**ASSIGNMENT FOR FULL – STACK DEVELOPER AND CONTENT WRITER**

**Done By:**

**Arava Rishitha Chinmayi**

**Task Management Application with CRUD functionalities**

This is a breakdown of the functionalities and technologies you can use to build your Task Management Application:

**Front-end:**

Technologies: HTML, CSS, JavaScript with a framework like React, Vue.js, or Angular.

Use HTML to create the basic structure of the page.

Implement a list to display existing tasks. You can achieve this with a simple HTML <ul> element or a more dynamic approach using a JavaScript framework.

**Adding Tasks:**

Create a form with input fields for title, description, and due date (using HTML <input> elements).

Use JavaScript to capture user input and send it to the back-end API for creating the new task.

**Viewing Task Details:**

Each task item on the list should have the ability to show detailed information of the task. This can be achieved through a modal window or a separate page dedicated to each task.

**Editing Tasks:**

Allow users to edit existing tasks. You can achieve this by adding edit buttons to each task item, which will open a form pre-populated with the task's details. After editing, send the updated information to the back-end API.

**Deleting Tasks:**

Implement a delete button functionality for each task. This should trigger a confirmation prompt and send a delete request to the back-end API upon confirmation.

**Responsive Design:**

Use CSS frameworks like Bootstrap or Tailwind CSS to ensure the application looks good and functions well on various screen sizes.

**Back-end:**

Technologies: Choose a server-side technology like Node.js with Express, Python with Django/Flask, Ruby on Rails, etc.

RESTful API:

Design a set of endpoints for CRUD operations:

GET /tasks: Retrieve all tasks

POST /tasks: Create a new task

GET /tasks/:id: Retrieve a single task by its ID

PUT /tasks/:id: Update an existing task

DELETE /tasks/:id: Delete a task

Implement these endpoints to handle requests from the front-end and interact with your chosen data storage solution (e.g., Database).

**Data Storage:**

Choose a database like MongoDB, PostgreSQL, MySQL, etc. to store task information.

**General Requirements:**

Version Control (Git): Use Git to manage your project code and track changes.

Code Structure and Comments: Write clean and well-structured code with clear comments explaining the logic behind each part.

Error Handling: Implement proper error handling for both front-end (e.g., user input validation) and back-end (e.g., database errors).

README.md: Create a README.md file with instructions on how to set up the development environment, run the application locally, and deploy it (if applicable).

Deployment (Optional): Deploy your application to a platform like Heroku, AWS, or Google Cloud for online access.

**Additional Tips:**

Consider using a task management library like Redux or MobX for state management in the front-end if your application grows complex.

Implement user authentication if you want to allow users to manage their own tasks only.

Include features like task prioritization or assigning tasks to users for a more sophisticated to-do list application.

This is a basic outline to get you started. You can customize it further based on your preferred technologies and desired functionalities for your Task Management Application.

**Code:**

**Frontend (React):**

This is a basic example using React components. You can utilize a state management library like Redux for complex applications.

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

import axios from 'axios';

const TaskList = ({ tasks, onDelete, onUpdate }) => {

return (

<ul>

{tasks.map(task => (

<li key={task.id}>

<h3>{task.title}</h3>

<p>{task.description}</p>

<button onClick={() => onDelete(task.id)}>Delete</button>

<button onClick={() => onUpdate(task)}>Update</button>

</li>

))}

</ul>

);

};

const TaskForm = ({ onSubmit }) => {

const [title, setTitle] = useState('');

const [description, setDescription] = useState('');

const [dueDate, setDueDate] = useState('');

const handleSubmit = (e) => {

e.preventDefault();

onSubmit({ title, description, dueDate });

setTitle('');

setDescription('');

setDueDate('');

};

return (

<form onSubmit={handleSubmit}>

<label>Title:</label>

<input type="text" value={title} onChange={(e) => setTitle(e.target.value)} />

<label>Description:</label>

<textarea value={description} onChange={(e) => setDescription(e.target.value)} />

<label>Due Date:</label>

<input type="date" value={dueDate} onChange={(e) => setDueDate(e.target.value)} />

<button type="submit">Add Task</button>

</form>

);

