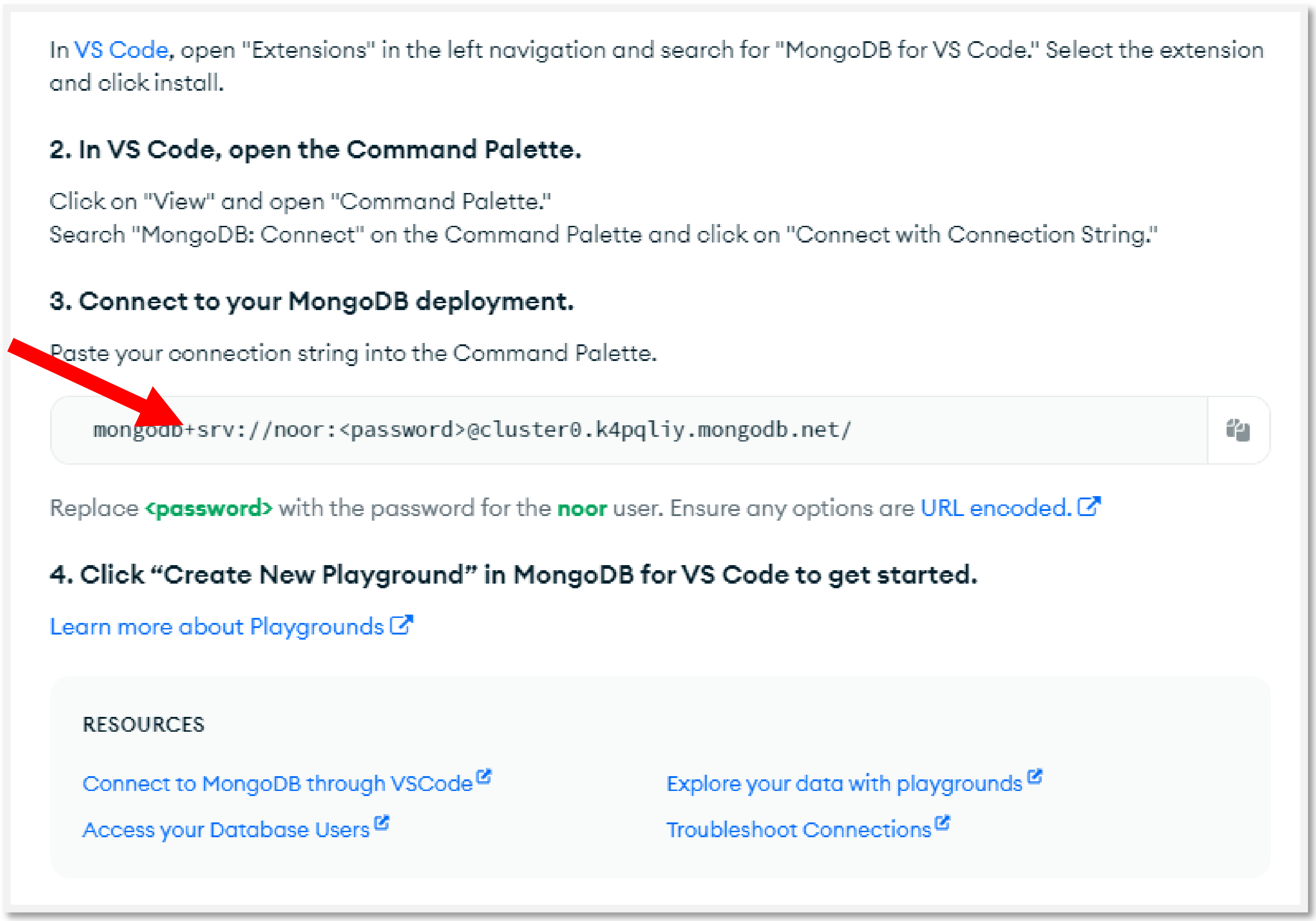
Drivers
Access your Atlas data using MongoDB's native drivers (e.g. Node.js, Go, etc.)

