

**COLLEGE CODE:9629**

**COLLEGE NAME:VINS CHRISTIAN COLLEGE OF ENGINEERING, NAGERCOIL.**

**DEPARTMENT : COMPUTER SCIENCE AND**

**ENGINEERING** STUDENT NM-ID:

CBEE46F5D0D5008B59B42D330AFCF1B1

ROLLNO:96923104006

**DATE:22/09/2025**

**Completed the project named as Phase:5 – FE**

**NAME: To-Do list application**

SUBMITTED BY : Albert Instin.V G

**MOBILE**

**NO:8122664706**

# ****To-Do List Application****

## 🧭 **Introduction**

The **To-Do List Application** is a productivity tool designed to help users manage their daily tasks efficiently. It allows users to create, update, delete, and organize tasks based on priorities, categories, or deadlines. The primary objective is to provide a clean, user-friendly interface that enhances task management and boosts personal productivity.

## 🛠️ **Tech Stack**

| **Layer** | **Technology Used** |
| --- | --- |
| **Frontend** | React.js, HTML5, CSS3, JavaScript, Tailwind CSS or Bootstrap |
| **Backend** | Node.js with Express.js |
| **Database** | MongoDB (with Mongoose ORM) |
| **Authentication** | JSON Web Token (JWT) (optional) |
| **Other Tools** | Git, GitHub, Postman, Vercel/Netlify (frontend), Render/Heroku (backend) |

## 🧱 **System Design**

### 🏗️ ****Architecture Diagram****

Client (React App)

↓

REST API (Express.js)

↓

Database (MongoDB)

* The **React frontend** communicates with the **Express backend** via RESTful APIs.
* The **backend** handles routing, business logic, and interacts with the **MongoDB database** for CRUD operations.

### 🔁 ****Data Flow Diagram****

1. User interacts with UI (e.g., adds a task).
2. Frontend sends a POST request to the backend API.
3. Backend validates and stores the data in MongoDB.
4. Updated task list is fetched and displayed in the UI.

### 🧩 ****Component Breakdown (Frontend)****

| **Component** | **Responsibility** |
| --- | --- |
| App.js | Root component, routing, layout |
| TaskList.js | Displays list of tasks |
| TaskItem.js | Individual task component (edit/delete options) |
| AddTask.js | Form to add new tasks |
| EditTask.js | Modal/form to edit tasks |
| FilterBar.js | Filters for task status, category, etc. |
| Navbar.js | Navigation/header |

## 🌟 **Key Features**

* ✅ Add, edit, and delete tasks
* ✅ Mark tasks as completed/incomplete
* ✅ Filter and search tasks
* ✅ Responsive design for mobile/desktop
* ✅ Persistent data storage (MongoDB)
* ✅ Optional user authentication (login/register)
* ✅ Dark mode (optional)

## 🔮 **Future Improvements**

* 🔐 User login/authentication with personalized task lists
* 📆 Task scheduling and reminders
* 📊 Analytics dashboard (e.g., completed tasks, weekly productivity)
* 🔁 Drag-and-drop task reordering
* ☁️ Sync with calendar (Google Calendar integration)
* 🔄 Offline mode with local storage fallback

## 🎓 **Learnings & Takeaways**

* Gained hands-on experience with the **MERN stack** (MongoDB, Express, React, Node).
* Learned how to build and consume **RESTful APIs**.
* Implemented **state management** using React hooks.
* Improved understanding of **CRUD operations**, **form validation**, and **component design**.
* Deployed a full-stack app using **Render**, **Netlify**, and **environment variables**.
* Learned how to debug and handle **CORS**, **API errors**, and **deployment issues**.

Here's how you can structure the **📷 Screenshots / API Documentation** section of your **To-Do List Application Project Report**:

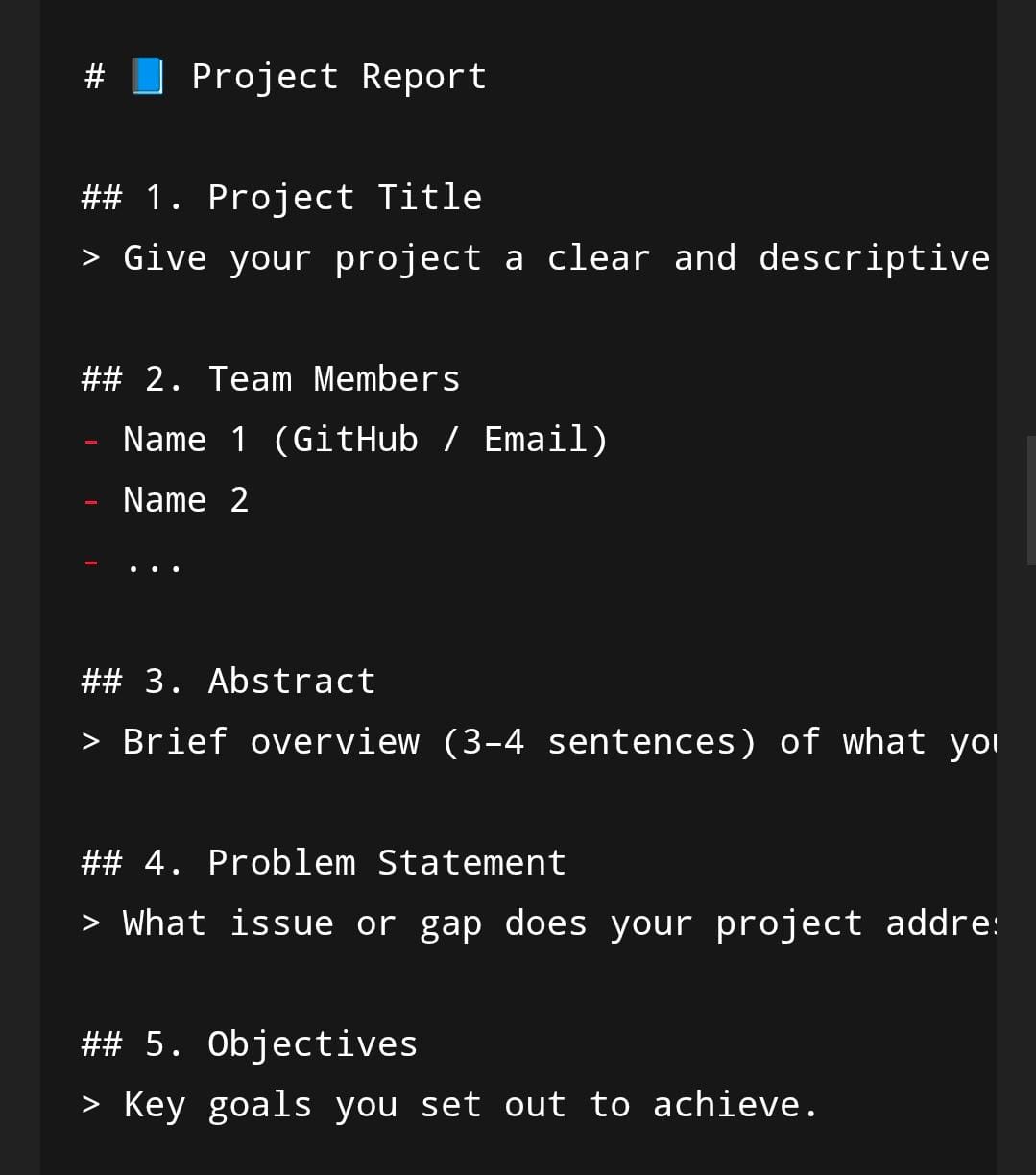
# 📷 ****Screenshots / API Documentation****

## 🖼️ **Screenshots**

📌 Note: Replace the placeholder text (*[Insert Screenshot Here]*) with actual images/screenshots.