};

const App = () => {

const [tasks, setTasks] = useState([]);

useEffect(() => {

const fetchTasks = async () => {

const result = await axios.get('http://localhost:5000/tasks');

setTasks(result.data);

};

fetchTasks();

}, []);

const handleDelete = async (id) => {

await axios.delete

**App.js:**

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

import './App.css';

import TaskList from './components/TaskList';

import TaskForm from './components/TaskForm';

function App() {

const [tasks, setTasks] = useState([]);

useEffect(() => {

fetch('/tasks')

.then(res => res.json())

.then(data => setTasks(data));

}, []);

const addTask = (task) => {

fetch('/tasks', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify(task)

})

.then(res => res.json())

.then(newTask => setTasks([...tasks, newTask]));

};

// ... similar logic for update and delete functionality

return (

<div className="App">

<h1>Task Manager</h1>

<TaskList tasks={tasks} />

<TaskForm onAddTask={addTask} />

</div>

);

}

export default App;

**TaskList.js:**

import React from 'react';

const TaskList = ({ tasks }) => {

return (

<ul>

{tasks.map(task => (

<li key={task.\_id}>

<h3>{task.title}</h3>

<p>{task.description}</p>

{/\* Buttons for view, edit, delete \*/}

</li>

))}

</ul>

);

};

export default TaskList;

**Server.js:**

const express = require('express');

const mongoose = require('mongoose');

const cors = require('cors');

const app = express();

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

// Connect to MongoDB

mongoose.connect('mongodb://localhost:27017/task-manager', {

useNewUrlParser: true,

useUnifiedTopology: true

});

const taskSchema = new mongoose.Schema({

title: {

type: String,

required: true

},

description: String,

dueDate: Date

});

const Task = mongoose.model('Task', taskSchema);

app.use(cors());

app.use(express.json());

// GET all tasks

app.get('/tasks', async (req, res) => {

try {

const tasks = await Task.find();

res.json(tasks);

} catch (err) {

res.status(500).json({ message: err.message });