Access your data through tools

Compass
Explore, modify, and visualize your data with MongoDB's GUI

Shell
Quickly add & update data using MongoDB's Javascript command-line interface

MongoDB for VS Code
Work with your data in MongoDB directly from your VS Code environment



In VS Code, open "Extensions" in the left navigation and search for "MongoDB for VS Code." Select the extension and click install.

2. In VS Code, open the Command Palette.

Click on "View" and open "Command Palette."
Search "MongoDB: Connect" on the Command Palette and click on "Connect with Connection String."

3. Connect to your MongoDB deployment.

Paste your connection string into the Command Palette.

```
mongodb+srv://noor:<password>@cluster0.k4pqliy.mongodb.net/
```

Replace **<password>** with the password for the **noor** user. Ensure any options are URL encoded.

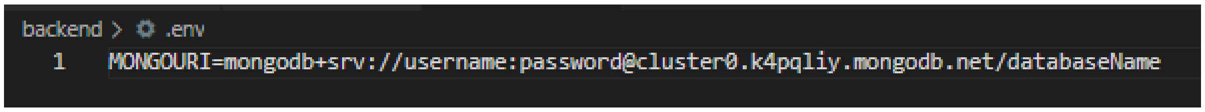
4. Click "Create New Playground" in MongoDB for VS Code to get started.

Learn more about Playgrounds

RESOURCES

Connect to MongoDB through VSCode
Explore your data with playgrounds
Access your Database Users
Troubleshoot Connections

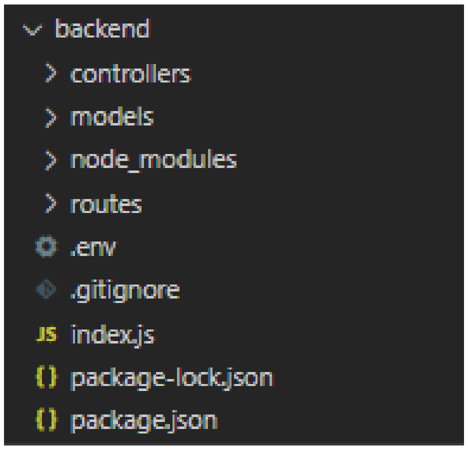
Paste the above connection string in .env file:

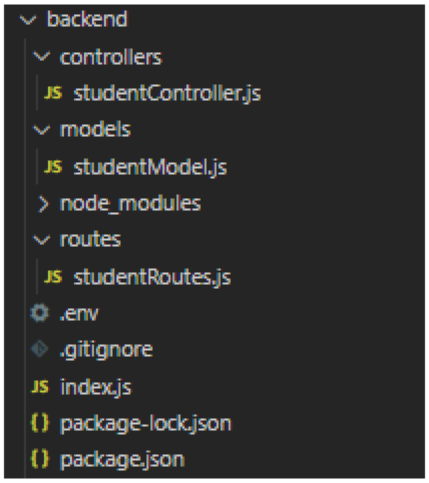


backend > .env

```
MONGOURI=mongodb+srv://username:password@cluster0.k4pqliy.mongodb.net/databaseName
```

Create Server & Backend Logic:
FILE STRUCTURE/HIERARCHY:


```
backend
  controllers
  models
  node_modules
  routes
  .env
  .gitignore
  index.js
  package-lock.json
  package.json
```


```
backend
  controllers
    studentController.js
  models
    studentModel.js
  node_modules
  routes
    studentRoutes.js
  .env
  .gitignore
  index.js
  package-lock.json
  package.json
```

Index.js (entry point/root file):

```
import express from 'express';
import mongoose from 'mongoose';
import cors from 'cors';
import studentRoutes from './routes/studentRoutes.js';
import dotenv from 'dotenv';
// Load environment variables from .env file
dotenv.config();
const app = express();
const port = 5000;

app.use(express.json());
app.use(cors());
app.use('/api', studentRoutes);

const dbURI = process.env.MONGOURI;
mongoose.connect(dbURI, {}).then(() => {
  console.log("Connection successful");
}).catch((error) => {
  console.error("Connection error", error);
});

app.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});
```

Separation of Concerns: The MVC pattern promotes a clean separation of concerns:

- **Models** manage the data and the logic to retrieve and manipulate it.
- **Views** handle the presentation layer, defining how the data is displayed.
- **Controllers** act as an intermediary, processing incoming requests, interacting with models to fetch or modify data, and then choosing the appropriate view to render the response.

