# Building a React To-Do App with useEffect

## Objective

*This hands-on tutorial is designed to guide you through building a functional React.js To-Do application from scratch. By the end of this tutorial, you will have created an interactive task management interface that allows users to add, display, and delete tasks dynamically.*

*Additionally, you will learn how to fetch and display external API data using the* `useEffect` *hook, enabling your app to handle side effects such as data fetching in a clean and efficient manner. This tutorial emphasizes practical skills including:*

- *Managing component state with the* `useState` *hook*
- *Handling side effects and asynchronous data fetching with* `useEffect`
- *Implementing dynamic UI updates based on user interactions and data changes*

*These foundational skills will prepare you to build React applications that efficiently manage state and integrate external data sources.*

## Prerequisites

Before starting this tutorial, ensure you have the following knowledge and tools ready to create the React To-Do application:

- **Basic JavaScript:** Understanding of variables, functions, arrays, and ES6 features like arrow functions, destructuring, and modules.
- **React.js Fundamentals:** Familiarity with React components, JSX syntax, and using hooks such as `useState`.
- **Node.js and npm:** Installed on your system to manage packages and run the development server.
- **Development Environment:** A code editor like Visual Studio Code set up for JavaScript and React development.
- **Internet Connectivity:** Required to fetch data from external APIs during the application run.
- **React Development Setup:** Knowledge of creating projects using Create React App or similar tools is helpful.

Having these prerequisites ensures you can focus on learning React state management and the `useEffect` hook smoothly throughout the tutorial.

## Project Setup

To begin building your React To-Do application, you first need to set up a new React project using Create React App (CRA), a comfortable environment that comes pre-configured with everything you need to start coding immediately.

### Open the Integrated Terminal

1. Launch VS Code

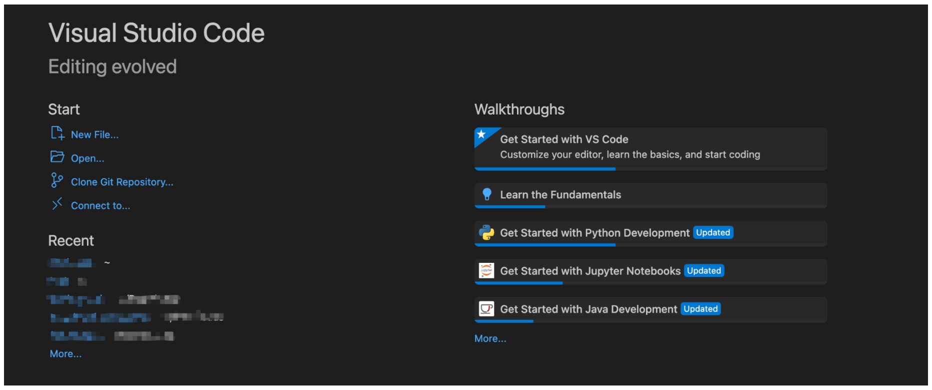

Visual Studio Code
Editing evolved
Start
New File...
Open...
Clone Git Repository...
Connect to...
Recent
More...
Walkthroughs
Get Started with VS Code
Customize your editor, learn the basics, and start coding
Learn the Fundamentals
Get Started with Python Development Updated
Get Started with Jupyter Notebooks Updated
Get Started with Java Development Updated
More...

***Figure 1**: VS Code Default Window*

2. Open your project folder or create a new one.

   a. Click on “Open”.

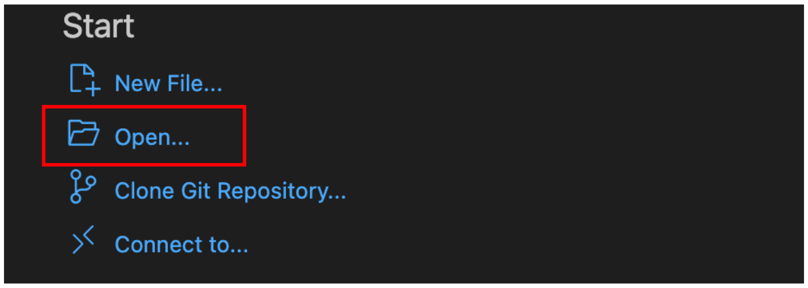


***Figure 2**: Opening the project directory in VS Code*

   b. Select the folder and click on “Open”.

