Week10: For the above experiment-06 application create authorized end points using JWT (JSON Web Token).

Aim:

- To develop an Express web application that performs CRUD operations (Create, Read, Update, Delete) on student data through a REST API, and to secure the endpoints using JSON Web Token (JWT) for authorization. This ensures that only authorized users can interact with the API.

- To set up MySQL with Express and adjust the project to perform CRUD operations with a MySQL database.

Prerequisites:

- **Node.js** and **npm**
- **MySQL Database**: Set up a MySQL database with a students table.
- **Postman**: Used to test the API.

Project Setup:

1. **Initialize a new Node.js project:**
   mkdir student-api
   cd student-api
   npm init -y
   npm install express mysql2 jsonwebtoken bcryptjs dotenv

- **express**: A Node.js framework for building web applications and APIs.
- **mysql2**: A MySQL client for Node.js, compatible with Sequelize.
- **jsonwebtoken**: A library to create and verify JSON Web Tokens (JWT) for authentication.
- **bcryptjs**: A JavaScript-only version of bcrypt for hashing passwords.
- **dotenv**: A library to load environment variables from a .env file.

Folder Structure:

```
student-api/
├── config/
│   └── db.js
├── models/
│   ├── Student.js
│   └── User.js
├── routes/
│   ├── auth.js
│   └── students.js
├── .env
└── server.js
```

Step 1: Environment Configuration
Create a .env file:

```
PORT=5000
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=
DB_NAME=cet_lab
JWT_SECRET=d8c2f40f54740f0828286c49cdea55161f357e75944d3ec07136952def67a0fe
```

Step 2: Database Configuration (config/db.js)
Set up the MySQL connection in config/db.js:

```
   // config/db.js
const { Sequelize } = require('sequelize');

const sequelize = new Sequelize(
  process.env.DB_NAME,
  process.env.DB_USER,
  process.env.DB_PASSWORD,
  {
    host: process.env.DB_HOST,
    dialect: 'mysql',
  }
);

module.exports = sequelize;
```

Step 3: Define Models for CRUD Operations (models/StudentModel.js)

```
// models/Student.js
const { DataTypes } = require('sequelize');
const sequelize = require('../config/db');

const Student = sequelize.define('Student', {
  name: {
    type: DataTypes.STRING,
    allowNull: false,
  },
  age: {
    type: DataTypes.INTEGER,
```

```
    allowNull: false,
  },
  grade: {
    type: DataTypes.STRING,
    allowNull: false,
  },
});

module.exports = Student;
```

User.js
// models/User.js

```
const { DataTypes } = require('sequelize');
const sequelize = require('../config/db');

const User = sequelize.define('User', {
  username: {
    type: DataTypes.STRING,
    allowNull: false,
    unique: true,
  },
  password: {
    type: DataTypes.STRING,
    allowNull: false,
  },
});

module.exports = User;
```

Step 4 : Sync Database: Modify **server.js** to sync the database and establish the MySQL connection.

```
require('dotenv').config();
const express = require('express');
const sequelize = require('./config/db');
const authRoutes = require('./routes/auth');
const studentRoutes = require('./routes/students');

const app = express();
const PORT = process.env.PORT || 5000;

app.use(express.json());

// Sync Sequelize with MySQL
sequelize.sync()
  .then(() => console.log('MySQL Database connected and synced'))
  .catch((err) => console.error('Error connecting to MySQL:', err));

app.use('/api/auth', authRoutes);
```

```
app.use('/api/students', studentRoutes);

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

Step 5: Set Up Routes (routes/student.js and routes/auth.js)

```
// routes/students.js
const express = require('express');
const router = express.Router();
const Student = require('../models/Student');
const jwt = require('jsonwebtoken');

const authenticateToken = (req, res, next) => {
  const token = req.header('Authorization')?.split(' ')[1];
  if (!token) return res.status(401).json({ message: 'Access Denied' });

  try {
    const verified = jwt.verify(token, process.env.JWT_SECRET);
    req.user = verified;
    next();
  } catch (err) {
    res.status(400).json({ message: 'Invalid Token' });
  }
};

// Create a new student
router.post('/', authenticateToken, async (req, res) => {
  try {
    const { name, age, grade } = req.body;
    const student = await Student.create({ name, age, grade });
    res.json(student);
  } catch (err) {
    res.status(400).json({ message: err.message });
  }
});

// Get all students
router.get('/', authenticateToken, async (req, res) => {
  try {
    const students = await Student.findAll();
    res.json(students);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

// Get a student by ID
```

```
router.get('/:id', authenticateToken, async (req, res) => {
  try {
    const student = await Student.findByPk(req.params.id);
    if (!student) return res.status(404).json({ message: 'Student not found' });
    res.json(student);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

// Update a student by ID
router.put('/:id', authenticateToken, async (req, res) => {
  try {
    const { name, age, grade } = req.body;
    const [updated] = await Student.update(
      { name, age, grade },
      { where: { id: req.params.id } }
    );
    if (!updated) return res.status(404).json({ message: 'Student not found' });
    res.json({ message: 'Student updated successfully' });
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

// Delete a student by ID
router.delete('/:id', authenticateToken, async (req, res) => {
  try {
    const deleted = await Student.destroy({ where: { id: req.params.id } });
    if (!deleted) return res.status(404).json({ message: 'Student not found' });
    res.json({ message: 'Student deleted successfully' });
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

module.exports = router;
```

Implement Authentication Logic in authRoutes

```
// routes/auth.js
const express = require('express');
const router = express.Router();
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const User = require('../models/User'); // Define a User model similar to Student

