EXPERIMENT NO. - 03

Roll No.	24
Name	Chaitanya Dinesh Dhayarkar
Class	D15B
Subject	Full Stack Development
Lab Outcome	Experiment based on Manage complex state with Redux or Context API
Date of Performance/ Submission	
Signature & Grades	

EXPERIMENT NO: 3

Aim: Manage complex state with Redux or Context API

Theory:

1. **Redux Overview:** Redux is a predictable state management library for React, centralizing application state in a single, immutable store.
2. **Single Source of Truth:** The entire application state is stored in one JavaScript object, ensuring consistency and accessibility.
3. **Unidirectional Data Flow:** Actions, which are plain JavaScript objects, describe state changes and are dispatched to the store.
4. **Reducers:** Pure functions that process actions to compute the new state, ensuring predictable and traceable updates.
5. **Middleware for Async Operations:** Tools like Redux Thunk or Redux Saga handle asynchronous tasks, such as API calls, within Redux.
6. **API Integration in React:** APIs are typically accessed using Fetch or Axios, with requests managed in useEffect hooks (functional components) or lifecycle methods (class components).
7. **State Management with APIs:** Fetched API data can be stored in component state or the Redux store for global access, enabling dynamic UI updates.
8. **Async Flow in Redux:** A common pattern involves dispatching actions for API request start, success, or failure, allowing Redux to update state and trigger React component re-renders.

30% Extra: -

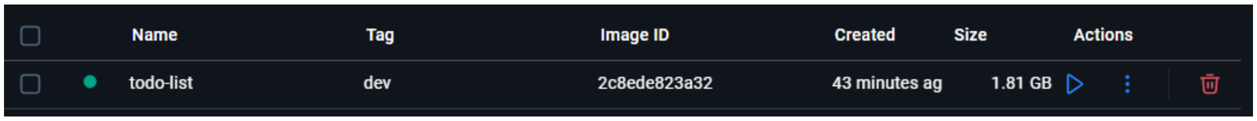
Docker is a platform used to package applications and their dependencies into containers.

- A **container** is a lightweight, standalone, and executable unit that includes everything needed to run the application: code, runtime, libraries, and system tools.
- This ensures that the application runs identically on any system, regardless of the host environment.

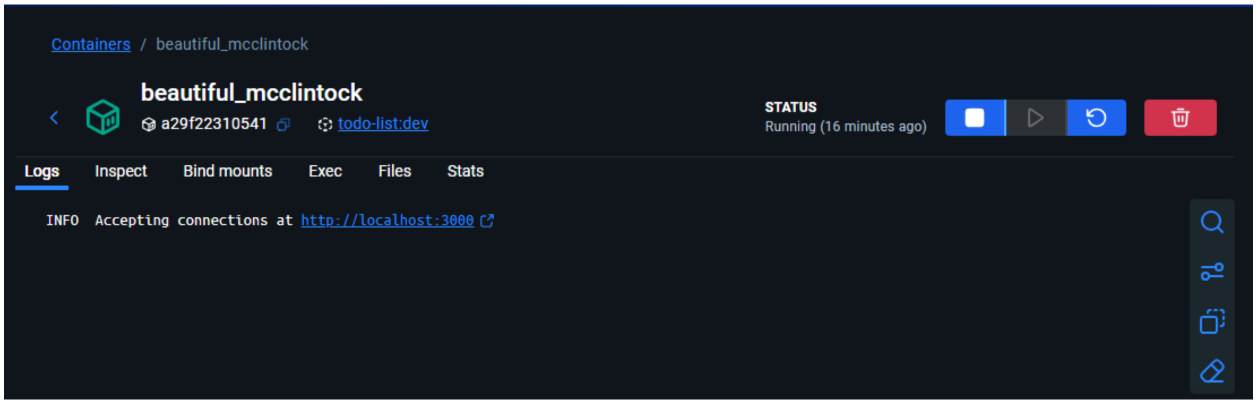
1. **Create Dockerfile**
   In the root of your project, create a file named Dockerfile:
   ```
   FROM node:18-alpine
   WORKDIR /app
   COPY package*.json ./
   RUN npm install
   COPY . .
   RUN npm run build
   EXPOSE 3000
   CMD ["npx", "serve", "-s", "build"]
   ```