1. **studentModel.js:**

```
import mongoose from 'mongoose';

const studentSchema = new mongoose.Schema({
  name: String,
  rollNumber: Number,
});

const Student = mongoose.model('Student', studentSchema);

export default Student;
```

2. StudentController.js:
Import Model to your controller file:

```
import Student from '../models/studentModel.js';
```

By importing models into controllers, you can use them to perform CRUD (Create, Read, Update, Delete) operations on the data.

```
export const createStudent = async (req, res) => {
  const student = new Student(req.body);
  try {
    await student.save();
    res.status(201).send(student);
  } catch (error) {
    res.status(400).send(error);
  }
};
```

This function **createStudent** asynchronously creates a new student record from the request body, saves it to the database, and responds with the created student and a status of 201 on success. If there's an error during saving, it responds with the error and a status of 400. It uses a try-catch block to handle the asynchronous operation and potential errors.

Use HTTP status code 200 for successful requests that retrieve or update a resource. Use HTTP status code 201 for successful requests that create a new resource on the server. The HyperText Transfer Protocol (HTTP) 400 Bad Request response status code indicates that the server cannot or will not process the request due to something that is perceived to be a client error (for example, malformed request syntax, invalid request message framing, or deceptive request routing).

400 Bad Request: This status code is used when the client sends invalid data. In this case, if the student data provided in the request body is incorrect or fails validation, a 400 error indicates that the client needs to fix the request data.

```
export const getStudents = async (req, res) => {
  try {
    const students = await Student.find();
    res.send(students);
  } catch (error) {
    res.status(500).send(error);
  }
};
```

500 Internal Server Error: This status code is used when something goes wrong on the server side. In this case, if there is an issue retrieving the students from the database (e.g., a database connection issue or an unexpected error in the server logic), a 500 error indicates that the server encountered an unexpected condition that prevented it from fulfilling the request.

```
export const updateStudent = async (req, res) => {
  try {
    const student = await Student.findByIdAndUpdate(req.params.id, req.body, {
new: true, runValidators: true });
    if (!student) {
      return res.status(404).send();
    }
    res.send(student);
  } catch (error) {
    res.status(400).send(error);
  }
};
```

Student.findByIdAndUpdate(req.params.id, req.body, { new: true, runValidators: true }): This is a Mongoose method used to find a document by its ID and update it with new data.

- **req.params.id**: This is the ID of the student to be updated, which is extracted from the request parameters.

- **req.body**: This contains the new data for the student, extracted from the request body.
- **{ new: true, runValidators: true }**: These are options passed to the method:
    - **new: true**: This option returns the updated document rather than the original document.
    - **runValidators: true**: This option ensures that the update operation runs validation checks on the new data according to the model's schema.

- The method attempts to find a student by their ID and update their data with the information provided in the request body.
- If a student with the specified ID is found and updated successfully, the updated document is returned.
- If no student is found, the function sends a 404 status.
- If there is a validation error or any other issue with the update, a 400 status is sent.

```
export const deleteStudent = async (req, res) => {
  try {
    const student = await Student.findByIdAndDelete(req.params.id);
    if (!student) {
      return res.status(404).send();
    }
    res.send(student);
  } catch (error) {
    res.status(500).send(error);
  }
};
```

```
export const getStudentByRollNumber = async (req, res) => {
  try {
    const rollNumber = req.params.rollNumber;
    const student = await Student.findOne({ rollNumber: rollNumber });
    if (!student) {
      return res.status(404).send({ error: 'Student not found' });
    }
    res.send(student); // Return the entire student object
  } catch (error) {
    res.status(500).send({ error: error.message });
  }
};
```

