API's using Node

Submitted To

Laeeq Khan Niazi

Submitted By

Ammad Aslam 2021-CS-67

Department of Computer Science

University of Engineering and Technology, Lahore

Pakistan

Contents

1 Introduction

1.1 Description

Node.js is an open-source, server-side JavaScript runtime environment that allows developers to build scalable and high-performance applications. It is based on the V8 JavaScript engine and excels in asynchronous I/O operations. Node.js is commonly used for developing APIs, which are Application Programming Interfaces, enabling communication between different software systems, platforms, or services, facilitating data exchange and functionality access across the web. It is a popular choice for building APIs due to its non-blocking, event-driven architecture, making it efficient for handling concurrent requests and real-time applications.

2 Code for CRUD API's

2.1 Create API

The Create operation in a CRUD API is responsible for adding new records or entities to the database. To implement the Create API, you typically need to set up a POST endpoint, which receives data from the client, validates it, and then inserts the data into the database. Validation is crucial to ensure data integrity and security. After successful validation and insertion, a response with the newly created entity's details or an acknowledgment is sent back to the client.

```
async function createUser (req ,res) {
    try {
        const newuser = await User.create(req.body);
        res.status(201).json(newuser);
    }
    catch (error) {
        res.status(500).json({error : error})
    }
}
```

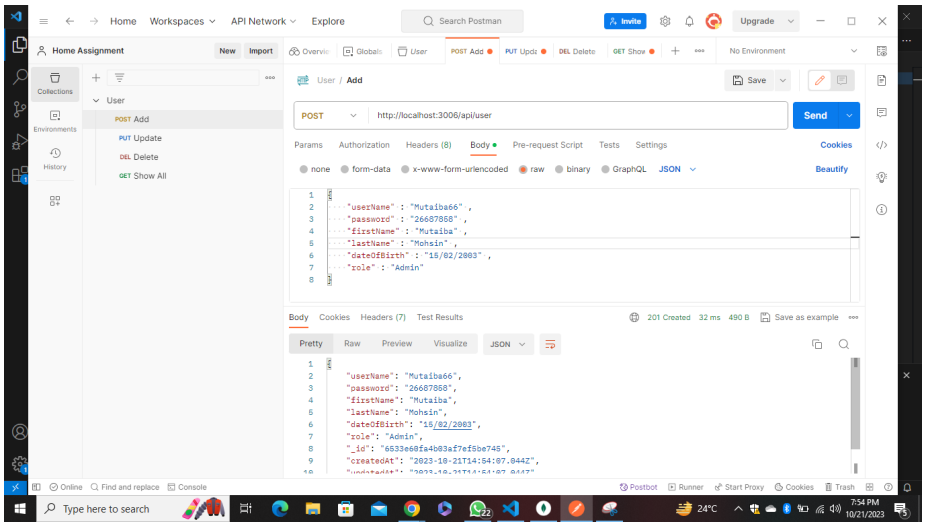

Figure 1: Adding User as Admin

2.2 Update API

The Update operation enables the modification of existing data in the database. Implementing the Update API usually requires a PUT or PATCH endpoint, where the client sends updated data along with a unique identifier (e.g., ID) for the record to be updated. The server processes the update request, validates the data, and then makes changes to the corresponding record in the database. Upon successful update, a response indicating the updated entity or a success message is sent back to the client.

```
async function updateUser (req , res) {
    try{
        const {id} = req.params;
        const updateUser = await User.findByIdAndUpdate(id,req.body,{new: true});
        res.json(updateUser);
    }
    catch (err) {
        res.status(500).json({error : err.message});
    }
}
```

