

Project Title :Task Management Application

Team Members:

Name :Aditya Jyoti Sahu

CAN ID NUMBER :33977669

Name :Ansh Kumar Jha

CAN ID NUMBER :33977303

Name :Gyan Deep

CAN ID NUMBER :33976087

Name :Aditya Kshatriya

CAN ID NUMBER :33977273

Institution Name :Acharya Institute of Technology

Phase 3:Implementation of Project

Objective

The goal of Phase 3 is to implement the core components of the task management application based on the planned architecture. This includes developing the front-end and back-end systems, implementing features for task creation, assignment, and tracking, integrating user authentication, and conducting initial testing to ensure a smooth and efficient user experience.

1.Front-end Development

Overview

The front-end will serve as the user interface for creating, assigning, and managing tasks. It will use React.js to ensure responsiveness and interactivity, with a focus on task organization, progress tracking, and user collaboration features.

Implementation

1.1. Project Structure:

- **App Component:**
 - App.js serves as the root component, importing and rendering the TaskManager component.
 - Contains basic JSX structure and CSS import from App.css.
- **TaskManager Component:**
 - Core UI component for handling task input, search, list rendering, and task management actions.
 - Uses React hooks (useState, useEffect) for managing component state and lifecycle methods.

1.2. UI Components:

- **Input Fields:**
 - Input field for adding a new task.
 - Separate input field for searching tasks.
- **Action Buttons:**
 - Add Task: Adds a new task or updates an existing one.
 - Check/Uncheck: Marks a task as done or not done.
 - Edit: Sets the task to update mode.
 - Delete: Deletes a task.
- **List Rendering:**
 - Dynamically renders tasks using map.
 - Applies conditional CSS classes for completed tasks (line-through styling).

1.3. State Management:

- input: State to manage the task input field.
- tasks: State to store the list of tasks fetched from the backend.
- copyTasks: State to maintain a copy of tasks for search filtering.

- updateTask: State to manage the task currently being updated.

1.4. API Integration:

- API methods are imported from ./api.
 - CreateTask: Sends a POST request to create a new task.
 - GetAllTasks: Sends a GET request to fetch all tasks.
 - DeleteTaskById: Sends a DELETE request to remove a task by ID.
 - UpdateTaskById: Sends a PUT request to update a task.
- API requests are asynchronous, using async/await.
- Error handling: Uses try-catch blocks and displays error messages with notify.

1.5. Search Functionality:

- handleSearch function filters tasks based on the input value.
- Converts both search term and task names to lowercase for case-insensitive searching.
- Updates the tasks state with the filtered results.

1.6. Notifications:

- Uses react-toastify to show success or error messages.
- notify function dynamically displays toasts based on message type (success/error).

1.7. CSS and Styling:

- Uses Bootstrap for layout and responsive design.
- Toastify CSS for notification styling.
- Custom CSS is imported from App.css.

1.8. External Dependencies:

- react-icons: Provides icons for task actions (add, edit, check, delete).
- react-toastify: Provides toast notifications.

- **bootstrap:** Provides responsive UI components and grid system.

Outcome

By the end of this phase, the front-end successfully allows users to create, update, delete, search, and manage tasks with a smooth and interactive experience.

2.Back-End Development

Overview

The back-end of this task management application is built using Node.js and Express.js to provide secure and scalable APIs for task management.

2.1. Project Structure:

- **Task Model:**
 - Defines a Mongoose schema with `taskName` (String) and `isDone` (Boolean) fields.
 - Collection name: `todos`.
- **Database Connection:**
 - Connects to MongoDB using Mongoose.
 - Uses `DB_URL` from environment variables.
- **Controllers:**
 - `createTask`: Creates and saves a new task.
 - `fetchAllTasks`: Retrieves all tasks from the database.
 - `updateTaskById`: Updates a task by ID.
 - `deleteTaskById`: Deletes a task by ID.
- **Routes:**
 - Defines endpoints for creating, fetching, updating, and deleting tasks.
 - Base route: `/tasks`.

2.2. API Endpoints:

- GET /tasks: Fetch all tasks.
- POST /tasks: Create a new task.
- PUT /tasks/:id: Update a task by ID.
- DELETE /tasks/:id: Delete a task by ID.

2.3. Middleware:

- body-parser: Parses incoming JSON requests.
- cors: Enables Cross-Origin Resource Sharing.
- dotenv: Loads environment variables.

2.4. Server Setup:

- Express server listens on PORT (from environment variables or default 8080).
- Base API route: /tasks (handled by TaskRouter).

2.5. Environment and Configuration:

- DB_URL: MongoDB connection string from .env file.
- PORT: Server port configuration.
- Uses Nodemon for hot-reloading in development.

2.6. Deployment:

- Vercel Configuration:
 - vercel.json specifies build and route configurations.
 - Uses @vercel/node for deployment.

Outcome

By the end of this phase, the back-end successfully manages task creation, retrieval, updating, and deletion with secure, scalable, and efficient API endpoints.

3.Database Design:

Overview

The database for this task management application uses **MongoDB** to efficiently store and manage task data.

Schema Design:

- **Task Schema:**
 - `taskName`: String (required) - Represents the name of the task.
 - `isDone`: Boolean (required) - Indicates whether the task is completed.

Data Relationships:

- The database maintains a simple, flat structure with tasks stored in the `todos` collection.
- Each task document contains a unique `_id` (auto-generated by MongoDB).

Outcome

By the end of this phase, the database successfully handles task creation, retrieval, updating, and deletion with data integrity and fast access.

4.Testing and Feedback:

Overview

Thorough testing ensures the application functions correctly, performs well, and delivers a great user experience.

Testing Implementation:

- **Unit Testing:**
 - Test individual components and back-end routes using Jest or Mocha.
- **Integration Testing:**
 - Verify seamless communication between front-end, back-end, and database.
- **End-to-End Testing:**
 - Simulate user flows such as task creation, editing, deletion, and search.

Feedback Loop:

- Gather feedback from test users.
- Identify UX issues and performance bottlenecks.
- Iterate with improvements based on collected feedback.

Outcome

By the end of testing, the task management application runs smoothly, handles edge cases, and provides a refined user experience.

Challenges and Solutions:

Challenge 1: API Error Handling

- **Issue:** Ensuring smooth error handling for failed API calls (e.g., network issues or server downtime).
- **Solution:** Implemented try-catch blocks around async API calls and used toast notifications to inform users of errors.