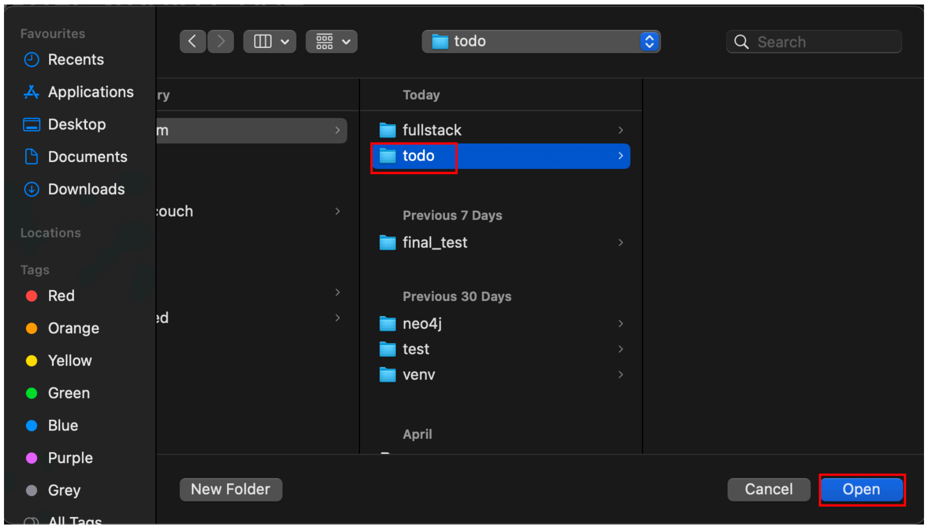
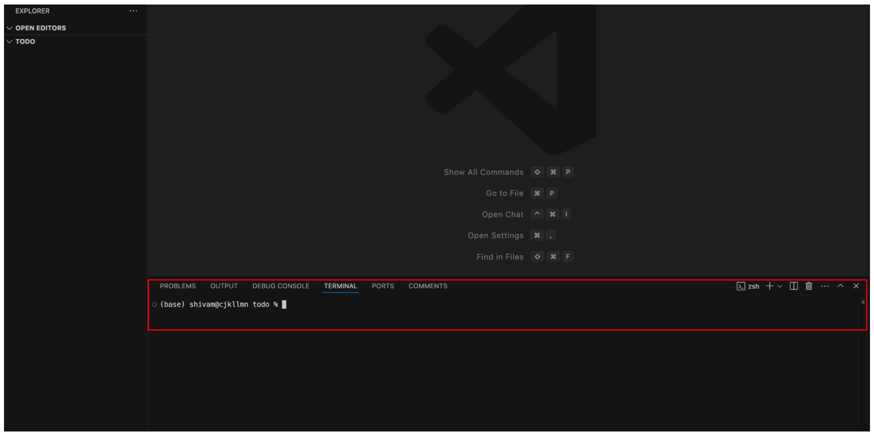


***Figure 3**: Selecting the  project directory*

3. Open the terminal with Ctrl+` (backtick)



```
(base) shivam@cjkllmn todo %
```

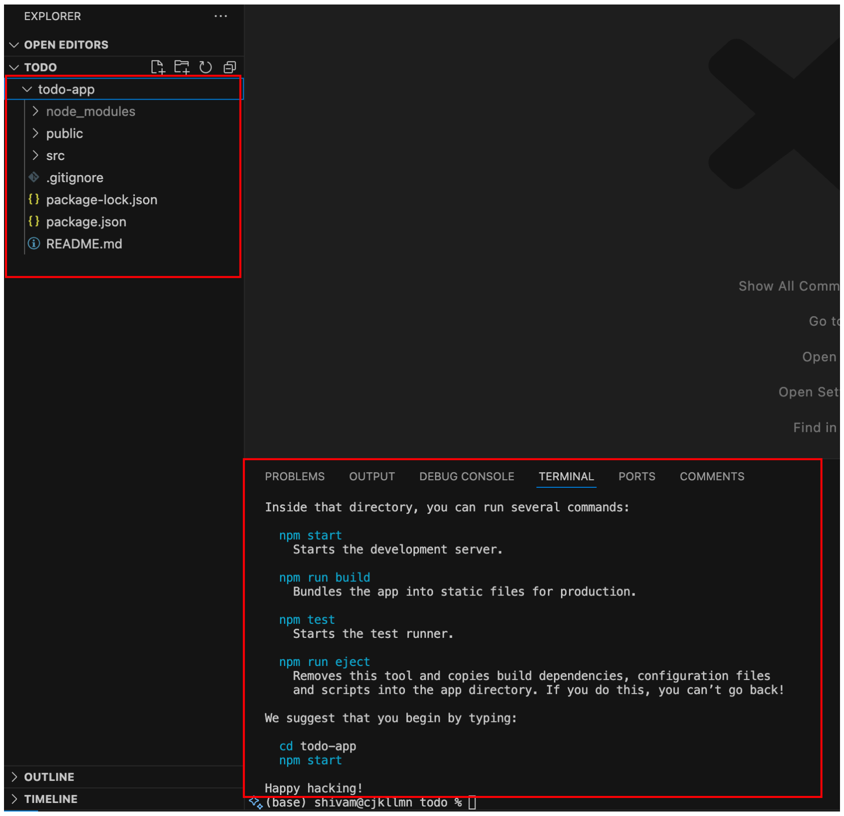
***Figure 4**: VS Code's Terminal*

Follow these steps to create and run your project:

1. **Create the React app:** Open your VS Code's terminal and run the following command to create a new React project named `todo-app`:

```
npx create-react-app todo-app
```