2. **BuildDocker Image**
   Run the following command in the terminal:
   ```
   docker build -t todo-list:dev .
   ```

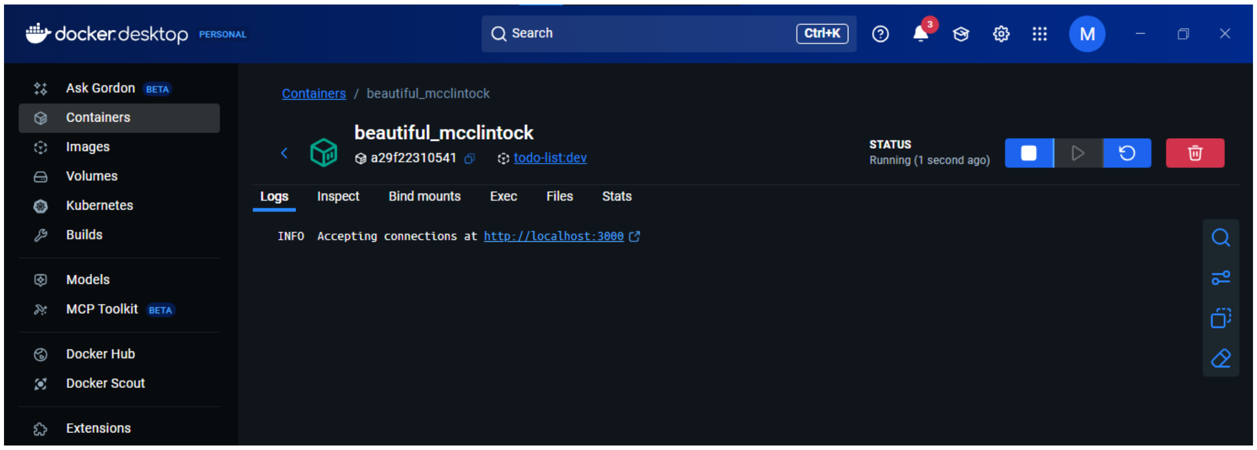
3. **Run Docker Container**
   Start the container and expose it on port 3000:
   docker run -p 3000:3000 todo-list:dev
4. **Access the Application**
   Open a browser and visit: *http://localhost:3000*



	Name	Tag	Image ID	Created	Size	Actions
●	todo-list	dev	2c8ede823a32	43 minutes ag	1.81 GB	