Challenge 2: Real-Time State Synchronization

- **Issue:** Keeping the UI state (tasks) synchronized with the database after creation, deletion, and updates.
- **Solution:** Triggered `fetchAllTasks` after every task operation to ensure the UI stays updated.

Challenge 3: Case-Insensitive Search

- **Issue:** Search functionality needed to be case-insensitive and performant.
- **Solution:** Converted task names and search input to lowercase before filtering the results.

Challenge 4: Database Connection Reliability

- **Issue:** Ensuring MongoDB stays connected, especially on deployment.
- **Solution:** Used Mongoose connection with retry strategies and logged connection states for better monitoring.

Outcomes of Phase 3:

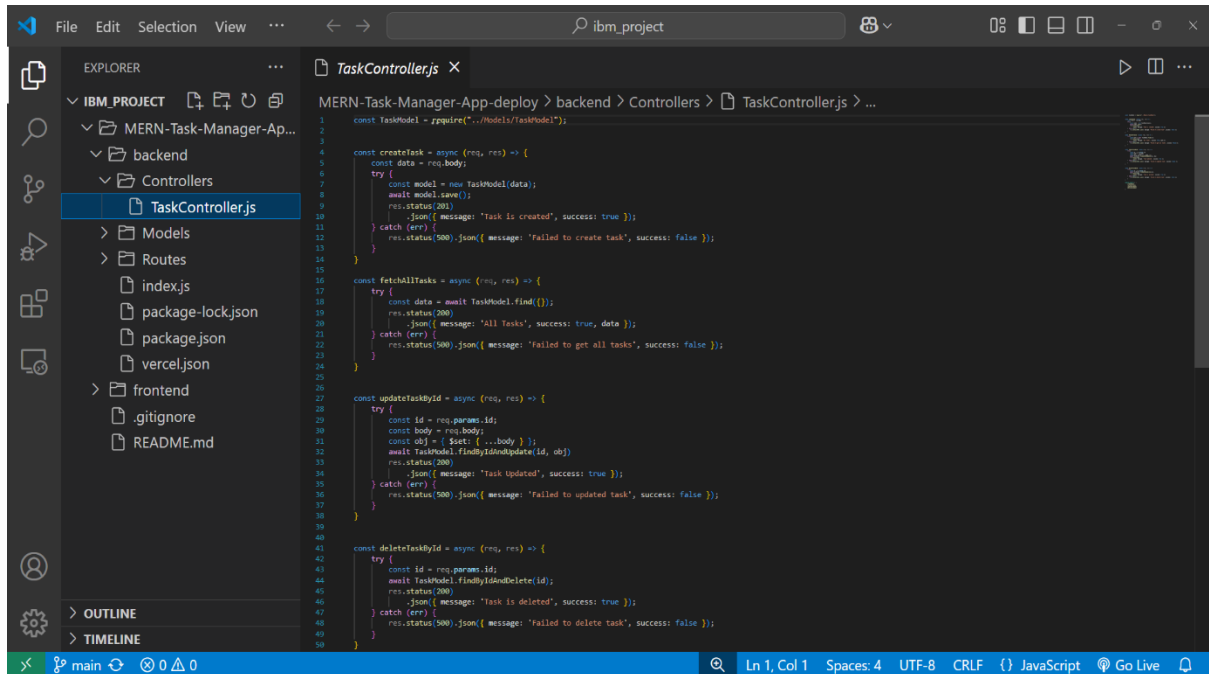
- **Front-End:** A functional UI for task creation, management, and search.
- **Back-End:** Secure and scalable APIs for task operations.
- **Database:** A structured MongoDB database for storing tasks efficiently.
- **Deployment:** Fully deployed application with a smooth user experience.
- **Testing:** Ensures stability, performance, and a seamless workflow.
- **Feedback:** Insights gathered for iterative improvements in the next phase.

Next Steps for Phase 4:

- Optimize performance by implementing lazy loading for large task lists.
- Introduce user authentication to manage individual task lists securely.
- Enhance the UI/UX with drag-and-drop functionality for task reordering.

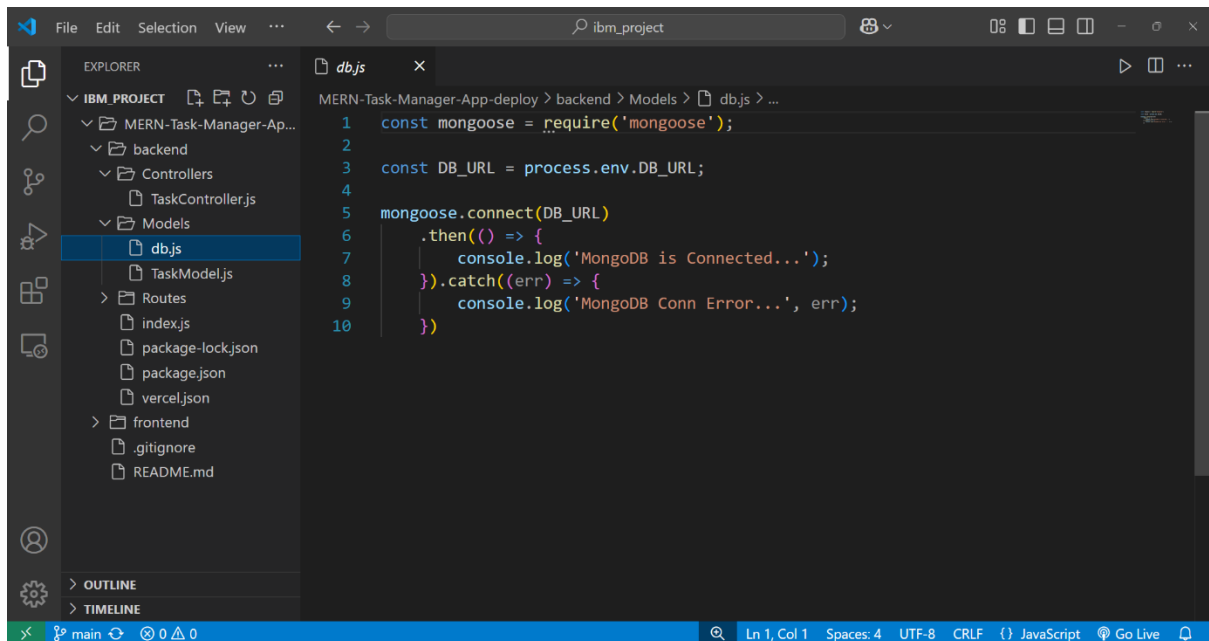
ScreenShots of Code and Progress

Back-End Side



This screenshot shows the Visual Studio Code editor with the file explorer on the left displaying the project structure. The Explorer pane shows the 'MERN-Task-Manager-App-deploy' project with folders for 'backend' and 'Controllers'. The 'TaskController.js' file is selected in the 'Controllers' folder. The main editor area displays the code for 'TaskController.js', which includes functions for creating, fetching, updating, and deleting tasks. The code is written in JavaScript and uses Express.js for handling HTTP requests and responses. The status bar at the bottom indicates the file is at line 1, column 1, with 4 spaces, UTF-8 encoding, and CRLF line endings.