Enter "**y**" to proceed further.



```
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd todo-app
  npm start

Happy hacking!
(base) shivam@cjkllmn todo %
```

***Figure 4**: Successful creation of a React app*

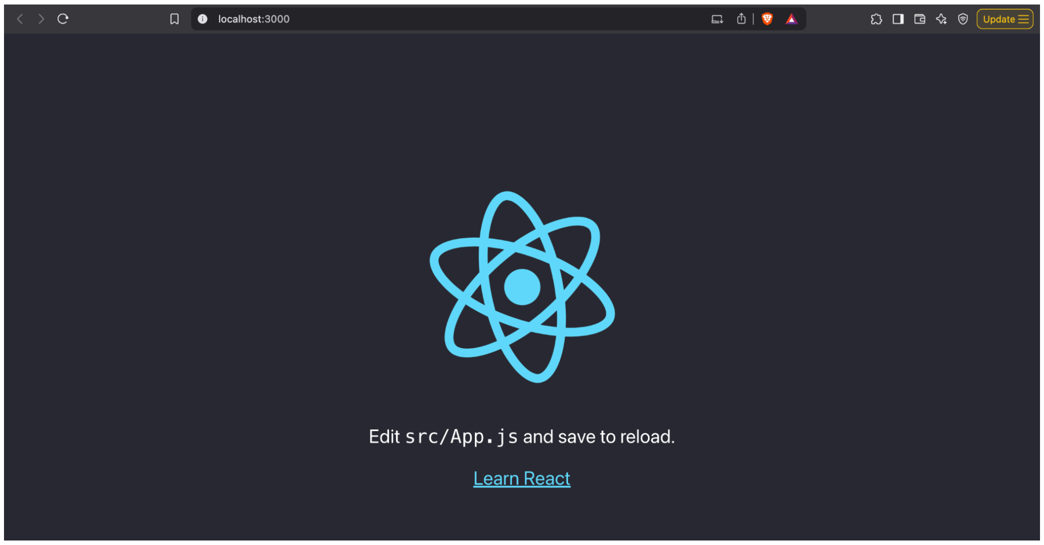
This command downloads and sets up the latest React environment along with all necessary dependencies. It might take a few minutes depending on your internet speed.

1. **Navigate into the project folder:** Move into the newly created project directory by running:

```
cd todo-app
```

2. **Start the development server:** Launch the React app locally using:

```
npm start
```



Edit `src/App.js` and save to reload.

***Figure 5**: React app's UI*

This command starts a development server and automatically opens your default browser to `http://localhost:3000`. Here, you can see your React app running live, and any code changes will refresh the page in real time.

1. **Open your project in a code editor:** Use Visual Studio Code or your preferred editor to open the `todo-app` folder. The main source code files are located inside the `src` directory, where you will build your components and add functionality.

For this basic To-Do application, no additional dependencies are required beyond what Create React App provides by default. This ensures a smooth experience focused solely on React fundamentals, state management, and fetching data with `useEffect`.

## Building the To-Do Application

In this section, we will build the core of our application: the `ToDoApp` component. This component will allow users to add new tasks, display the list of existing tasks, and delete tasks as needed. We will achieve this by leveraging React's `useState` hook to manage state and simple event handlers for user interactions.

### Step 1: Create the ToDoApp Functional Component

First, let's create a new functional component named `ToDoApp`. This will be the main component handling the entire To-Do functionality.

Create a components directory inside src and name it “todo.jsx”