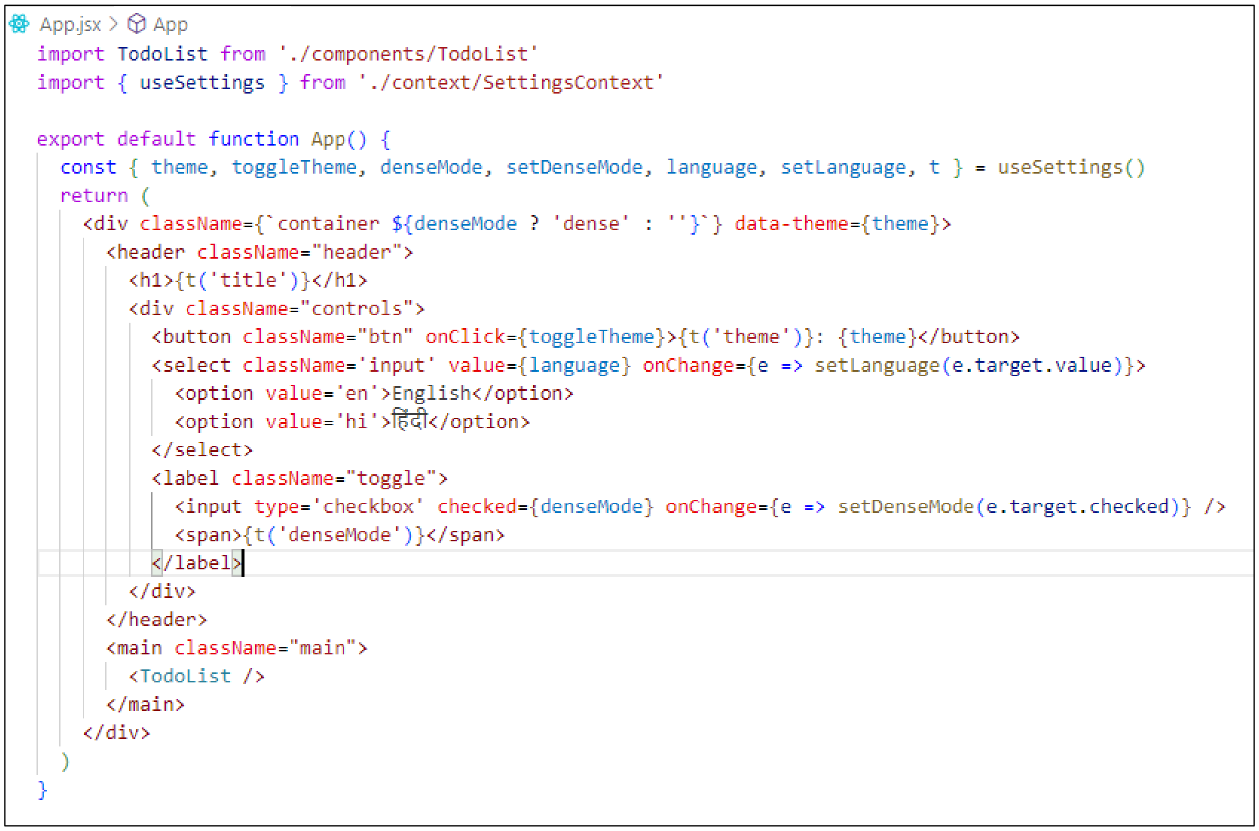


```
INFO  Accepting connections at http://localhost:3000
```



```
INFO  Accepting connections at http://localhost:3000
```

Source Code:



```
import TodoList from './components/TodoList'
import { useSettings } from './context/SettingsContext'

export default function App() {
  const { theme, toggleTheme, denseMode, setDenseMode, language, setLanguage, t } = useSettings()
  return (
    <div className={`container ${denseMode ? 'dense' : ''}`} data-theme={theme}>
      <header className="header">
        <h1>{t('title')}</h1>
        <div className="controls">
          <button className="btn" onClick={toggleTheme}>{t('theme')}: {theme}</button>
          <select className='input' value={language} onChange={e => setLanguage(e.target.value)}>
            <option value='en'>English</option>
            <option value='hi'>हिंदी</option>
          </select>
          <label className="toggle">
            <input type='checkbox' checked={denseMode} onChange={e => setDenseMode(e.target.checked)} />
            <span>{t('denseMode')}</span>
          </label>
        </div>
      </header>
      <main className="main">
        <TodoList />
      </main>
    </div>
  )
}
```