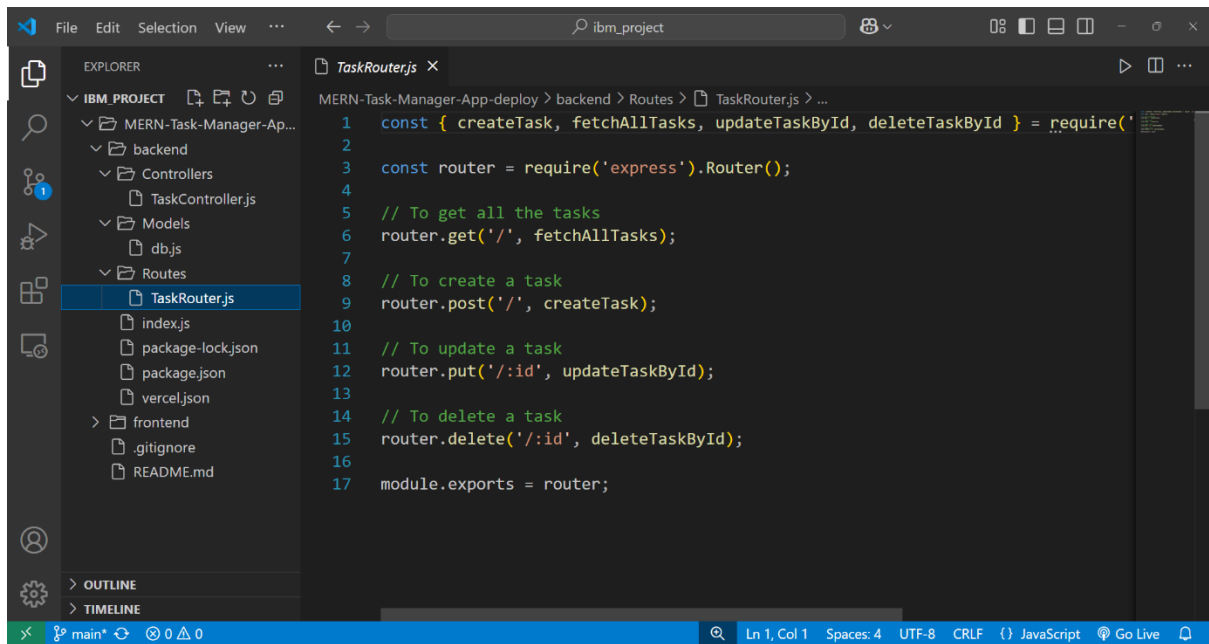
```
1 const TaskModel = require("../Models/TaskModel");
2
3
4 const createTask = async (req, res) => {
5   const data = req.body;
6   try {
7     const model = new TaskModel(data);
8     await model.save();
9     res.status(201)
10    .json({ message: "Task is created", success: true });
11  } catch (err) {
12    res.status(500).json({ message: "failed to create task", success: false });
13  }
14 }
15
16 const fetchallTasks = async (req, res) => {
17   try {
18     const data = await TaskModel.find({});
19     res.status(200)
20     .json({ message: "All Tasks", success: true, data });
21   } catch (err) {
22     res.status(500).json({ message: "failed to get all tasks", success: false });
23   }
24 }
25
26 const updateTaskById = async (req, res) => {
27   try {
28     const id = req.params.id;
29     const body = req.body;
30     const obj = { $set: { ...body } };
31     await TaskModel.findByIdAndUpdate(id, obj);
32     res.status(200)
33     .json({ message: "Task updated", success: true });
34   } catch (err) {
35     res.status(500).json({ message: "failed to update task", success: false });
36   }
37 }
38
39 const deleteTaskById = async (req, res) => {
40   try {
41     const id = req.params.id;
42     await TaskModel.findByIdAndDelete(id);
43     res.status(200)
44     .json({ message: "Task is deleted", success: true });
45   } catch (err) {
46     res.status(500).json({ message: "failed to delete task", success: false });
47   }
48 }
49
50
```