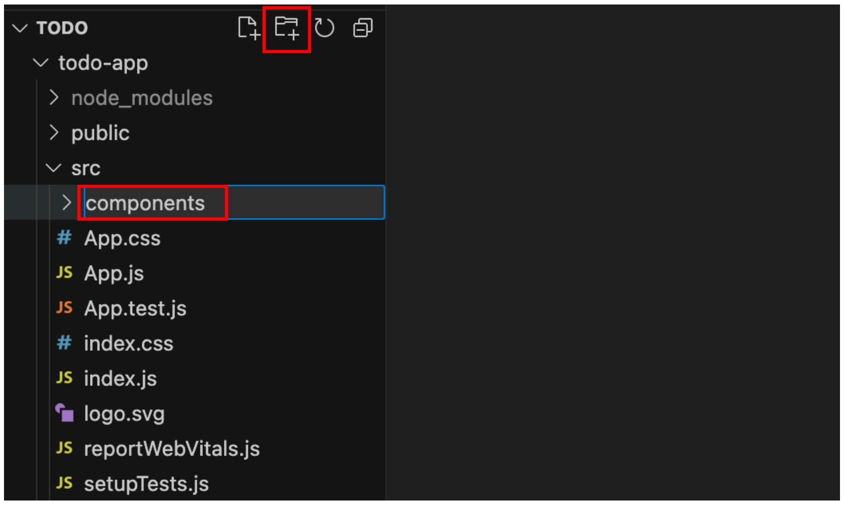


***Figure 6**: components directory creation*

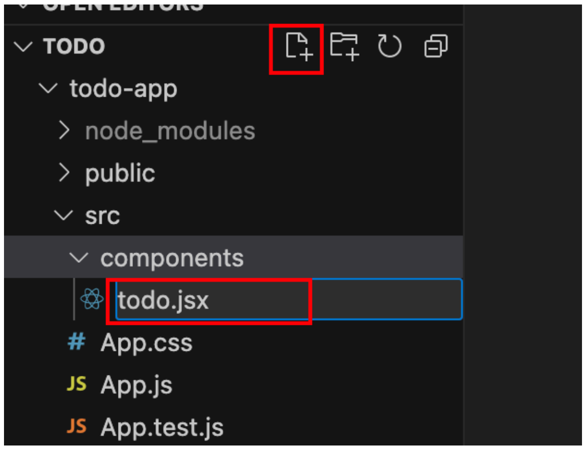


***Figure 7**: todo.jsx creation*

```
import React from 'react';

function ToDoApp() {
  return (
    <div>
      <h2>My To-Do List</h2>
    </div>
  );
}
export default ToDoApp;
```

Here, we simply return a container with a heading. In the next steps, we'll add input fields and functionality.

## Step 2: Add State for Tasks and Current Input

Add the states inside the function. We need two pieces of state:

- **tasks**: an array to store all to-do items.
- **taskInput**: a string representing the current value of the input field.

We use the `useState` hook from React to create and update these state variables.

```
import React from 'react';
import { useState } from 'react';

function ToDoApp() {

  //Added states
  const [tasks, setTasks] = useState([]);
  const [taskInput, setTaskInput] = useState('');

  const handleInputChange = (e) => {
    setTaskInput(e.target.value);
  };

  const handleAddTask = () => {
    if (taskInput.trim() === '') return; // Prevent adding empty tasks
    setTasks([...tasks, taskInput.trim()]);
    setTaskInput('');
  };


  return (
    <div>
      <h2>My To-Do List</h2>
    </div>
  );
}
export default ToDoApp;
```

With this setup, `tasks` start as an empty array, and `taskInput` is initially an empty string.

## Step 3: Add Input Field and Button to Add Tasks

Next, add an input field where users can type a new task, and a button to add this task to the list.

The input field needs to:

- Display the current `taskInput` value.
- Update `taskInput` state when the user types.

The button will trigger a function to add the input as a new task. ToDoApp function looks like this:

```
import React from 'react';
import { useState } from 'react';

function ToDoApp() {
  const [tasks, setTasks] = useState([]);
  const [taskInput, setTaskInput] = useState('');

  const handleInputChange = (e) => {
    setTaskInput(e.target.value);
  };

  const handleAddTask = () => {
    if (taskInput.trim() === '') return; // Prevent adding empty tasks
    setTasks([...tasks, taskInput.trim()]);
    setTaskInput('');
  };

  return (
    <div>
      <h2>My To-Do List</h2>  // Added input field
      <input
        type="text"
        placeholder="Enter new task"
        value={taskInput}
        onChange={handleInputChange}
      />
      <button onClick={handleAddTask}>Add Task</button> // Buttons
    </div>
  );
}

export default ToDoApp;
```

**Explanation:**

- `handleInputChange` updates the `taskInput` whenever the user types.

- `handleAddTask` checks if the input is not empty, adds the new task to the `tasks` array (using the spread operator to keep existing tasks), and clears the input field.

## Step 4: Display the List of Tasks

Now let's render the list of tasks below the input. We will map over the `tasks` array and display each item in an unordered list (`<ul>`) with a delete button next to it.

For deletion we will be creating a handleDeleteTask state:

```
 const handleAddTask = () => {
    if (taskInput.trim() === '') return;
    setTasks([...tasks, taskInput.trim()]);
    setTaskInput('');
  };
```

```
import React from 'react';
import { useState } from 'react';

function ToDoApp() {
  const [tasks, setTasks] = useState([]);
  const [taskInput, setTaskInput] = useState('');

  const handleInputChange = (e) => {
    setTaskInput(e.target.value);
  };

  const handleAddTask = () => {
    if (taskInput.trim() === '') return;
    setTasks([...tasks, taskInput.trim()]);
    setTaskInput('');
  };

  const handleDeleteTask = (index) => {
    const newTasks = tasks.filter((_, i) => i !== index);
    setTasks(newTasks);
  };
  return (
    <div>
      <h2>My To-Do List</h2>   // Added input field
      <input
        type="text"
        placeholder="Enter new task"
        value={taskInput}
        onChange={handleInputChange}
      />
      <button onClick={handleAddTask}>Add Task</button>  // Buttons

      <ul>  // Unordered listed items added
        {tasks.map((task, index) => (
          <li key={index}>
            {task}
            <button
              onClick={() => handleDeleteTask(index)}
              style={{ marginLeft: '10px' }}
            >Delete</button>
          </li>
        ))}
      </ul>
    </div>
  );
}
export default ToDoApp;
```

### Step 5: Explaining Task Deletion

The `handleDeleteTask` function takes the index of the task to delete and filters it out from the array. Then it updates the state with the new filtered list, and React re-renders the UI without that task.