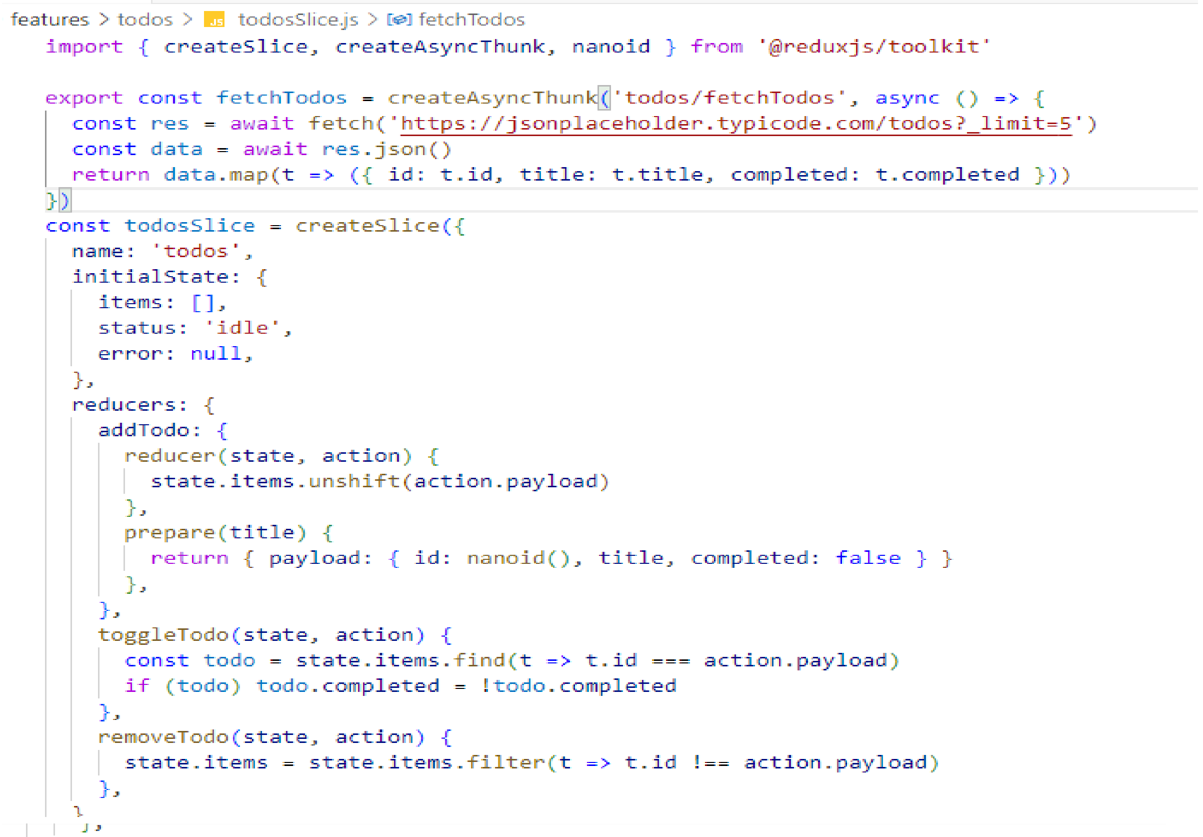
SettingsProvider.jsx (Set Context)

```
import { createContext, useContext, useMemo, useState, useEffect } from 'react'
const SettingsContext = createContext(null)

export function SettingsProvider({ children }) {
  const [theme, setTheme] = useState(() => localStorage.getItem('theme') || 'light')
  const [denseMode, setDenseMode] = useState(false)
  const [language, setLanguage] = useState(() => localStorage.getItem('lang') || 'en')

  useEffect(() => {
    localStorage.setItem('theme', theme)
    document.documentElement.dataset.theme = theme
  }, [theme])

  useEffect(() => {
    localStorage.setItem('lang', language)
    document.documentElement.lang = language
  }, [language])

  const value = useMemo(() => {
    const translations = {
      en: {
        title: 'Redux + Context Demo',
        theme: 'Theme',
        denseMode: 'Dense mode',
        add: 'Add',
        placeholder: 'Add todo',
        loading: 'Loading…',
      },
      hi: {
        title: 'Redux + Context डेमो',
        theme: 'थीम',
        denseMode: 'घना मोड',
        add: 'जोड़ें',
        placeholder: 'कार्य जोड़ें',
        loading: 'लोड हो रहा है…',
      },
    }
    const t = key => (translations[language] && translations[language][key]) || key
    return {
      theme,
      setTheme,
      denseMode,
      setDenseMode,
      toggleTheme: () => setTheme(t0 => (t0 === 'light' ? 'dark' : 'light')),
      language,
      setLanguage,
      t,
    }
  }, [theme, denseMode, language])

  return <SettingsContext.Provider value={value}>{children}</SettingsContext.Provider>
}
export function useSettings() {
  const ctx = useContext(SettingsContext)
  if (!ctx) throw new Error('useSettings must be used within SettingsProvider')
  return ctx;
}
```