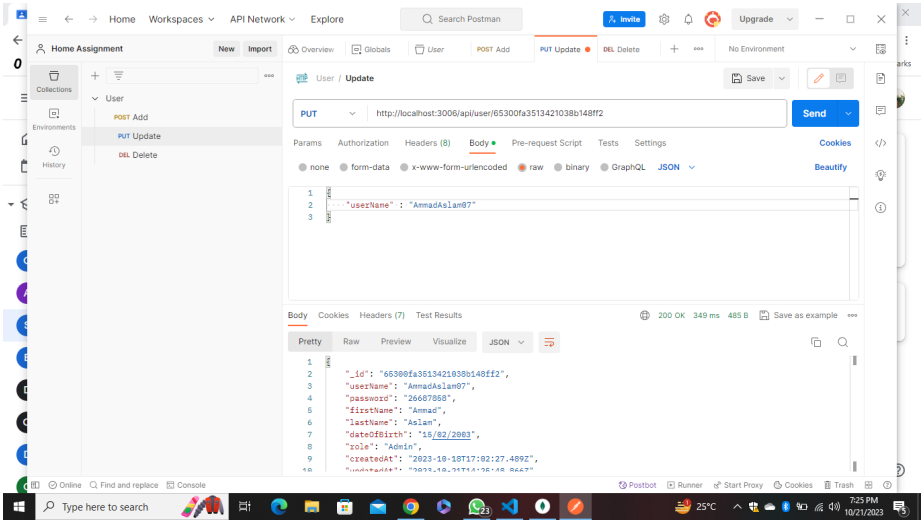

Figure 2: Before Update

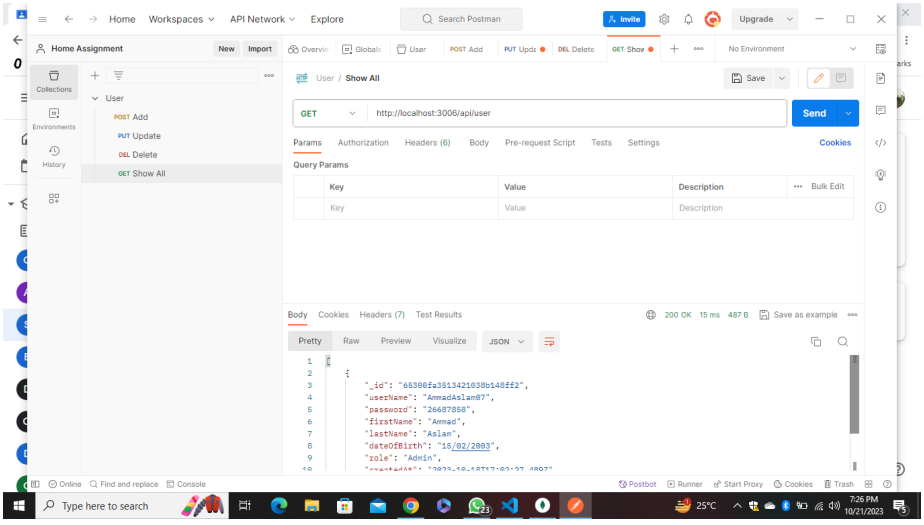

Figure 3: After Update

2.3 Delete API

The Delete operation is used to remove records from the database. It involves setting up a DELETE endpoint. The client specifies the record to be deleted, usually by providing its ID. The server processes the delete request, ensures that the record exists and is eligible for deletion, and then removes it from the

database. The response typically confirms the successful deletion or provides an acknowledgment.

```
async function deleteUser (req , res) {
    try {
        const {id} = req.params;
        await User.findByIdAndRemove(id);
        res.sendStatus(200).json({message : "Deleted Successfully"});
    }
    catch (err) {
        res.status(500).json({error : err.message});
    }
}
```