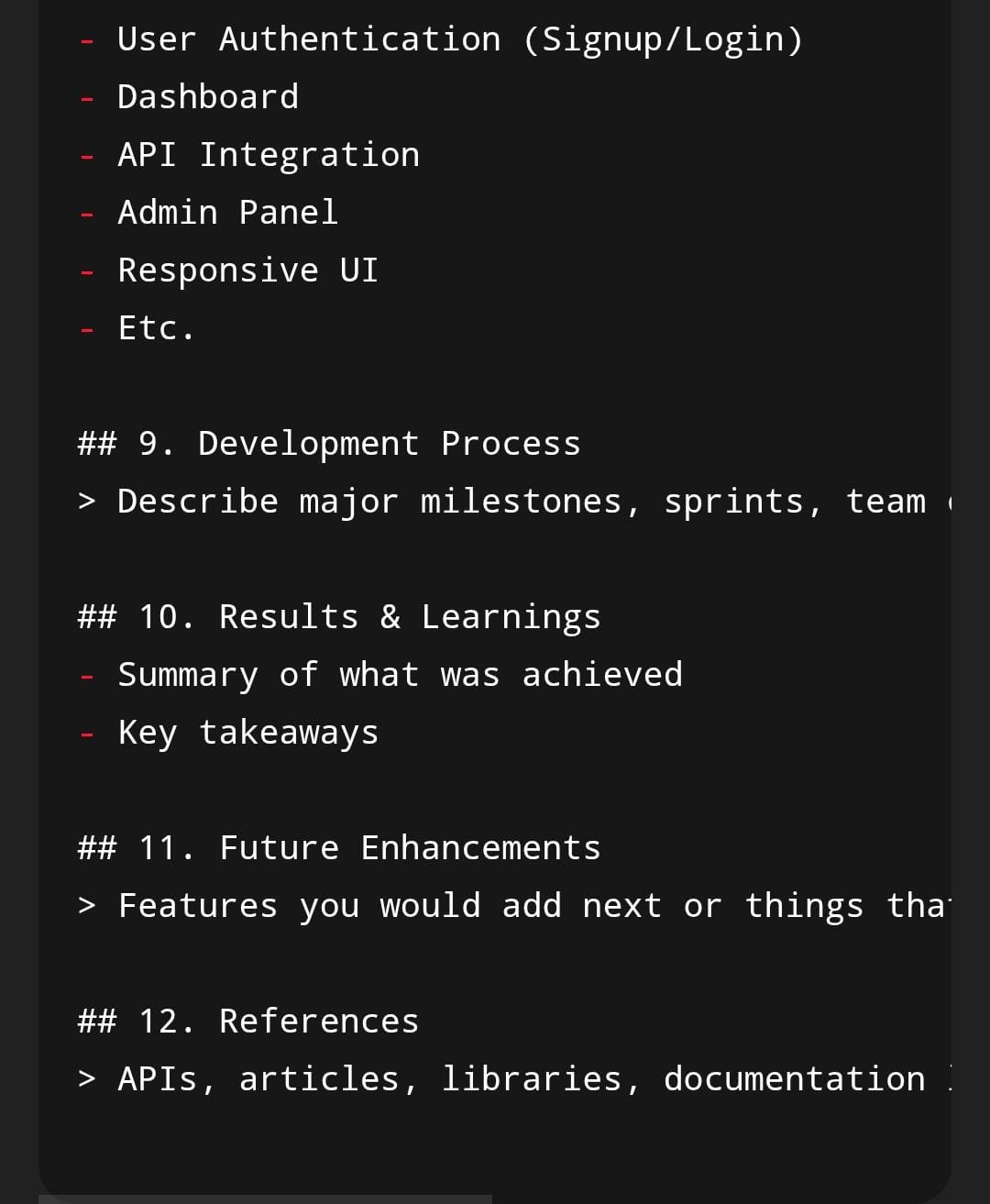
### 1. 🏠 ****Home Page****

* Description: The landing page or dashboard displaying the list of all tasks, navigation bar, and task controls.
* Screenshot:



### 2. ➕ ****Task Creation / Edit Interface****

* Description: Interface where users can add new tasks or edit existing ones.
* Screenshot:



### 3. 📋 ****Task List with Filters****

* Description: Shows tasks with options to filter by status (completed/pending), category, or date.
* Screenshot:

### 4. 📱 ****Mobile/Responsive View****

* Description: Responsive version of the app on smaller screen sizes.
* Screenshot:

## 📡 **API Documentation**

📌 Base URL (for example): https://your-todo-app-api.com/api

### 🔧 ****Authentication Endpoints**** (Optional)

#### POST /auth/register

* **Description:** Register a new user
* **Request:**

{

"username": "johndoe",

"email": "john@example.com",

"password": "password123"

}

* **Response:**

{

"message": "User registered successfully",

"token": "jwt\_token\_here"

}

#### POST /auth/login

* **Description:** Authenticate user
* **Request:**

{

"email": "john@example.com",

"password": "password123"

}

* **Response:**

{

"message": "Login successful",

"token": "jwt\_token\_here"

}

### 📋 ****Task CRUD Endpoints****

#### GET /tasks

* **Description:** Fetch all tasks for the user
* **Headers:**  
  Authorization: Bearer <jwt\_token> (if authentication is implemented)
* **Response:**