This screenshot shows the Visual Studio Code editor with the file explorer on the left displaying the project structure. The Explorer pane shows the 'MERN-Task-Manager-App-deploy' project with folders for 'backend' and 'Models'. The 'db.js' file is selected in the 'Models' folder. The main editor area displays the code for 'db.js', which sets up the MongoDB connection using Mongoose. The code includes the Mongoose library, the database URL, and the connection function. The status bar at the bottom indicates the file is at line 1, column 1, with 4 spaces, UTF-8 encoding, and CRLF line endings.

```
1 const mongoose = require('mongoose');
2
3 const DB_URL = process.env.DB_URL;
4
5 mongoose.connect(DB_URL)
6   .then(() => {
7     console.log('MongoDB is Connected...');
8   }).catch((err) => {
9     console.log('MongoDB Conn Error...', err);
10  })

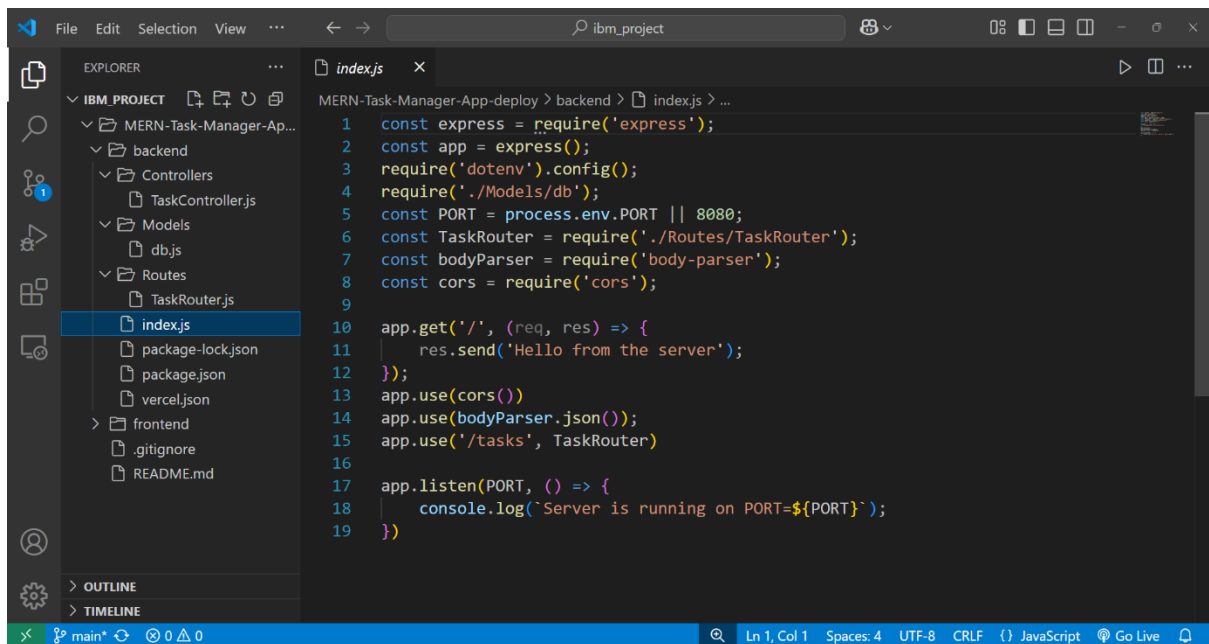
```



This screenshot shows the Visual Studio Code editor with the `TaskRouter.js` file open. The Explorer sidebar on the left shows the project structure: `IBM_PROJECT` > `MERN-Task-Manager-App-deploy` > `backend` > `Routes` > `TaskRouter.js`. The main editor displays the following JavaScript code:

```
1  const { createTask, fetchAllTasks, updateTaskById, deleteTaskById } = require('...')
2
3  const router = require('express').Router();
4
5  // To get all the tasks
6  router.get('/', fetchAllTasks);
7
8  // To create a task
9  router.post('/', createTask);
10
11 // To update a task
12 router.put('/:id', updateTaskById);
13
14 // To delete a task
15 router.delete('/:id', deleteTaskById);
16
17 module.exports = router;
```

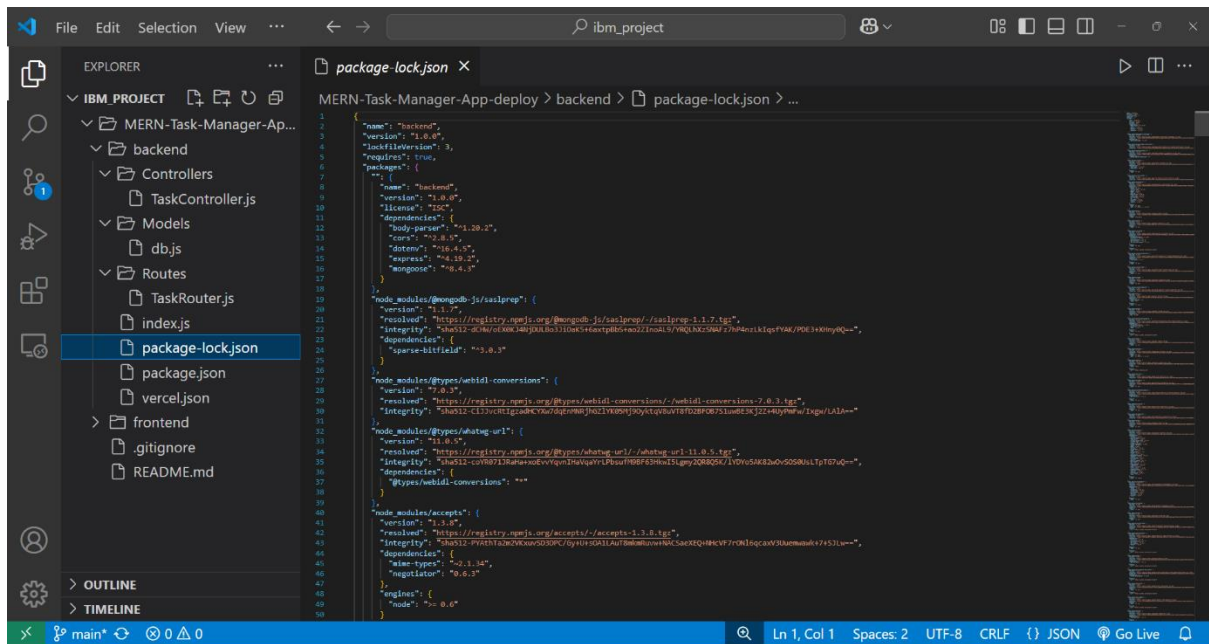
The status bar at the bottom indicates the file is at line 1, column 1, with 4 spaces, UTF-8 encoding, CRLF line endings, and is a JavaScript file.



This screenshot shows the Visual Studio Code editor with the `index.js` file open. The Explorer sidebar on the left shows the project structure: `IBM_PROJECT` > `MERN-Task-Manager-App-deploy` > `backend` > `index.js`. The main editor displays the following JavaScript code:

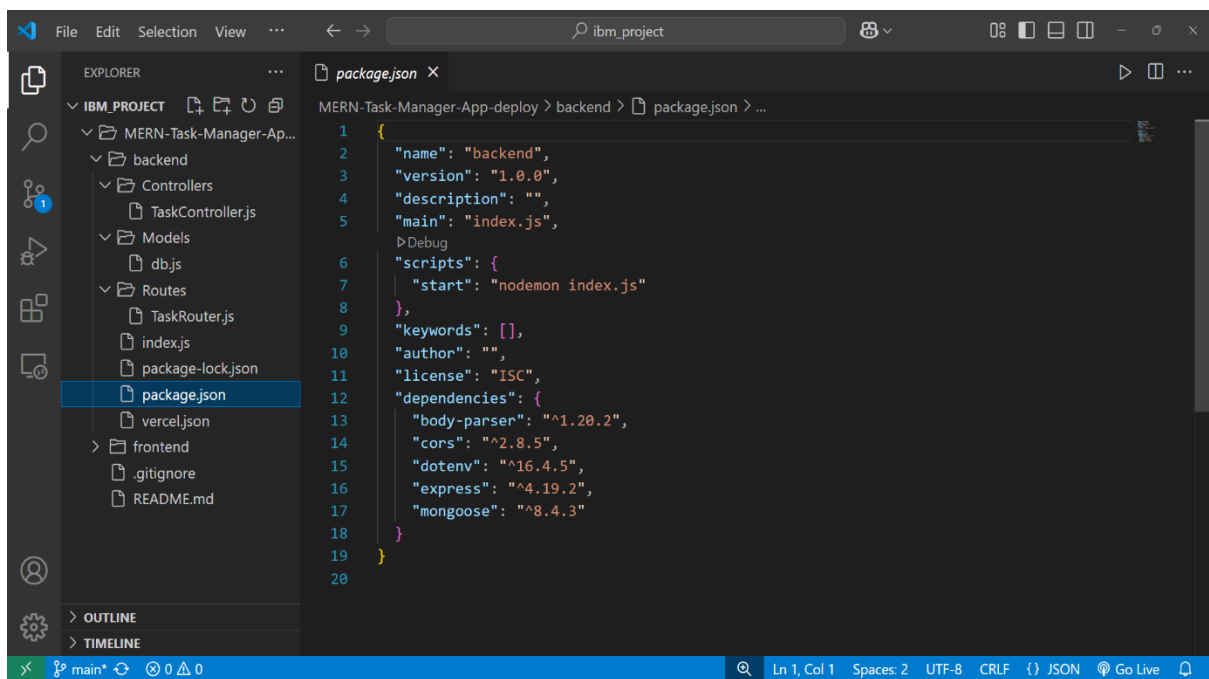
```
1  const express = require('express');
2  const app = express();
3  require('dotenv').config();
4  require('./Models/db');
5  const PORT = process.env.PORT || 8080;
6  const TaskRouter = require('./Routes/TaskRouter');
7  const bodyParser = require('body-parser');
8  const cors = require('cors');
9
10 app.get('/', (req, res) => {
11   res.send('Hello from the server');
12 });
13 app.use(cors())
14 app.use(bodyParser.json());
15 app.use('/tasks', TaskRouter)
16
17 app.listen(PORT, () => {
18   console.log(`Server is running on PORT=${PORT}`);
19 })
```

The status bar at the bottom indicates the file is at line 1, column 1, with 4 spaces, UTF-8 encoding, CRLF line endings, and is a JavaScript file.



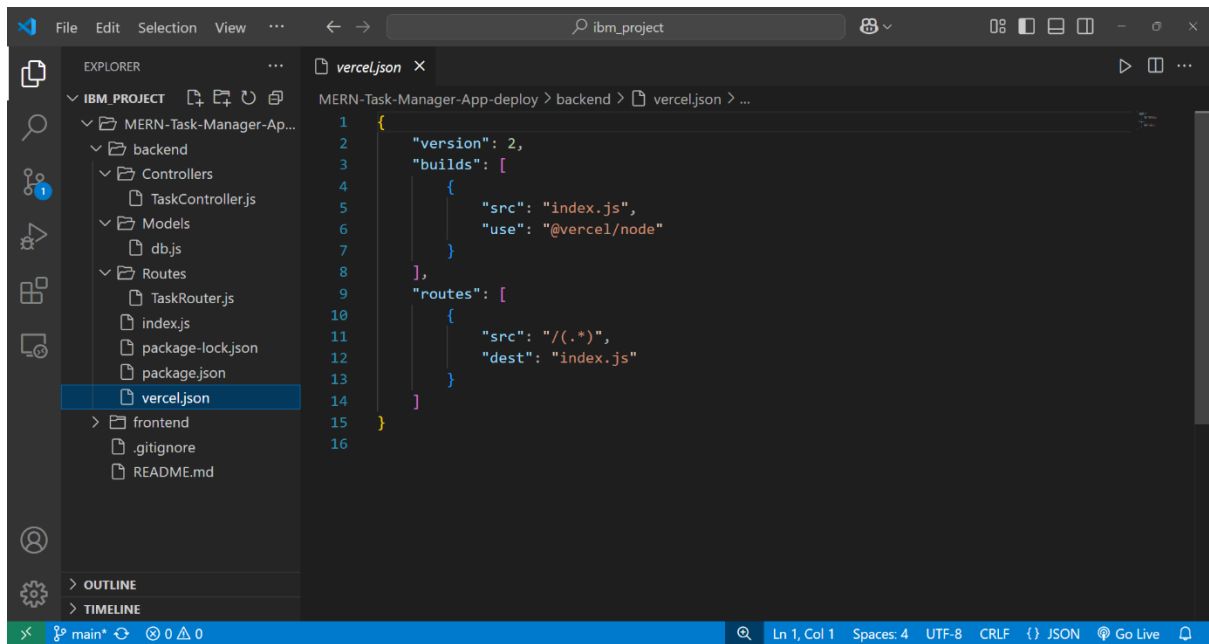
This screenshot shows the Visual Studio Code interface with the `package-lock.json` file open in the editor. The Explorer sidebar on the left shows the project structure for `IBM_PROJECT`, including a `backend` folder with subfolders for `Controllers`, `Models`, and `Routes`, and files like `index.js`, `package-lock.json`, `package.json`, `vercel.json`, `TaskRouter.js`, `db.js`, `TaskController.js`, `frontent`, `.gitignore`, and `README.md`. The Editor pane displays the content of `package-lock.json`, which is a JSON file containing metadata and integrity hashes for the project's dependencies. The status bar at the bottom indicates the file is at line 1, column 1, with 2 spaces, in UTF-8 encoding, with CRLF line endings, and is a JSON file.

```
1 {
2   "name": "backend",
3   "version": "1.0.0",
4   "lockfileVersion": 3,
5   "requires": true,
6   "packages": {
7     "": {
8       "name": "backend",
9       "version": "1.0.0",
10      "license": "ISC",
11      "dependencies": {
12        "body-parser": "^1.20.2",
13        "cors": "^2.8.5",
14        "dotenv": "^16.4.5",
15        "express": "^4.19.2",
16        "mongoose": "^8.4.3"
17      }
18    },
19     "node_modules/@mongodb-js/saslprep": {
20       "version": "1.1.3",
21       "resolved": "https://registry.npmjs.org/@mongodb-js/saslprep/-/saslprep-1.1.3.tgz",
22       "integrity": "sha512-8o8YQ8tYnTnsu6t1Mqc/20OY41YcUgShlDy5RvTbMIx5z2P9TLFbXs5B7cHqMm6WzYYHwR02440yMw/Igw/LALA==",
23       "dependencies": {
24         "sparse-bitfield": "^3.0.3"
25       }
26     },
27     "node_modules/@types/webidl-conversions": {
28       "version": "7.0.0",
29       "resolved": "https://registry.npmjs.org/@types/webidl-conversions/-/webidl-conversions-7.0.0.tgz",
30       "integrity": "sha512-7k2U3/KEWDBxvtNp+/+fsK76qVqrThK+UeB9OZ7Z0jYHqD+kYcXbO0/UnZuZdrOd7JdLlLk+UzH0L4XoPpQA==",
31       "license": "MIT"
32     },
33     "node_modules/@types/whatwg-url": {
34       "version": "11.0.5",
35       "resolved": "https://registry.npmjs.org/@types/whatwg-url/-/whatwg-url-11.0.5.tgz",
36       "integrity": "sha512-coYw0R55WBsBakW03V9KvViVt7h3QqEhSbm9m5LeD49hqgZCd5/X2gWkUa3UH7oWwz9msjFJ9iyyo94JnLWU==",
37       "dependencies": {
38         "@types/webidl-conversions": "*"
39       }
40     },
41     "node_modules/accepts": {
42       "version": "1.3.8",
43       "resolved": "https://registry.npmjs.org/accepts/-/accepts-1.3.8.tgz",
44       "integrity": "sha512-LNVOyhDrloeq71RzD1qRUfYq+y4gUjv2o6k1yr6kbQ9qmLHbnjDzd1VH9bWf8/yE2eI7dlGIAT8/pHwzWW1km==",
45       "dependencies": {
46         "mime-types": "~2.1.34",
47         "negotiator": "0.6.3"
48       }
49     },
50     "node_modules/engines": {
51       "version": "0.0.0"
52     }
53   }
54 }
```