We pass the current index to the delete button's `onClick` handler so it knows which task to remove.

### Step 6: Include todo component in App.js

```
import './App.css';
import ToDoApp from './components/todo';

function App() {
 return (
   <div className="App">
    <ToDoApp/>
   </div>
 );
}

export default App;
```

## Summary

At this point, your `ToDoApp` component supports:

- Entering new tasks in the input box.
- Adding tasks to the state-managed list.
- Displaying all tasks dynamically as a list.
- Deleting individual tasks using their associated delete buttons.

This completes the core functionality of your To-Do application. In the next section, we will enhance it by fetching tasks from an external API using the `useEffect` hook.

**Run:**

```
npm start
```

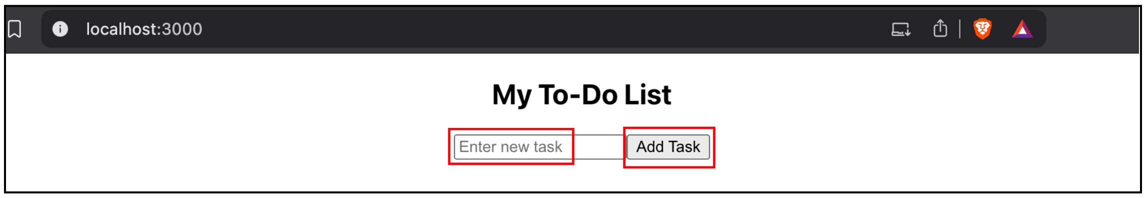


***Figure 8**: Basic To do list UI*

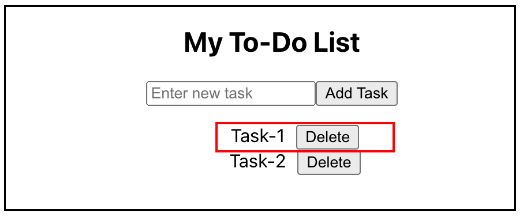


***Figure 9**: Creation and Deletion functionality*

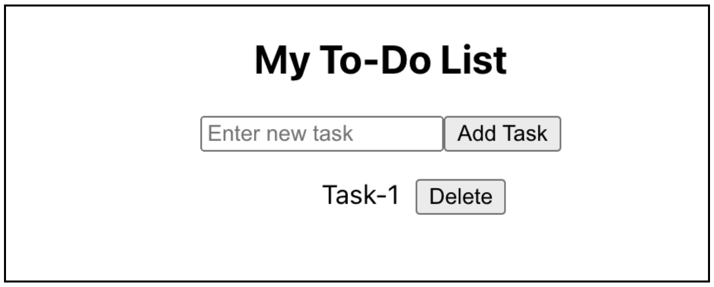


***Figure 10**: Successful Deletion*

## Fetching API Data with useEffect

To enhance our To-Do application, we will now fetch a list of tasks from a public API and display them alongside our local tasks. React's `useEffect` hook is ideal for running side effects like data fetching when a component mounts or updates.

### Step 1: Setting Up State for Fetched Tasks

First, we create state variables to hold the API data, a loading indicator, and any potential errors during fetching:

```
const [fetchedTasks, setFetchedTasks] = useState([]);
const [isLoading, setIsLoading] = useState(false);
const [fetchError, setFetchError] = useState(null);
```

- `fetchedTasks` stores the array of tasks retrieved from the API.
- `isLoading` tracks whether the fetch is in progress.
- `fetchError` captures any error message if fetching fails.

### Step 2: Using useEffect to Fetch Data

Next, we use the `useEffect` hook to fetch data when the component first mounts. We provide an empty dependencies array `[]` to ensure this effect runs only once:

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

function ToDoApp() {
  // ...existing state declarations...
  const [fetchedTasks, setFetchedTasks] = useState([]);
  const [isLoading, setIsLoading] = useState(false);
  const [fetchError, setFetchError] = useState(null);