[

{

"\_id": "1",

"title": "Buy groceries",

"description": "Milk, Bread, Eggs",

"completed": false,

"dueDate": "2025-10-18"

}

]

#### POST /tasks

* **Description:** Create a new task
* **Request:**

{

"title": "Finish assignment",

"description": "Due by tomorrow",

"dueDate": "2025-10-16"

}

* **Response:**

{

"message": "Task created successfully",

"task": {

"\_id": "2",

"title": "Finish assignment",

"description": "Due by tomorrow",

"completed": false

}

}

#### PUT /tasks/:id

* **Description:** Update an existing task
* **Request:**

{

"title": "Finish assignment (edited)",

"completed": true

}

* **Response:**

{

"message": "Task updated successfully",

"task": { ...updated\_task\_object }

}

#### DELETE /tasks/:id

* **Description:** Delete a task
* **Response:**

{

"message": "Task deleted successfully"

}

### 🛠️ ****Tools for API Testing****

* **Postman Collection**: (Attach or link your collection JSON file if available)
* **Swagger/OpenAPI**: (Optional — If you’ve documented APIs using Swagger UI, provide a link or screenshot)

# 🧩 ****Challenges & Solutions****

### 🔄 ****1. Handling State Management in the Frontend****

**Problem:**  
Managing the state of tasks (add, edit, delete, complete) across multiple components became complex as the application grew. Updating the UI instantly and maintaining data consistency was challenging.

**Solution:**  
Used **React's useState and useEffect hooks** for basic state management. Lifted the state up to parent components where necessary to ensure consistent data flow. Used **context API** to avoid prop drilling in deeply nested components.

**Tools/Techniques Used:**

* React Hooks (useState, useEffect)
* Context API (for global task state)
* Component-driven architecture

### 🛡️ ****2. Backend Data Validation****

**Problem:**  
Tasks with missing or invalid fields (e.g., empty titles or invalid dates) caused errors or corrupted database entries.

**Solution:**  
Implemented server-side validation using **Mongoose schema validators** and custom middleware in Express. Ensured that every request is validated before interacting with the database.

**Tools/Techniques Used:**

* Mongoose schema validation
* Express.js error handling middleware
* Postman (for testing validation)

### 🔁 ****3. Syncing Real-Time Updates (Optional Feature)****

**Problem:**  
Ensuring real-time sync across tabs or users was difficult, especially when tasks were updated or deleted.

**Solution:**  
Due to time constraints, real-time sync via **WebSockets (Socket.io)** was considered but not implemented. As a temporary solution, **manual refresh** or **interval-based polling** was used to keep task lists updated.

**Tools/Techniques Used:**

* Planned use of WebSockets
* Fallback: polling using setInterval() or on-demand refresh
* Redux (if used) could also help in larger apps

### 🚀 ****4. Deployment Issues****

**Problem:**  
Faced issues deploying the full-stack app where the frontend couldn’t connect to the backend due to **CORS errors**, incorrect API URLs, or missing environment variables.

**Solution:**

* Fixed CORS issues by configuring **CORS middleware** on the Express server.
* Used **relative API paths** or configured **proxy** in frontend during development.
* Added necessary **environment variables** for production environments.
* Ensured deployment platforms (Netlify, Render) were properly configured to support Node.js and static hosting.

**Tools/Techniques Used:**

* cors npm package
* .env files
* Render (backend), Netlify or Vercel (frontend)
* Cross-env & build scripts for production

### 🔐 ****5. Authentication & Authorization (Optional)****

**Problem:**  
Restricting access to task data based on users was tricky without a proper authentication flow.

**Solution:**  
Implemented **JWT-based authentication**, storing tokens in localStorage. Added middleware to protect routes and verify tokens before accessing protected resources.

**Tools/Techniques Used:**

* JSON Web Tokens (JWT)
* Express middleware for auth checks
* LocalStorage for token persistence
* Postman for testing protected routes

Absolutely — here are **more real-world challenges** that often arise during development of a full-stack **To-Do List Application**, along with detailed **solutions** and **tools/techniques** used.

### 📱 ****6. Mobile Responsiveness and UI Consistency****

**Problem:**  
The application looked fine on desktop but broke on smaller screen sizes. Buttons overlapped, text overflowed, and layout elements were misaligned.