This screenshot shows the Visual Studio Code interface with the `package.json` file open in the editor. The Explorer sidebar on the left shows the project structure for `IBM_PROJECT`, including a `backend` folder with subfolders for `Controllers`, `Models`, and `Routes`, and files like `index.js`, `package-lock.json`, `package.json`, `vercel.json`, `TaskRouter.js`, `db.js`, `TaskController.js`, `frontent`, `.gitignore`, and `README.md`. The Editor pane displays the content of `package.json`, which is a JSON file containing metadata and integrity hashes for the project's dependencies. The status bar at the bottom indicates the file is at line 1, column 1, with 2 spaces, in UTF-8 encoding, with CRLF line endings, and is a JSON file.

```
1 {
2   "name": "backend",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "start": "nodemon index.js"
8   },
9   "keywords": [],
10  "author": "",
11  "license": "ISC",
12  "dependencies": {
13    "body-parser": "^1.20.2",
14    "cors": "^2.8.5",
15    "dotenv": "^16.4.5",
16    "express": "^4.19.2",
17    "mongoose": "^8.4.3"
18  }
19 }
```



The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left. The project structure is as follows:

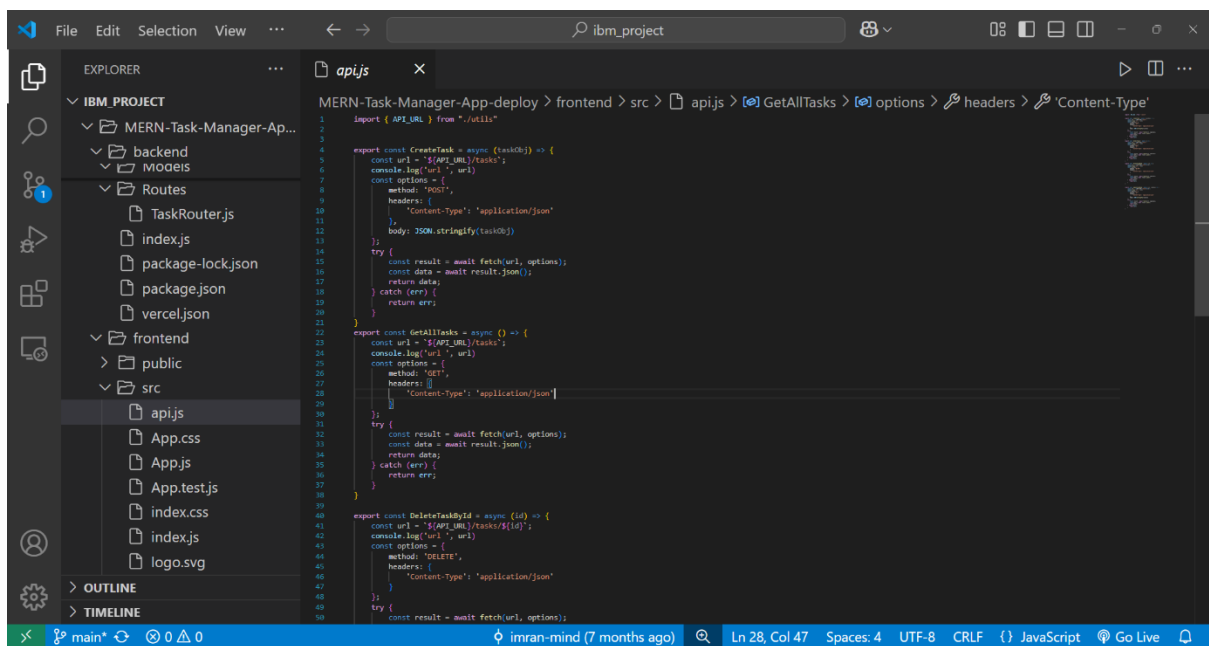
- IBM_PROJECT
 - MERN-Task-Manager-App-deploy
 - backend
 - Controllers
 - TaskController.js
 - Models
 - db.js
 - Routes
 - TaskRouter.js
 - index.js
 - package-lock.json
 - package.json
 - vercel.json (selected)
 - frontend
 - .gitignore
 - README.md

The main editor displays the content of `vercel.json`:

```
1 {
2   "version": 2,
3   "builds": [
4     {
5       "src": "index.js",
6       "use": "@vercel/node"
7     }
8   ],
9   "routes": [
10    {
11      "src": "/(.*)",
12      "dest": "index.js"
13    }
14  ]
15 }
16
```

The status bar at the bottom indicates: Ln 1, Col 1, Spaces: 4, UTF-8, CRLF, JSON, Go Live.