  useEffect(() => {
       setIsLoading(true);
       fetch('https://dummyjson.com/todos?limit=5')
         .then(response => {
           if (!response.ok) {
             throw new Error('Network response was not ok');
           }
           return response.json();
         })
         .then(data => {
           setFetchedTasks(data.todos); // correct assignment
           setFetchError(null);
         })
         .catch(error => {
           setFetchError(error.message);
         })
         .finally(() => {
           setIsLoading(false);
         });
     }, []);
  // Empty array ensures fetch runs only once on mount
  // Rest of the code as it is
```

**Explanation:**

- `setIsLoading(true)` signals the start of data fetching.
- `fetch()` calls the API endpoint `https://jsonplaceholder.typicode.com/todos` limiting the result to 5 tasks.
- We check if the response is OK; otherwise, we throw an error to be caught later.
- `setFetchedTasks(data)` updates the state with the fetched tasks.
- `setFetchError(error.message)` captures any errors during the fetch.
- `finally()` sets `isLoading` to false, ending the loading state.
- The empty dependency array `[]` means this effect runs only once after the initial render.

## Step 3: Rendering API Data with Loading and Error Handling

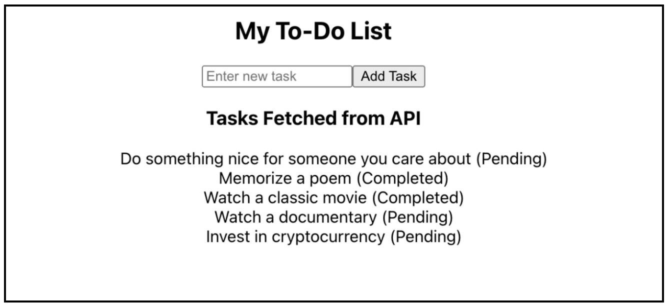
Now, update the JSX to show:

- A loading message or spinner while data is fetching.
- An error message if the fetch fails.
- The fetched list of tasks after successful data retrieval.

```
return (
  <div>
    <h2>My To-Do List</h2>

    {/* existing input, add, and local tasks UI here */}

    {/* add below code as it is from here: */}
    <h3>Tasks Fetched from API</h3>

    {isLoading && <p>Loading tasks from API...</p>}
    {fetchError && <p style={{ color: 'red' }}>Error: {fetchError}</p>}

    {!isLoading && !fetchError && (
      <ul>
        {fetchedTasks.map(task => (
          <li key={task.id}>
            {task.todo} {task.completed ? '(Completed)' : '(Pending)'}
          </li>
        ))}
      </ul>
    )}
    {/* till here */}
  </div>
);
```

**Explanation:**

- While `isLoading` is true, a simple loading message informs the user.
- If an error occurs, its message is displayed in red below the header.
- Once loading is done with no errors, the fetched tasks are listed using `map()`, showing each task's title and completion status.


My To-Do List

Tasks Fetched from API

Do something nice for someone you care about (Pending)
Memorize a poem (Completed)
Watch a classic movie (Completed)
Watch a documentary (Pending)
Invest in cryptocurrency (Pending)

***Figure 11****: Tasks fetched from API successfully*

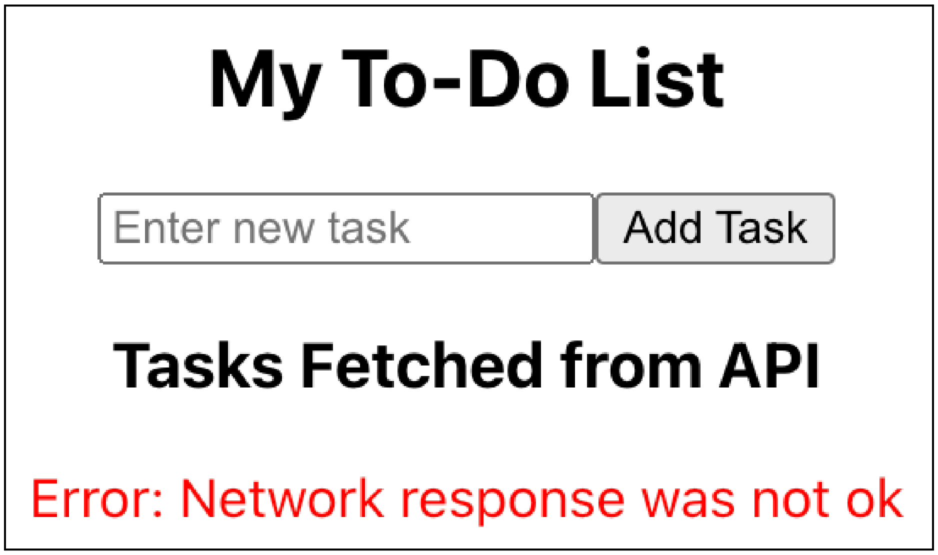
### Why Use the Dependencies Array?

The second argument to `useEffect` is critical:

- An empty array (`[]`) means the effect runs once after the component mounts, preventing repeated fetches on every render.
- If omitted, the fetch would run after every update, causing unnecessary network requests and performance issues.
- You can add variables inside the array to re-run the effect only when those variables change.

This controlled fetching ensures your app is efficient and responsive.

You can edit the URL to something non-existing to verify the error handling.


My To-Do List

Tasks Fetched from API

Error: Network response was not ok

***Figure 12****: Error Handling*

## Styling (Optional)

To improve the visual clarity of your To-Do app, you can add minimal styling using plain CSS or CSS-in-JS. Below is a simple example of CSS styles focused on readability and basic interactivity.

**Sample CSS Styles**
Add this in src/App.css or your chosen stylesheet