**Solution:**  
Adopted a **mobile-first approach** using **Tailwind CSS** (or media queries in plain CSS). Used responsive grid and flexbox layouts to ensure adaptability. Also tested across multiple screen sizes using browser dev tools.

**Tools/Techniques Used:**

* Tailwind CSS / Bootstrap classes (sm:, md:, lg:)
* CSS Flexbox and Grid
* Chrome DevTools for responsive testing

### 🧪 ****7. Testing API and UI Interactions****

**Problem:**  
Manually testing each API endpoint and UI interaction became time-consuming and error-prone during development iterations.

**Solution:**  
Used **Postman** to test backend APIs and automate repeated test cases. Added **console logs** and React DevTools for frontend debugging. Considered adding unit tests using Jest for critical components.

**Tools/Techniques Used:**

* Postman (API testing)
* React Developer Tools (Chrome extension)
* Console logging & breakpoints
* (Optional) Jest + React Testing Library

### 🔄 ****8. Syncing Frontend with Backend Changes****

**Problem:**  
Changes made to the backend models or API routes sometimes caused the frontend to break due to mismatched expectations.

**Solution:**  
Maintained clear documentation of all endpoints and response structures. Used shared constants and ensured regular communication between frontend and backend code (even if solo dev). Created a mock JSON server at one point to test UI independently.

**Tools/Techniques Used:**

* Swagger / OpenAPI (for documenting API)
* REST Client / Postman mock server
* Version control (Git) to track model changes
* Consistent naming conventions across frontend and backend

### 🧾 ****9. Managing Forms and Input Validation****

**Problem:**  
Form inputs for creating/editing tasks did not validate properly. Users could submit empty or invalid data, causing unexpected behavior.

**Solution:**  
Used **HTML5 validation attributes** and added extra checks in React before submitting data to backend. Also implemented real-time input validation and user-friendly error messages.

**Tools/Techniques Used:**

* required, minLength, type="date" in HTML
* React form state management
* Toast notifications for feedback (e.g., react-toastify)

### 🧠 ****10. Maintaining Code Quality and Readability****

**Problem:**  
As the app grew, the code became messy — with large components, repeated logic, and unclear naming conventions.

**Solution:**  
Refactored code into **modular components**, followed **naming conventions**, and separated logic from presentation. Used **Prettier** and **ESLint** for formatting and linting.

**Tools/Techniques Used:**

* Prettier + ESLint (code formatting & quality)
* Component-based structure in React
* Comments and consistent naming
* Folder structure: /components, /pages, /api, /utils

### 🔄 ****11. Handling Asynchronous Data Fetching****

**Problem:**  
While fetching data from the backend (e.g., on page load), the UI showed empty states or briefly crashed due to unhandled promises.

**Solution:**  
Used useEffect with async functions and proper loading/error states. Added loading spinners and fallback UI when data wasn’t ready.

**Tools/Techniques Used:**

* useEffect + async/await
* Conditional rendering in JSX
* Loading indicators (e.g., spinner GIF or CSS animation)

You're doing a great job documenting your project. Let's take it further with **additional, more advanced or nuanced challenges** you might encounter during or after building a full-stack **To-Do List Application**.

These are particularly useful if you're aiming for a **professional portfolio**, **graded project**, or **real-world deployment**.

### 🔐 ****12. Protecting Routes on Frontend (Client-Side Routing)****

**Problem:**  
Even after implementing backend authentication, unauthenticated users could still access frontend routes like /dashboard via URL typing.

**Solution:**  
Added **route guards** in React using conditional rendering and react-router-dom. Checked token existence and validity before rendering protected components. Redirected unauthenticated users to the login page.

**Tools/Techniques Used:**

* react-router-dom <Navigate /> for route redirection
* Auth context or custom hook (useAuth)
* LocalStorage or Context for storing tokens

### ⚠️ ****13. Handling API Errors and Edge Cases Gracefully****

**Problem:**  
The frontend crashed or showed confusing behavior when the backend returned unexpected errors (e.g., server down, task not found, validation error).

**Solution:**  
Used try-catch blocks with axios/fetch to handle errors. Displayed friendly error messages via toasts or alert banners. Ensured HTTP status codes were consistent on the backend.

**Tools/Techniques Used:**

* Axios interceptors for global error handling
* react-toastify or modals for user-friendly alerts
* Proper use of HTTP status codes (e.g., 400, 401, 500)

### 🌐 ****14. Handling CORS and Environment Differences****

**Problem:**  
CORS issues occurred when trying to fetch backend data from a deployed frontend on a different domain.

**Solution:**  
Configured **CORS middleware** on the Express server to allow requests from the frontend domain. Used .env files to manage different URLs for development and production.