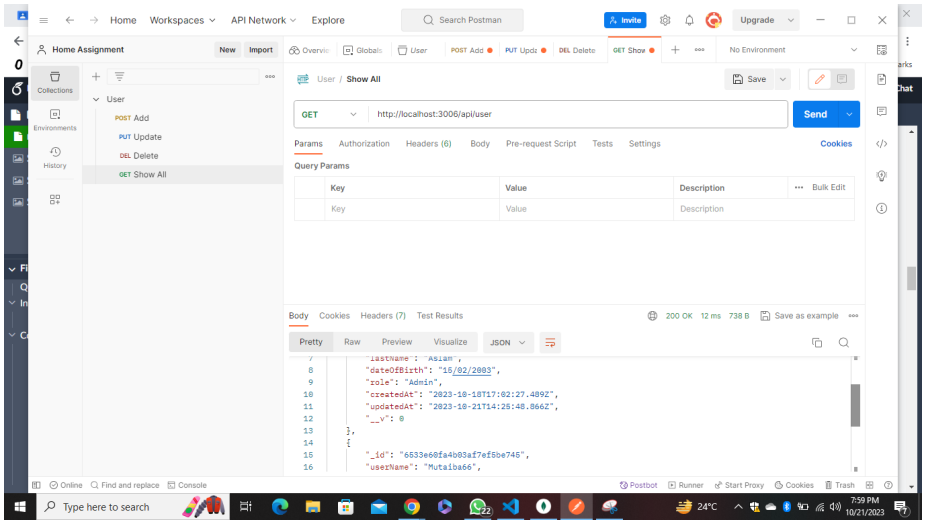

Figure 4: Before Delete

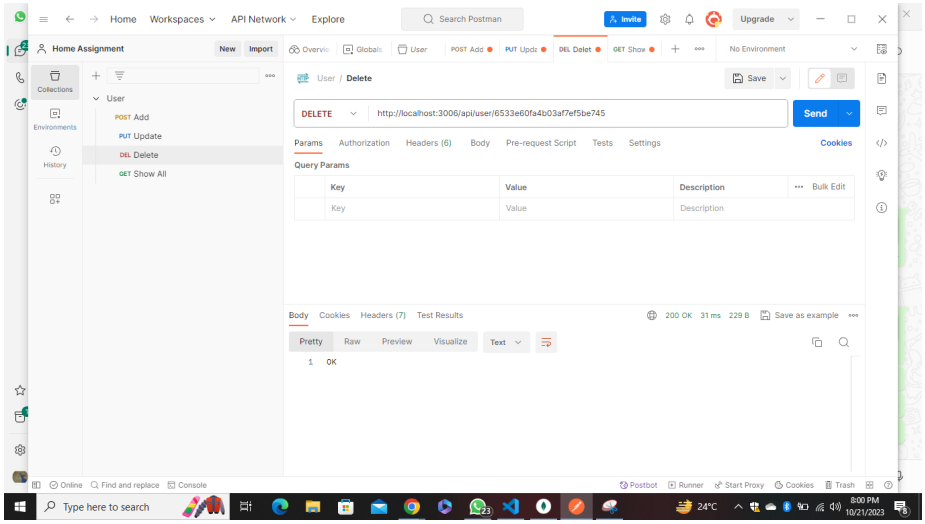

Figure 5: Deleting Record

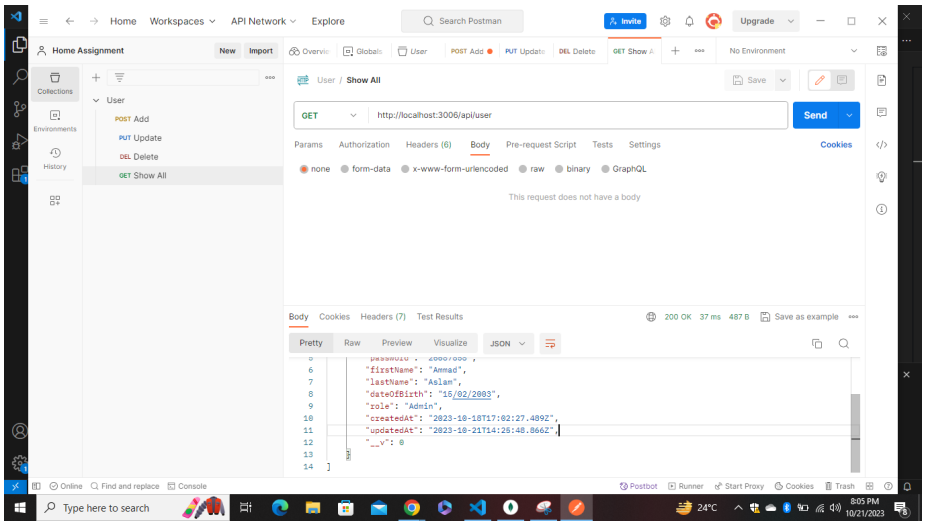

Figure 6: After Delete

2.4 Read API

The Read operation allows you to retrieve data from the database. To implement the Read API, you need to set up a GET endpoint, which can either fetch all records or specific records based on criteria like IDs or filters. This operation is critical for providing data to clients or users. It often involves querying the

database and returning the results as a response.

```
async function getAllusers (req,res) {
    try{
        const cards = await User.find();
        res.json(cards);
    }
    catch (err) {
        res.status(500).json({error : err.message});
    }
}
```