// Register a new user
router.post('/register', async (req, res) => {
  try {
    const { username, password } = req.body;
```

```
    const hashedPassword = await bcrypt.hash(password, 10);
    const user = await User.create({ username, password: hashedPassword });
    res.json({ message: 'User registered successfully' });
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

// Login user
router.post('/login', async (req, res) => {
  try {
    const { username, password } = req.body;
    const user = await User.findOne({ where: { username } });
    if (!user || !await bcrypt.compare(password, user.password)) {
      return res.status(400).json({ message: 'Invalid credentials' });
    }
    const token = jwt.sign({ id: user.id }, process.env.JWT_SECRET, { expiresIn: '1h' });
    res.json({ token });
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

module.exports = router;
```

6.Start the Server:

- In your terminal, navigate to the project's root directory.
- In XAMP, Start the MYSQL and Apache server, then enter in cmd as
  node server.js

7.. Steps to Test Endpoints in Postman

1. Open Postman

- Launch the Postman application on your computer.

2. Create a New Collection (Optional)

- You can create a collection to organize your requests. Click on the "Collections" tab on the left sidebar, then click "New Collection" and give it a name (e.g., "Student API").

3. Register a New User (POST /api/auth/register)

- **Create Request**:
  - Click on "New" > "Request".
  - Name the request (e.g., "Register User") and save it in your collection.
- **Set Method and URL**:
  - Set the request method to POST.
  - Enter the URL: http://localhost:5000/api/auth/register.
- **Add Body**:
  - Go to the **Body** tab.

    - Select raw and choose JSON from the dropdown.
    - Enter the following JSON data:

```
{
"username": "testuser",
"password": "testpassword"
}
```

- **Send Request**:
    - Click the “Send” button.
    - You should receive a response indicating that the user was registered successfully.

4. Login User to Obtain JWT Token (POST /api/auth/login)

- **Create Login Request**:
    - Click on “New” > “Request”.
    - Name the request (e.g., "Login User") and save it.
- **Set Method and URL**:
    - Set the method to POST.
    - Enter the URL: http://localhost:5000/api/auth/login.
- **Add Body**:
    - Go to the **Body** tab.
    - Select raw and choose JSON.
    - Enter the following JSON data:

```
{
 "username": "testuser",
 "password": "testpassword"
}
```

- **Send Request**:
    - Click the “Send” button.
    - You should receive a response with a JWT token:

```
{
 "token": "your_jwt_token_here"
}
```

    - Copy this token; you will use it for the next steps.

5. Create a New Student (POST /api/students)

- **Create Student Request**:
    - Click on “New” > “Request”.
    - Name the request (e.g., "Create Student") and save it.
- **Set Method and URL**:
    - Set the method to POST.
    - Enter the URL: http://localhost:5000/api/students.
- **Add Authorization**:
    - Go to the **Authorization** tab.
    - Choose Bearer Token from the dropdown.
    - Paste the JWT token you copied earlier into the token field.
- **Add Body**:
    - Go to the **Body** tab.
    - Select raw and choose JSON.
    - Enter the following JSON data for the student:

```
{
 "name": "John Doe",
```

```
 "age": 20,
 "grade": "A"
}
```

- **Send Request**:
  - Click the “Send” button.
  - You should receive a response with the created student's details.

6. Get All Students (GET /api/students)

- **Create Get Students Request**:
  - Click on “New” > “Request”.
  - Name the request (e.g., "Get All Students") and save it.
- **Set Method and URL**:
  - Set the method to GET.
  - Enter the URL: http://localhost:5000/api/students.
- **Add Authorization**:
  - Go to the **Authorization** tab and use the same JWT token as before.
- **Send Request**:
  - Click the “Send” button.
  - You should receive a list of all students in the response.

7. Get a Student by ID (GET /api/students/:id)

- **Create Get Student Request**:
  - Click on “New” > “Request”.
  - Name the request (e.g., "Get Student by ID") and save it.
- **Set Method and URL**:
  - Set the method to GET.
  - Enter the URL: http://localhost:5000/api/students/1 (replace 1 with the actual ID of a student).
- **Add Authorization**:
  - Use the JWT token again in the **Authorization** tab.
- **Send Request**:
  - Click the “Send” button.
  - You should receive the details of the specified student.

8. Update a Student by ID (PUT /api/students/:id)

- **Create Update Student Request**:
  - Click on “New” > “Request”.
  - Name the request (e.g., "Update Student") and save it.
- **Set Method and URL**:
  - Set the method to PUT.
  - Enter the URL: http://localhost:5000/api/students/1 (replace 1 with the actual student ID).
- **Add Authorization**:
  - Use the JWT token in the **Authorization** tab.
- **Add Body**:
  - Go to the **Body** tab.
  - Select raw and choose JSON.
  - Enter updated student details:

```
{
 "name": "ramu",
 "age": 21,
 "grade": "B"
}
```

- **Send Request**:
    - Click the “Send” button.
    - You should receive a success message indicating the student was updated.

9. Delete a Student by ID (DELETE /api/students/:id)

- **Create Delete Student Request**:
    - Click on “New” > “Request”.
    - Name the request (e.g., "Delete Student") and save it.
- **Set Method and URL**:
    - Set the method to DELETE.
    - Enter the URL: http://localhost:5000/api/students/1 (replace 1 with the actual student ID).
- **Add Authorization**:
    - Use the JWT token in the **Authorization** tab.
- **Send Request**:
    - Click the “Send” button.
    - You should receive a confirmation message indicating the student was deleted successfully.