Front-end code:



The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left. The project structure is as follows:

- IBM_PROJECT
 - MERN-Task-Manager-App-deploy
 - backend
 - Controllers
 - TaskRouter.js
 - Models
 - index.js
 - package-lock.json
 - package.json
 - vercel.json
 - frontend
 - public
 - App.css
 - App.js
 - App.test.js
 - index.css
 - index.js
 - logo.svg
 - src
 - api.js (selected)

The main editor displays the content of `api.js`:

```
1 import { API_URL } from './utils'
2
3
4 export const CreateTask = async (taskObj) => {
5   const url = `${API_URL}/tasks`;
6   console.log('url', url);
7   const options = {
8     method: 'POST',
9     headers: {
10       'Content-Type': 'application/json'
11     },
12     body: JSON.stringify(taskObj)
13   };
14   try {
15     const result = await fetch(url, options);
16     const data = await result.json();
17     return data;
18   } catch (err) {
19     return err;
20   }
21 }
22
23 export const GetAllTasks = async () => {
24   const url = `${API_URL}/tasks`;
25   console.log('url', url);
26   const options = {
27     method: 'GET',
28     headers: {
29       'Content-Type': 'application/json'
30     }
31   };
32   try {
33     const result = await fetch(url, options);
34     const data = await result.json();
35     return data;
36   } catch (err) {
37     return err;
38   }
39 }
40
41 export const DeleteTaskById = async (id) => {
42   const url = `${API_URL}/tasks/${id}`;
43   console.log('url', url);
44   const options = {
45     method: 'DELETE',
46     headers: {
47       'Content-Type': 'application/json'
48     }
49   };
50   try {
51     const result = await fetch(url, options);
52   }
53 }
```

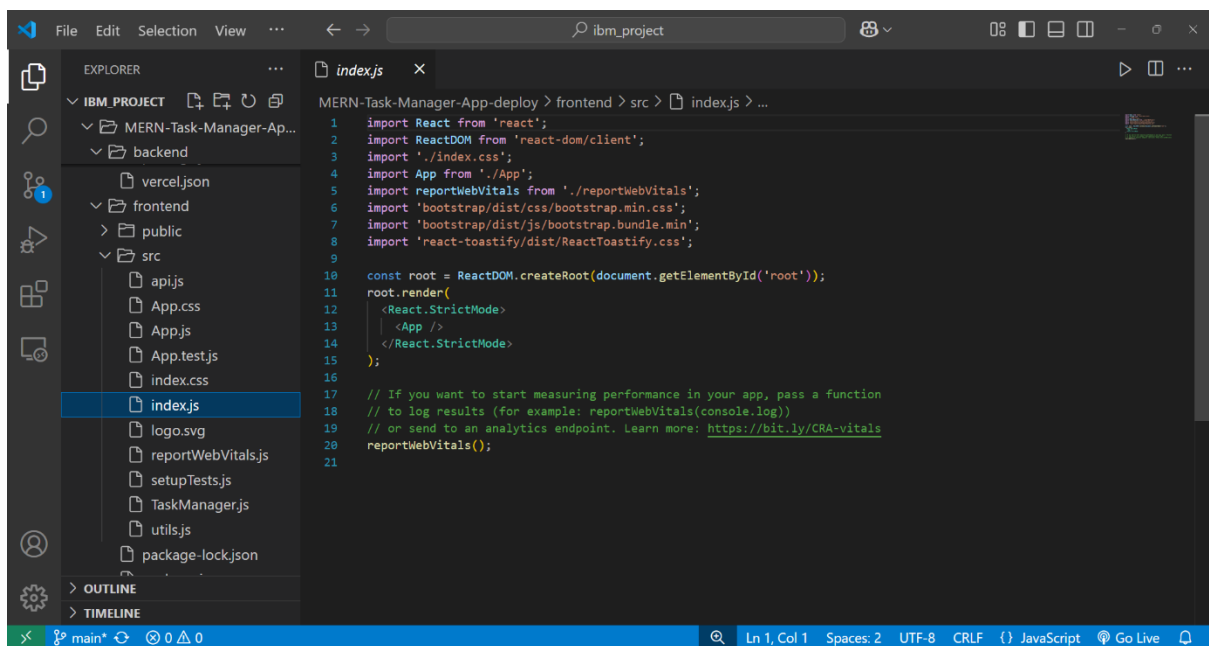
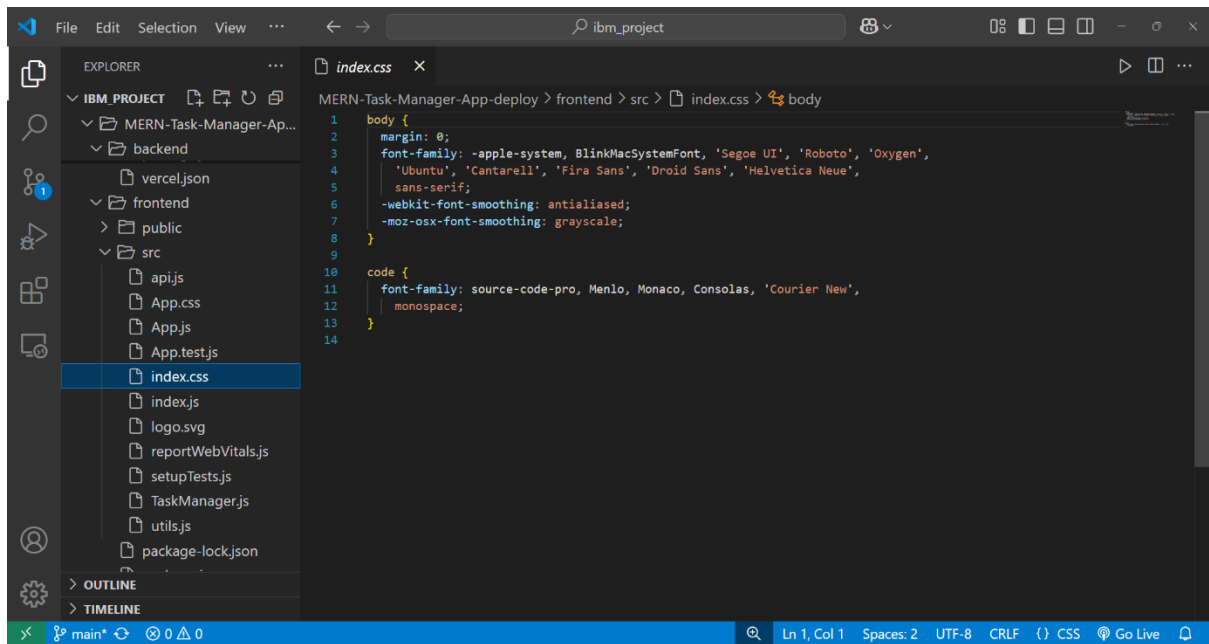
The status bar at the bottom indicates: Ln 28, Col 47, Spaces: 4, UTF-8, CRLF, JavaScript, Go Live.