```css
.container {
  max-width: 600px;
  margin: 20px auto;
  padding: 15px;
  border: 1px solid #ddd;
  border-radius: 8px;
  background-color: #fafafa;
  font-family: Arial, sans-serif;
}

input[type="text"] {
  width: 70%;
  padding: 8px;
  margin-right: 8px;
  border: 1px solid #ccc;
  border-radius: 4px;
}

button {
  padding: 8px 12px;
  border: none;
  background-color: #007bff;
  color: white;
  border-radius: 4px;
  cursor: pointer;
  transition: background-color 0.3s ease;
}

button:hover {
  background-color: #0056b3;
}

ul {
  list-style-type: none;
  padding-left: 0;
  margin-top: 15px;
}

li {
  padding: 8px;
  margin-bottom: 6px;
  background: #fff;
  border: 1px solid #ddd;
  border-radius: 4px;
  display: flex;
  justify-content: space-between;
  align-items: center;
}

li button {
  background-color: #dc3545;
}

li button:hover {
  background-color: #a71d2a;
}
```

**Where to place these styles:**

- *External CSS file:* Save the styles in `src/App.css` or a custom CSS file, then import it into your component file with `import './App.css';`.
- *Inline styles:* You can also apply styles using the `style` attribute in JSX but using a stylesheet keeps the code cleaner.
- *CSS Modules or CSS-in-JS:* For scoped or dynamic styling, you may use CSS Modules or styled-components, though plain CSS is simpler for beginners.

Wrap your To-Do app's JSX inside a container with the `container` class to apply the layout styles. These minimal styles will improve your app's user experience by visually separating tasks and making buttons intuitive to interact with.

Final **todo.jsx:**

```
import React from 'react';
import { useState, useEffect } from 'react';
import '../App.css'
function ToDoApp() {
```

```
const [fetchedTasks, setFetchedTasks] = useState([]);
const [isLoading, setIsLoading] = useState(false);
const [fetchError, setFetchError] = useState(null);

useEffect(() => {
    setIsLoading(true);
    fetch('https://dummyjson.com/todos?limit=5')
      .then(response => {
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
        return response.json();
      })
      .then(data => {
        setFetchedTasks(data.todos); // correct assignment
        setFetchError(null);
      })
      .catch(error => {
        setFetchError(error.message);
      })
      .finally(() => {
        setIsLoading(false);
      });
  }, []);

const [tasks, setTasks] = useState([]);
const [taskInput, setTaskInput] = useState('');
 const handleInputChange = (e) => {
  setTaskInput(e.target.value);
};
 const handleAddTask = () => {
  if (taskInput.trim() === '') return;
  setTasks([...tasks, taskInput.trim()]);
  setTaskInput('');
};
 const handleDeleteTask = (index) => {
  const newTasks = tasks.filter((_, i) => i !== index);
  setTasks(newTasks);
};
 return (
  <div className='container'>
    <h2>My To-Do List</h2>
    <input
```

```
          type="text"
          placeholder="Enter new task"
          value={taskInput}
          onChange={handleInputChange}
        />
        <button onClick={handleAddTask}>Add Task</button>
         <ul>
          {tasks.map((task, index) => (
            <li key={index}>
              {task}
              <button
                onClick={() => handleDeleteTask(index)}
                style={{ marginLeft: '10px' }}
              >Delete</button>
            </li>
          ))}
        </ul>

        <h3>Tasks Fetched from API</h3>
        {isLoading && <p>Loading tasks from API...</p>}
        {fetchError && <p style={{ color: 'red' }}>Error: {fetchError}</p>}

        {!isLoading && !fetchError && (
        <ul>
            {fetchedTasks.map(task => (
            <li key={task.id}>
                {task.todo} {task.completed ? '(Completed)' : '(Pending)'}
            </li>
            ))}
        </ul>
        )}

      </div>
    );
  }

export default ToDoApp;
```

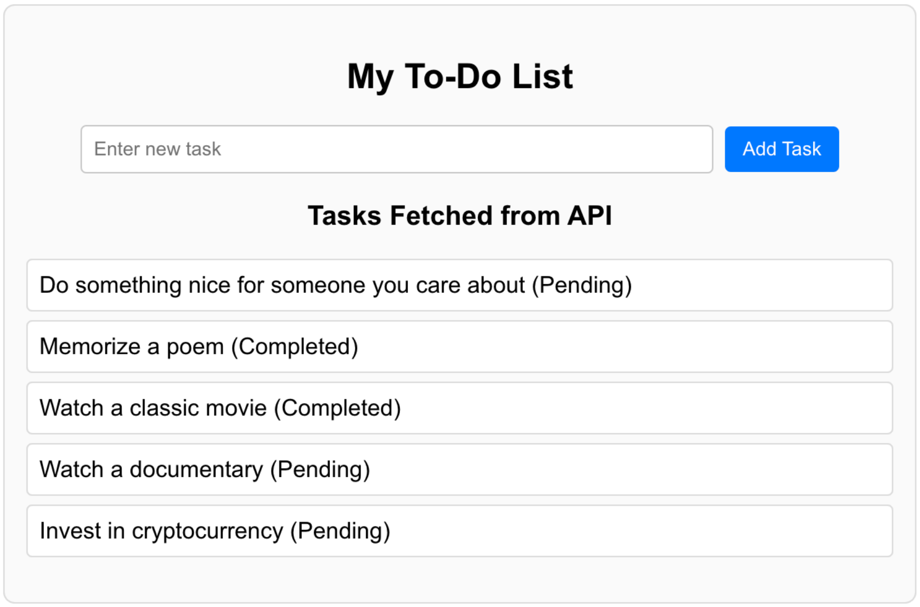
## Running and Testing

To run your React To-Do application, open your terminal and navigate to the project directory (e.g., `todo-app`). Then, start the development server by running:

```
npm start
```

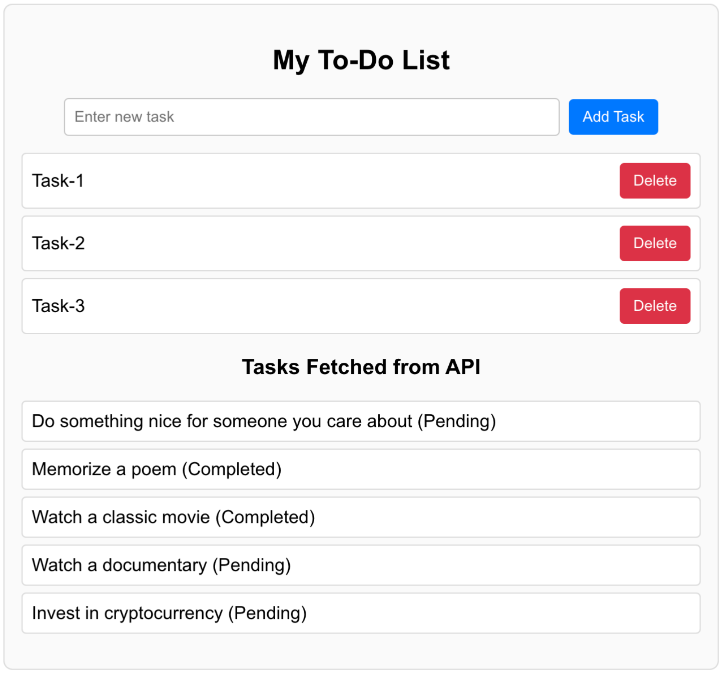
This command launches the app at `http://localhost:3000` and opens it in your default web browser. Here's what you should expect to see:

- An input field to enter new tasks and an "Add Task" button.
- A dynamically updating list displaying all added tasks, each with a delete button.
- A separate list showing tasks fetched from the external API, preceded by a loading message if data is still being retrieved.
- An error message if the API fetch fails for any reason.



Do something nice for someone you care about (Pending)
Memorize a poem (Completed)
Watch a classic movie (Completed)
Watch a documentary (Pending)
Invest in cryptocurrency (Pending)

***Figure 13**: Final UI after CSS addition*


My To-Do List

Task-1

Task-2

Task-3

Tasks Fetched from API

Do something nice for someone you care about (Pending)

Memorize a poem (Completed)

Watch a classic movie (Completed)

Watch a documentary (Pending)

Invest in cryptocurrency (Pending)

***Figure 14**: Fully working ToDo list*

To test your app's functionality, try the following steps:

1. **Add tasks:** Type a task into the input and click "Add Task." The new task should immediately appear in the local task list.
2. **Delete tasks:** Click the "Delete" button next to any task and confirm it is removed from the list.
3. **Verify fetched tasks:** Scroll to the API tasks section and ensure the fetched task titles appear once loading completes.
4. **Observe loading and error states:** If you disable your internet connection and refresh, you should see an error message where API data would normally display.

## Troubleshooting tips:

- If the app does not open automatically in your browser, manually visit `http://localhost:3000`.
- Ensure Node.js and npm are installed and up to date.
- If changes do not appear on save, check that the development server is running and your files are saved correctly.
- Review the browser console for any error messages if the app behaves unexpectedly.

Following these steps will help you confirm that your To-Do app is fully functional and correctly integrates state management with API data fetching.

## Conclusion

*Congratulations! You have successfully built a basic React To-Do application that demonstrates essential React concepts including state management with* `useState` *and side effect handling with* `useEffect`*. Throughout this tutorial, you learned how to:*

- *Create functional components to organize your UI*
- *Manage and update task lists dynamically using React state*
- *Fetch external data from a public API and display it within your app*
- *Implement loading indicators and error handling for asynchronous operations*

*These skills form a strong foundation for developing interactive and responsive React applications. By understanding hooks and component design, you can now confidently explore more advanced features and build projects with real-world data integrations.*

*Keep practicing by enhancing this app further or starting new React projects to deepen your proficiency and creativity. With consistent effort, you'll continue growing as a React developer—great work on completing this tutorial!*