**Tools/Techniques Used:**

* cors middleware in Express
* .env files (REACT\_APP\_API\_URL)
* Deployment-aware configurations

### 📂 ****15. Structuring Backend for Scalability****

**Problem:**  
As API complexity grew, the backend code became hard to manage — all routes, controllers, and logic were in one file.

**Solution:**  
Split code into separate **routes**, **controllers**, **models**, and **middleware** directories. This helped maintain clean separation of concerns and made it easier to add features.

**Tools/Techniques Used:**

* MVC (Model-View-Controller) pattern
* Folder structure:
* /routes
* /controllers
* /models
* /middleware
* /config

### 🔃 ****16. Delayed UI Updates After Async Operations****

**Problem:**  
When a user created or deleted a task, the change wasn't immediately reflected in the UI, causing confusion.

**Solution:**  
Updated frontend state immediately after API response. In some cases, implemented **optimistic UI updates**, updating the state before the API response was received, then reverting if an error occurred.

**Tools/Techniques Used:**

* useState + setState updates after fetch
* Optimistic UI pattern
* setTimeout() to simulate API delay during testing

### 📦 ****17. Managing Static Assets (Icons, Images, Fonts)****

**Problem:**  
Icons and other assets were inconsistent or missing in production because of incorrect paths or misconfiguration.

**Solution:**  
Used libraries like **FontAwesome**, **Heroicons**, or **Material UI Icons** for reliable, scalable vector icons. Ensured all assets were bundled correctly during the build process.

**Tools/Techniques Used:**

* FontAwesome or React Icons
* public/ vs src/assets folder usage
* Webpack or Vite build config awareness

### 🧠 ****18. Maintaining App Performance with Large Task Lists****

**Problem:**  
As the number of tasks grew, the app became slower, especially when filtering or rendering a large list.

**Solution:**  
Optimized rendering using **React keys**, **memoization** (React.memo), and pagination or lazy loading for task lists.

**Tools/Techniques Used:**

* React.memo, useMemo, useCallback
* Pagination or virtual scrolling
* Efficient data structures (e.g., maps vs arrays for lookups)

### 📜 ****19. Implementing Task History or Undo Feature (Optional)****

**Problem:**  
Users wanted the ability to undo a deleted task or see a history of completed tasks.

**Solution:**  
Added a **soft delete** option — instead of removing the task from the database, marked it as deleted: true. Created a "Recycle Bin" view. For undo, stored the last deleted task temporarily in state and allowed quick recovery.

**Tools/Techniques Used:**

* Soft delete pattern
* Toast with undo button
* Temporary cache of last deleted task in memory

### 🔍 ****20. Search Functionality with Debouncing****

**Problem:**  
Search input triggered API calls or heavy filtering logic on every keystroke, reducing performance.

**Solution:**  
Implemented **debouncing** using a timeout or libraries like lodash.debounce to wait for user to stop typing before executing the filter/search function.

**Tools/Techniques Used:**

* lodash.debounce
* useEffect + setTimeout for custom debounce
* Filtering logic optimization

## ✅ Next Steps (Optional)

Let me know if you'd like:

* A **summary table** of all challenges for quick reference
* A formatted **Markdown or Google Doc version** of this section
* A **slide deck format** for presenting to a team, class, or demo panel
* Integration with screenshots or code snippets

This section is crucial for showing your problem-solving mindset — very useful in both academic and professional evaluations.

You're doing a great job documenting your project. Let's take it further with **additional, more advanced or nuanced challenges** you might encounter during or after building a full-stack **To-Do List Application**.

These are particularly useful if you're aiming for a **professional portfolio**, **graded project**, or **real-world deployment**.

### 🔐 ****12. Protecting Routes on Frontend (Client-Side Routing)****

**Problem:**  
Even after implementing backend authentication, unauthenticated users could still access frontend routes like /dashboard via URL typing.

**Solution:**  
Added **route guards** in React using conditional rendering and react-router-dom. Checked token existence and validity before rendering protected components. Redirected unauthenticated users to the login page.