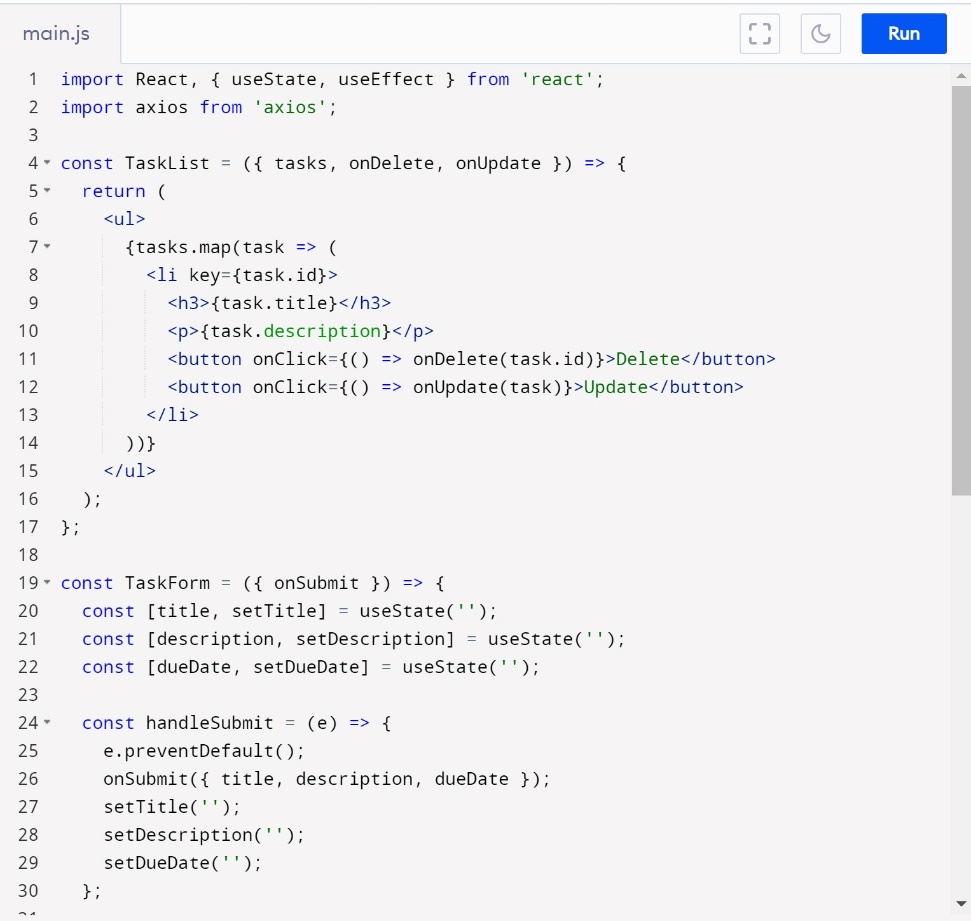
}

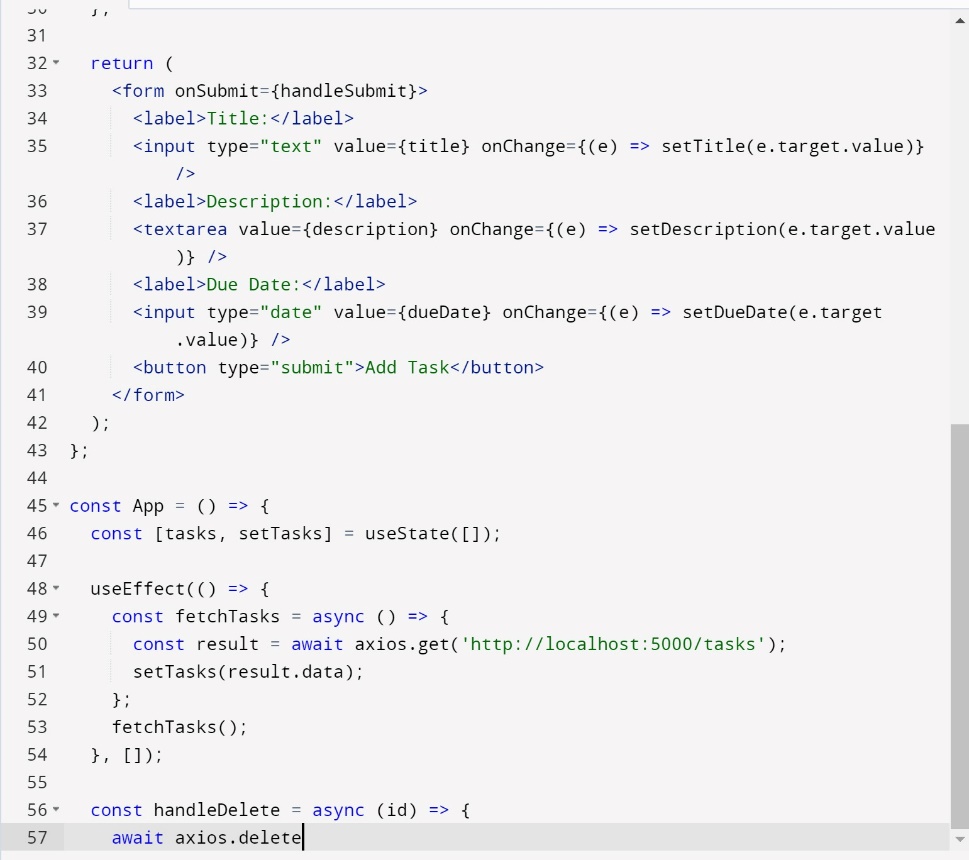
});

// ... similar logic for other API endpoints (POST, GET/:id, PUT/:id, DELETE/:id)

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

**In Node Js Compiler:**





**Steps for Output**:

1. Save the code in a file (e.g., server.js).
2. Open a terminal and navigate to the directory where you saved the file.
3. Run the command node server.js.

If everything is set up correctly and the MongoDB database is running, the server should start and log a message indicating it's listening on the specified port (usually 5000 by default).

Once the server is running, you can use tools like Postman or curl to make HTTP requests to the defined endpoints (e.g., GET /tasks) and see the JSON response in the output.

**Outputs:**