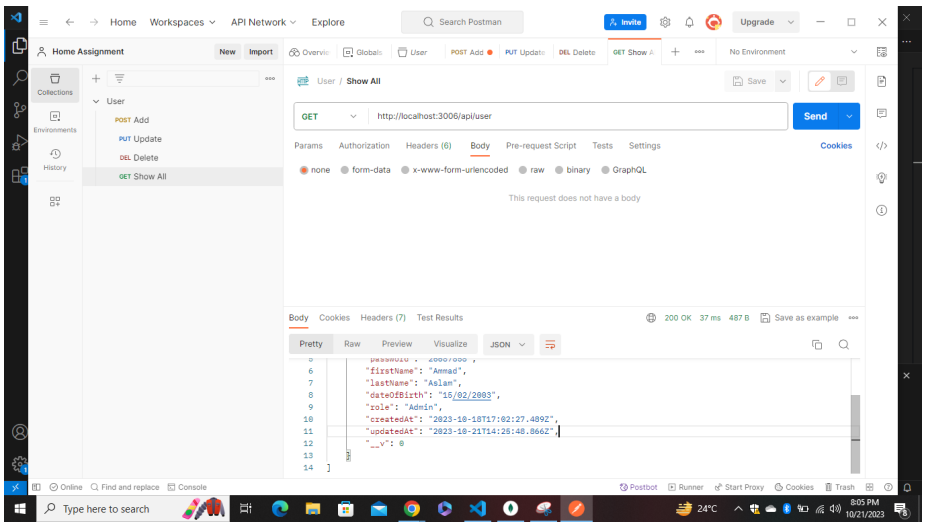

Figure 7: Get All Users

2.5 Login API

The Login operation is a critical part of many applications, especially those requiring user authentication and authorization. It allows users to access their accounts by providing valid credentials, typically a username and password. To implement the Login API, you need to create a POST endpoint that handles user login requests. The client submits the username and password in the request body. The server validates the credentials by checking them against stored user information, usually stored securely in a database.

```
async function login (req,res,next) {
    const {userName , password} = req.body;
    try{
        const user = await User.findOne ({userName});
```

```
        if (!user) return res.status(404).json({error : 'User not found'});
        return res.status(200).json({
            message : 'Logged in successfully' ,
            userName : userName ,
            userid : user.id,
        });
    }
    catch (err) {
        res.status(500).json({error : err.message});
    }
}
```

3 Directory Structure

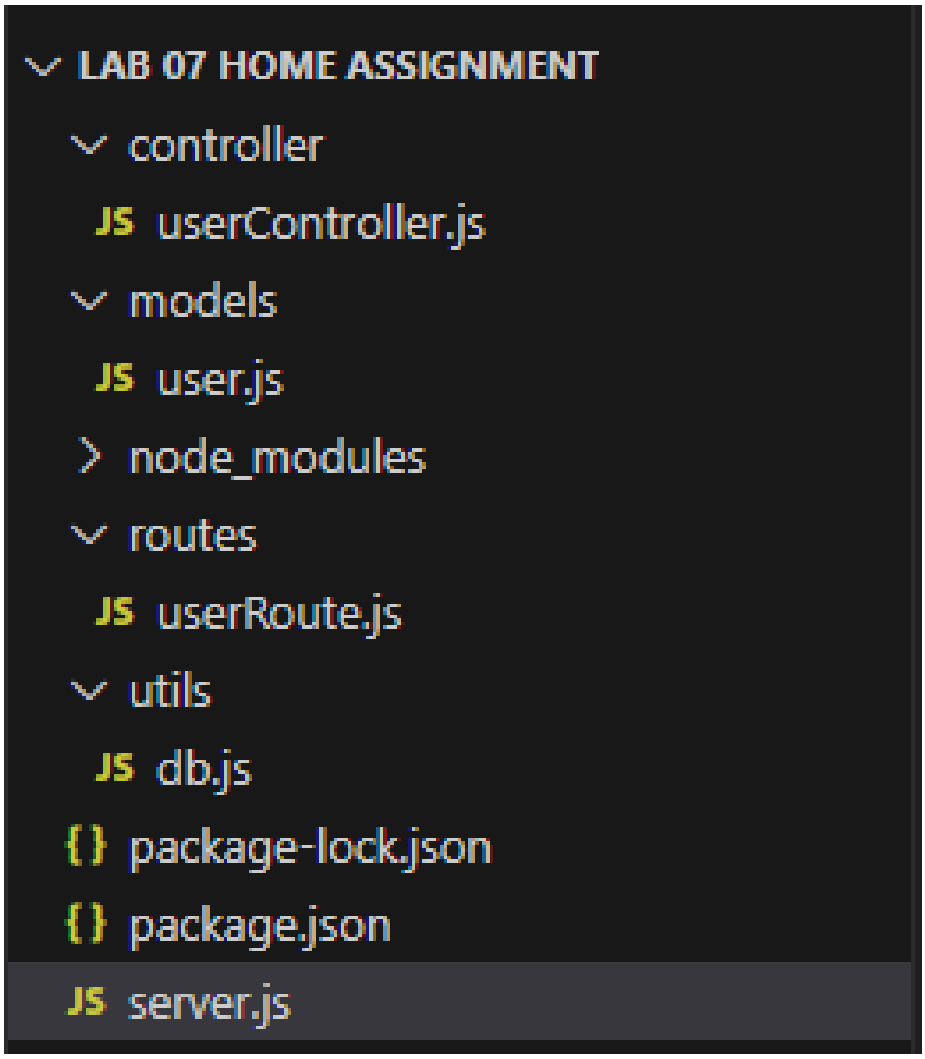

Figure 8: Directories