**Tools/Techniques Used:**

* react-router-dom <Navigate /> for route redirection
* Auth context or custom hook (useAuth)
* LocalStorage or Context for storing tokens

### ⚠️ ****13. Handling API Errors and Edge Cases Gracefully****

**Problem:**  
The frontend crashed or showed confusing behavior when the backend returned unexpected errors (e.g., server down, task not found, validation error).

**Solution:**  
Used try-catch blocks with axios/fetch to handle errors. Displayed friendly error messages via toasts or alert banners. Ensured HTTP status codes were consistent on the backend.

**Tools/Techniques Used:**

* Axios interceptors for global error handling
* react-toastify or modals for user-friendly alerts
* Proper use of HTTP status codes (e.g., 400, 401, 500)

### 🌐 ****14. Handling CORS and Environment Differences****

**Problem:**  
CORS issues occurred when trying to fetch backend data from a deployed frontend on a different domain.

**Solution:**  
Configured **CORS middleware** on the Express server to allow requests from the frontend domain. Used .env files to manage different URLs for development and production.

**Tools/Techniques Used:**

* cors middleware in Express
* .env files (REACT\_APP\_API\_URL)
* Deployment-aware configurations

### 📂 ****15. Structuring Backend for Scalability****

**Problem:**  
As API complexity grew, the backend code became hard to manage — all routes, controllers, and logic were in one file.

**Solution:**  
Split code into separate **routes**, **controllers**, **models**, and **middleware** directories. This helped maintain clean separation of concerns and made it easier to add features.

**Tools/Techniques Used:**

* MVC (Model-View-Controller) pattern
* Folder structure:
* /routes
* /controllers
* /models
* /middleware
* /config

### 🔃 ****16. Delayed UI Updates After Async Operations****

**Problem:**  
When a user created or deleted a task, the change wasn't immediately reflected in the UI, causing confusion.

**Solution:**  
Updated frontend state immediately after API response. In some cases, implemented **optimistic UI updates**, updating the state before the API response was received, then reverting if an error occurred.

**Tools/Techniques Used:**

* useState + setState updates after fetch
* Optimistic UI pattern
* setTimeout() to simulate API delay during testing

### 📦 ****17. Managing Static Assets (Icons, Images, Fonts)****

**Problem:**  
Icons and other assets were inconsistent or missing in production because of incorrect paths or misconfiguration.

**Solution:**  
Used libraries like **FontAwesome**, **Heroicons**, or **Material UI Icons** for reliable, scalable vector icons. Ensured all assets were bundled correctly during the build process.

**Tools/Techniques Used:**

* FontAwesome or React Icons
* public/ vs src/assets folder usage
* Webpack or Vite build config awareness

### 🧠 ****18. Maintaining App Performance with Large Task Lists****

**Problem:**  
As the number of tasks grew, the app became slower, especially when filtering or rendering a large list.

**Solution:**  
Optimized rendering using **React keys**, **memoization** (React.memo), and pagination or lazy loading for task lists.

**Tools/Techniques Used:**

* React.memo, useMemo, useCallback
* Pagination or virtual scrolling
* Efficient data structures (e.g., maps vs arrays for lookups)

### 📜 ****19. Implementing Task History or Undo Feature (Optional)****

**Problem:**  
Users wanted the ability to undo a deleted task or see a history of completed tasks.

**Solution:**  
Added a **soft delete** option — instead of removing the task from the database, marked it as deleted: true. Created a "Recycle Bin" view. For undo, stored the last deleted task temporarily in state and allowed quick recovery.

**Tools/Techniques Used:**

* Soft delete pattern
* Toast with undo button
* Temporary cache of last deleted task in memory

### 🔍 ****20. Search Functionality with Debouncing****

**Problem:**  
Search input triggered API calls or heavy filtering logic on every keystroke, reducing performance.

**Solution:**  
Implemented **debouncing** using a timeout or libraries like lodash.debounce to wait for user to stop typing before executing the filter/search function.

