Name : Pratiksha Angad Gore

Email: pratikshagore08@gmail.com

Contact No: 9834939003

Project Report: Task Manager API

Introduction:

This project involves developing a RESTful API using Node.js and Express.js to manage a collection of tasks. The API supports CRUD (Create, Read, Update, Delete) operations, and tasks are stored in memory. This report outlines the requirements, implementation, and testing of the API .The application uses EJS for rendering views.

Project Requirements:

1. **Framework**: Use Express.js to build the API.
2. **Validation**: Ensure that task properties such as title and description are required.
3. **Data Storage**: Store tasks in memory using an array or object.
4. **API Documentation**: Provide clear and concise API documentation, including sample requests and responses.
5. **Status Codes**: Use appropriate status codes (e.g., 200 for success, 404 for not found, 400 for bad requests).
6. **Testing**: Test the API using tools like Postman or curl, and include instructions for running the API locally.
7. **Error Handling**: Gracefully handle unexpected issues.
8. **Bonus Points**: Implement pagination, sorting, and filtering for the GET /tasks endpoint, and include authentication and authorization mechanisms for protecting certain endpoints.

Setup:

Prerequisites:

- Node.js and npm installed
- Git installed

Installation:

1. **Clone the repository:**

   git clone https://github.com/Pratiksha1302/osumare-backend.git

   cd osumare-backend

2. **Setup the backend**:

   cd backend

   npm install

   npm start

Implementation:

Step 1: Setting Up the Project

Create '*server.js*':

```
const express = require('express');
const app = express();
const port = 8080;
// Middleware to parse JSON
app.use(express.json());
// In-memory storage for tasks
let tasks = [];
```

```
let Item_Id = 1;
// Routes
app.get('/tasks', (req, res) => {
    res.json(tasks);
});
app.get('/tasks/:id', (req, res) => {
    const id = parseInt(req.params.id);
    if (isNaN(id)) {
        return res.status(400).send('Invalid task ID');
    }
    const task = tasks.find(t => t.id === id);
    if (!task) return res.status(404).send('Task not found');
    res.json(task);
});
app.post('/tasks', (req, res) => {
    const { title, description } = req.body;
    if (!title || !description) {
        return res.status(400).send('Title and description are required');
    }
    const newTask = { id: Item_Id++, title, description };
    tasks.push(newTask);
    res.status(201).json(newTask);
});
app.put('/tasks/:id', (req, res) => {
    const task = tasks.find(t => t.id === parseInt(req.params.id));
    if (!task) return res.status(404).send('Task not found');
    const { title, description } = req.body;
    if (!title || !description) {
```

```
        return res.status(400).send('Title and description are required');
    }
    task.title = title;
    task.description = description;
    res.json(task);
});
app.delete('/tasks/:id', (req, res) => {
    const taskIndex = tasks.findIndex(t => t.id === parseInt(req.params.id));
    if (taskIndex === -1) return res.status(404).send('Task not found');
    const deletedTask = tasks.splice(taskIndex, 1);
    res.json(deletedTask);
});
// Error handling middleware
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).send('Something broke!');
});

// Start the server
app.listen(port, () => {
    console.log(`Server is running on http://localhost:${port}`);
});
```

Step 2: Running the Server

Start the server using:

npm start

The server will run at http://localhost:8080.

Step 3: Testing Using Postman

1. **GET /tasks**

- URL: http://localhost:8080/tasks
- Method: GET

2. **GET /tasks/**
   - URL: http://localhost:8080/tasks/1
   - Method: GET
3. **POST /tasks**
   - URL: http://localhost:8080/tasks
   - Method: POST
   - Headers:
     - Content-Type: application/json
   - Body:

```
{
  "title": "New Task",
  "description": "Task description"
}
```

4. **PUT /tasks/**
   - URL: http://localhost:8080/tasks/1
   - Method: PUT
   - Headers:
     - Content-Type: application/json
   - Body:

```
{
  "title": "Updated Task",
  "description": "Updated description"
}
```

5. **DELETE /tasks/**
   - URL: http://localhost:8080/tasks/1
   - Method: DELETE

API Usage with CURL:

- **Create a Task**:

  curl -X POST http://localhost:8080/tasks -H "Content-Type: application/json" -d '{"title": "Task 1", "description": "This is a sample task."}'

- **All Tasks**:

  curl http://localhost:8080/tasks

- **Get a Task by ID**:

  curl http://localhost:8080/tasks/1

- **Update a Task**:

  curl -X PUT http://localhost:8080/tasks/1 -H "Content-Type: application/json" -d '{"title": "Updated Task", "description": "This is an updated task."}'

- **Delete a Task:**

  curl -X DELETE http://localhost:8080/tasks/1

Approach:

1. Environment Setup

● Installed necessary dependencies: express, ejs, method-override, and uuid.

2. Routes and Handlers

● Defined routes for each CRUD operation.

● Implemented handlers to process incoming requests and generate appropriate responses.

3. Views

● Created EJS templates for displaying tasks, creating new tasks, editing tasks, and showing error messages.

4. Validation and Error Handling

● Added validation to ensure that tasks have both a title and description.

● Implemented error handling to provide user-friendly error messages.

5. Testing

● Tested the API using curl commands and a web browser to ensure all endpoints function correctly.

● Verified that appropriate status codes and error messages are returned.

Implementation Details:

1. Task Storage

● Tasks are stored in an in-memory in the form of an array of objects. Each task has a unique ID generated using uuid.

2. Endpoints

● Implemented endpoints for retrieving all tasks, retrieving a specific task, creating a new task, updating a task, and deleting a task.

3. Validation

● Ensured that the title and description fields are present in the request body for creating and updating tasks.

4. Error Handling

● Added error handling middleware to catch unexpected errors and respond with a 500 Internal Server Error.

Challenges and Solutions:

● **Validation**

● **Challenge:** Ensuring that all required fields are present for POST and PUT request.

● **Solution:** Implemented validation logic in the POST and PUT request handlers.

In-Memory Storage:

- **Challenge:** Maintaining unique IDs for tasks.
- **Solution:** Used the uuid package to generate unique IDs for each task.

Conclusion:

The RESTful API was successfully developed and tested. It supports all required CRUD operations and includes validation and error handling. The use of EJS for rendering views enhances the user experience by providing a clear and organized way to interact with the tasks.