3. **StudentRoutes.js:**

```
import express from 'express';
import {
  createStudent,
  getStudents,
  updateStudent,
  deleteStudent,
  getStudentByRollNumber
} from '../controllers/studentController.js';

const router = express.Router();

router.post('/students', createStudent);
router.get('/students', getStudents);
router.get('/students/rollNumber/:rollNumber', getStudentByRollNumber);
router.put('/students/:id', updateStudent);
router.delete('/students/:id', deleteStudent);

export default router;
```

JWT (JSON Web Tokens)

Install Required Packages
npm install jsonwebtoken bcrypt

What is JWT?

JWT (JSON Web Token) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature structure or as the plaintext of a JSON Web Encryption structure, enabling the claims to be digitally signed or integrity protected with a message authentication code.

What is bcrypt?

bcrypt is a popular password-hashing function designed to securely store passwords by hashing them in a way that is computationally intensive, making it more resistant to brute-force attacks.

JWT Structure

A JWT consists of three parts, separated by dots (.):

1. **Header**: Contains metadata about the token, typically the type of token (JWT) and the signing algorithm (e.g., HMAC SHA256).

2. **Payload**: Contains the claims (statements about an entity, typically the user, and additional data).
3. **Signature**: Used to verify that the sender of the JWT is who it claims to be and to ensure that the message wasn't changed along the way.

The structure looks like this:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Implementation Steps

Step 1: Set Up Environment Variables

Create a `.env` file and add a secret key for signing the tokens:

```
JWT_SECRET=your_secret_key
```

Step 2: Create the Server

Open `index.js` and set up a basic Express server:

```
const express = require('express');
const jwt = require('jsonwebtoken');
const cors = require('cors');
const bcrypt = require('bcrypt');
require('dotenv').config();

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

const PORT = process.env.PORT || 5000;

// Dummy user data (normally you would use a database)
let users = [];

// Endpoint to register a new user
app.post('/register', async (req, res) => {
    const { username, password } = req.body;
    const existingUser = users.find(u => u.username === username);
    if (existingUser) return res.status(400).send('User already exists');

    // Hash password before storing
    const hashedPassword = await bcrypt.hash(password, 10);
    const newUser = { id: Date.now(), username, password: hashedPassword };
    users.push(newUser);
    res.send('User registered successfully');
});

// Endpoint to log in and get a token
app.post('/login', async (req, res) => {
    const { username, password } = req.body;
    const user = users.find(u => u.username === username);

    // Check if user exists and verify password with bcrypt
    if (user && await bcrypt.compare(password, user.password)) {
        const token = jwt.sign({ id: user.id }, process.env.JWT_SECRET, {
expiresIn: '1h' });
        return res.json({ token });
    }
    return res.status(401).send('Invalid credentials');
});

// Middleware to verify token
const verifyToken = (req, res, next) => {
    const token = req.headers['authorization'];
    if (!token) return res.sendStatus(403);

    jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
        if (err) return res.sendStatus(403);
        req.user = user;
        next();
    });
};

// Protected route
app.get('/protected', verifyToken, (req, res) => {
    res.send('This is a protected route. Welcome, user ' + req.user.id);
});

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

Step 3: Testing the Implementation

1. **Start the Server**
   - In your terminal, run:

     ```
     node index.js or npm start if you configured nodemon
     ```

2. **Test the Login Endpoint**
   - Use Postman or ThunderClient to send a POST request to `http://localhost:5000/login` with the following JSON body:

     ```
     {
       "username": "user1",
       "password": "password1"
     }
     ```

   - You will receive a JWT token in response.
3. **Access the Protected Route**
   - Send a GET request to `http://localhost:5000/protected` with the token in the Authorization header:

     ```
     Authorization: Bearer <your_token_here>
     ```

   - If the token is valid, you should receive a message confirming access to the protected route.