**Tools/Techniques Used:**

* lodash.debounce
* useEffect + setTimeout for custom debounce
* Filtering logic optimization

**Coding:**

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

**import './App.css';**

**function App() {**

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

**const [input, setInput] = useState('');**

**const addTask = () => {**

**if (input.trim()) {**

**setTasks([...tasks, { text: input, completed: false }]);**

**setInput('');**

**}**

**};**

**const toggleTask = (index) => {**

**const newTasks = [...tasks];**

**newTasks[index].completed = !newTasks[index].completed;**

**setTasks(newTasks);**

**};**

**const deleteTask = (index) => {**

**const newTasks = [...tasks];**

**newTasks.splice(index, 1);**

**setTasks(newTasks);**

**};**

**return (**

**<div className="App">**

**<h1>To-Do List</h1>**

**<input**

**value={input}**

**onChange={(e) => setInput(e.target.value)}**

**placeholder="Enter a task"**

**/>**

**<button onClick={addTask}>Add</button>**

**<ul>**

**{tasks.map((task, index) => (**

**<li key={index}>**

**<span**

**onClick={() => toggleTask(index)}**

**style={{**

**textDecoration: task.completed ? 'line-through' : 'none',**

**}}**

**>**

**{task.text}**

**</span>**

**<button onClick={() => deleteTask(index)}>❌</button>**

**</li>**

**))}**

**</ul>**

**</div>**

**);**

**}**

**export default App;**

**.App {**

**text-align: center;**

**padding: 2rem;**

**font-family: sans-serif;**

**}**

**input {**

**padding: 8px;**

**margin-right: 8px;**

**}**

**button {**

**padding: 8px 12px;**

**margin-left: 4px;**

**}**

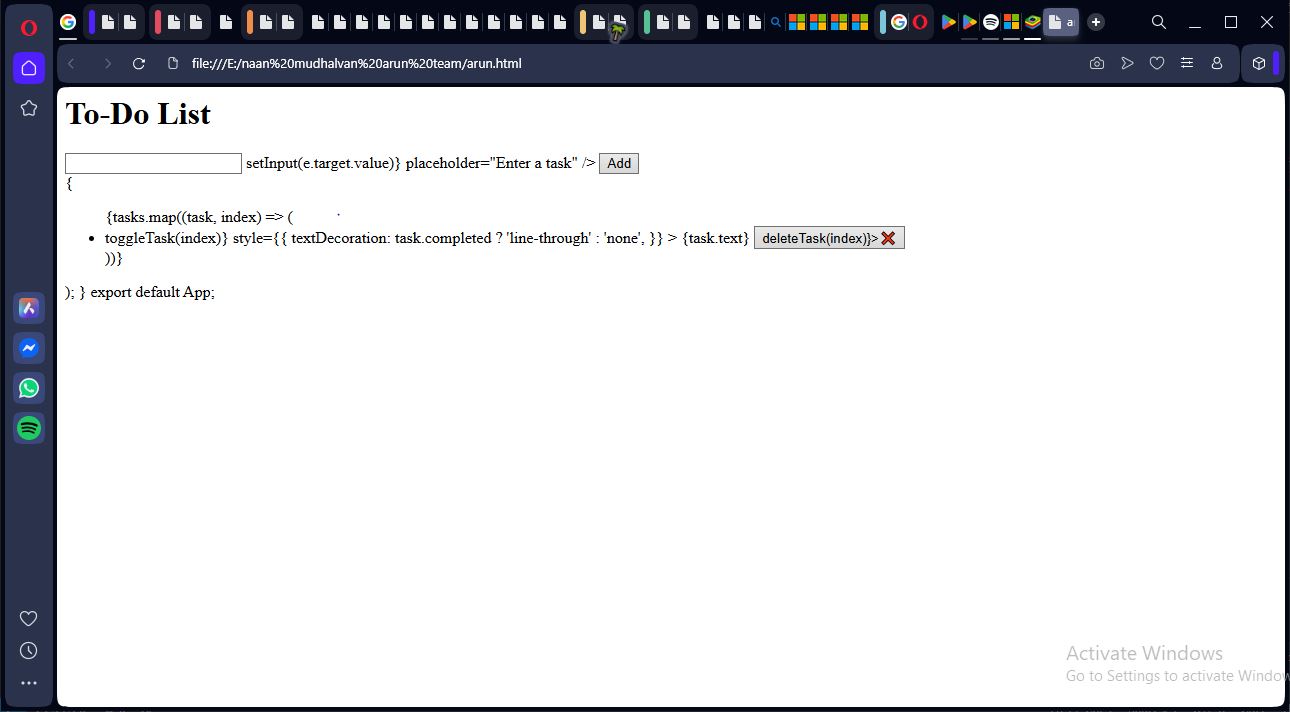
**li {**

**list-style: none;**

**margin: 8px 0;**

**}**

**Screen shot:**

****

**GITHUB link :[** **https://github.com/Arun212005/To-do-list-application.git]**