This screenshot shows the Visual Studio Code editor with a project named 'ibm_project'. The Explorer sidebar on the left displays the file structure, with the 'src' directory expanded to show 'App.js' selected. The main editor window displays the content of 'App.js', which is a React component. The code includes imports for './App.css' and 'TaskManager' from './TaskManager'. The 'App' function returns a JSX element consisting of a 'div' with 'className="App"' containing a 'TaskManager' component. The file is exported as the default export.

```
1 import './App.css';
2 import TaskManager from './TaskManager';
3
4 function App() {
5   return (
6     <div className="App">
7       <TaskManager />
8     </div>
9   );
10 }
11
12 export default App;
13
```

This screenshot shows the Visual Studio Code editor with the same 'ibm_project'. The Explorer sidebar shows the 'src' directory expanded, with 'App.test.js' selected. The main editor window displays the content of 'App.test.js', which is a Jest test file. It imports 'render' and 'screen' from '@testing-library/react' and the 'App' component from './App'. The test function 'renders learn react link' uses 'render' to render the component, 'screen.getByText' to find a link element, and 'expect' to verify it is in the document.

```
1 import { render, screen } from '@testing-library/react';
2 import App from './App';
3
4 test('renders learn react link', () => {
5   render(<App />);
6   const linkElement = screen.getByText(/learn react/i);
7   expect(linkElement).toBeInTheDocument();
8 });
9
```



The screenshot shows the Visual Studio Code editor with the file explorer on the left and the editor window on the right. The file explorer shows the project structure: IBM_PROJECT > MERN-Task-Manager-Ap... > backend > src. The file reportWebVitals.js is selected in the src folder. The editor window shows the code for reportWebVitals.js, which is a JavaScript file that exports a default function reportWebVitals. The code is as follows:

```
1 const reportWebVitals = onPerfEntry => {
2   if (onPerfEntry && onPerfEntry instanceof Function) {
3     import('web-vitals').then(({ getCLS, getFID, getFCP, getLCP, getTTFB }) => {
4       getCLS(onPerfEntry);
5       getFID(onPerfEntry);
6       getFCP(onPerfEntry);
7       getLCP(onPerfEntry);
8       getTTFB(onPerfEntry);
9     });
10  }
11 };
12
13 export default reportWebVitals;
```

The status bar at the bottom shows the file is on line 1, column 1, with 2 spaces, UTF-8 encoding, CRLF line endings, and is a JavaScript file. The Go Live button is also visible.

The screenshot shows the Visual Studio Code editor with the file explorer on the left and the editor window on the right. The file explorer shows the project structure: IBM_PROJECT > MERN-Task-Manager-Ap... > backend > src. The file setupTests.js is selected in the src folder. The editor window shows the code for setupTests.js, which is a JavaScript file that imports jest-dom and sets up the testing environment. The code is as follows:

```
1 // jest-dom adds custom jest matchers for asserting on DOM nodes.
2 // allows you to do things like:
3 // expect(element).toHaveTextContent(/react/i)
4 // learn more: https://github.com/testing-library/jest-dom
5 import '@testing-library/jest-dom';
6
```

The status bar at the bottom shows the file is on line 1, column 1, with 4 spaces, UTF-8 encoding, CRLF line endings, and is a JavaScript file. The Go Live button is also visible.

The screenshot shows the Visual Studio Code editor with the file explorer on the left and the code editor on the right. The file explorer shows the project structure: IBM_PROJECT > MERN-Task-Manager-App... > frontend > src. The file TaskManager.js is selected. The code editor shows the following JavaScript code:

```
1 import React, { useEffect, useState } from 'react';
2 import { FaCheck, FaPencilAlt, FaPlus, FaSearch, FaTrash } from 'react-icons/fa';
3 import { ToastContainer } from 'react-toastify';
4 import { CreateTask, DeleteTaskById, GetAllTasks, UpdateTaskById } from './api';
5 import { notify } from './utils';
6
7 function TaskManager() {
8   const [input, setInput] = useState('');
9   const [tasks, setTasks] = useState([]);
10   const [copyTasks, setCopyTasks] = useState([]);
11   const [updateTask, setUpdateTask] = useState(null);
12
13   const handleTask = () => {
14     if (updateTask && input) {
15       //update api call
16       console.log('update api call');
17       const obj = {
18         taskName: input,
19         isDone: updateTask.isDone,
20         _id: updateTask._id
21       };
22       handleUpdateTask(obj);
23     } else if (updateTask === null && input) {
24       console.log('create api call');
25       //create api call
26       handleAddTask();
27     }
28     setInput('');
29   };
30
31   useEffect(() => {
32     if (updateTask) {
33       setInput(updateTask.taskName);
34     }
35     setUpdateTask();
36   }, [updateTask]);
37
38   const handleAddTask = async () => {
39     const obj = {
40       taskName: input,
41       isDone: false
42     };
43     try {
44       const { success, message } = await CreateTask(obj);
45       if (success) {
46         //show message toast
47         notify(message, 'success');
48       } else {
49         //show error toast
50         notify(message, 'error');
51       }
52     }
53   };
54 }
```

The screenshot shows the Visual Studio Code editor with the file explorer on the left and the code editor on the right. The file explorer shows the project structure: IBM_PROJECT > MERN-Task-Manager-App... > frontend > src. The file utils.js is selected. The code editor shows the following JavaScript code:

```
1 import { toast } from 'react-toastify';
2
3 export const notify = (message, type) => {
4   toast[type](message);
5 }
6
7 export const API_URL = 'https://mern-task-manager-app-deploy-api.vercel.app';
8
```


Progress