TodoList.jsx

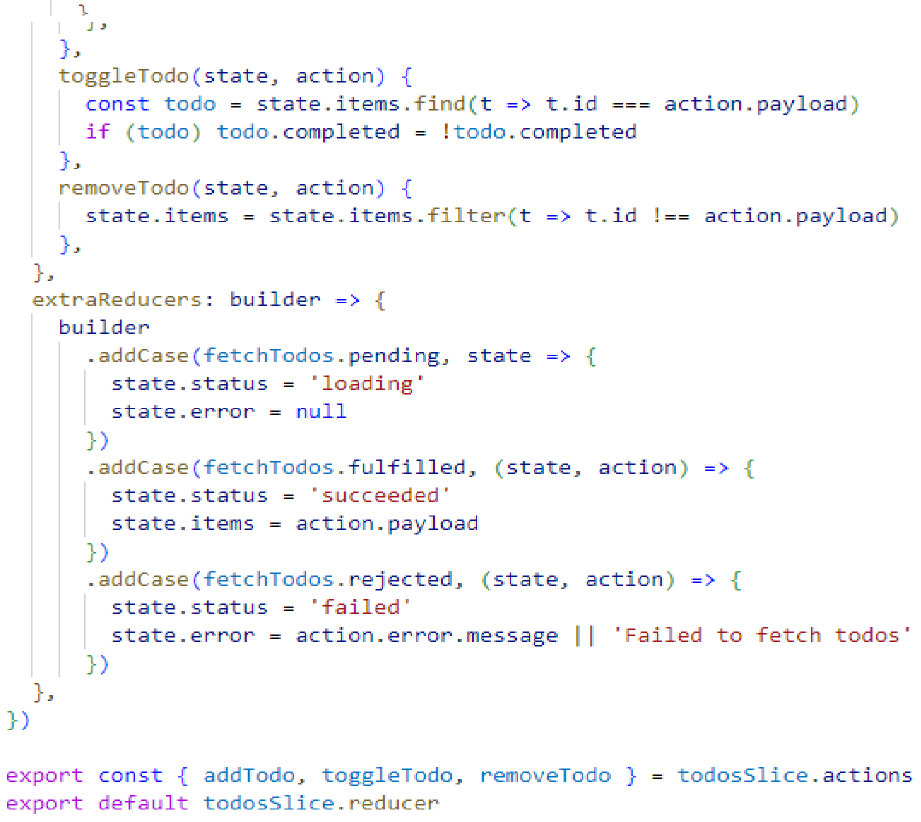
```
import { useEffect, useState } from 'react'
import { useDispatch, useSelector } from 'react-redux'
import { addTodo, toggleTodo, removeTodo, fetchTodos } from '../features/todos/todosSlice'
import { useSettings } from '../context/SettingsContext'

export default function TodoList() {
  const dispatch = useDispatch()
  const { items, status, error } = useSelector(s => s.todos)
  const [input, setInput] = useState('')
  const { t } = useSettings()

  useEffect(() => {
    if (status === 'idle') dispatch(fetchTodos())
  }, [status, dispatch])

  return (
    <div className='card'>
      <form className='form' onSubmit={e => { e.preventDefault(); if (input.trim()) {
dispatch(addTodo(input.trim())); setInput('') } }}>
        <input className='input' placeholder={t('placeholder')} value={input} onChange={e =>
setInput(e.target.value)} />
        <button className='btn primary' type='submit'>{t('add')}</button>
      </form>
      {status === 'loading' && <p className='muted'>{t('loading')}</p>}
      {status === 'failed' && <p className='error'>{error}</p>}
      <ul className='list'>
        {items.map(t => (
          <li className='list-item' key={t.id}>
            <label className='todo'>
              <input type='checkbox' checked={t.completed} onChange={() => dispatch(toggleTodo(t.id))} />
              <span className={t.completed ? 'done' : ''}>{t.title}</span>
            </label>
            <button className='btn danger' onClick={() => dispatch(removeTodo(t.id))}>×</button>
          </li>
        ))}
      </ul> </div> ) }
```

Todo’s Action



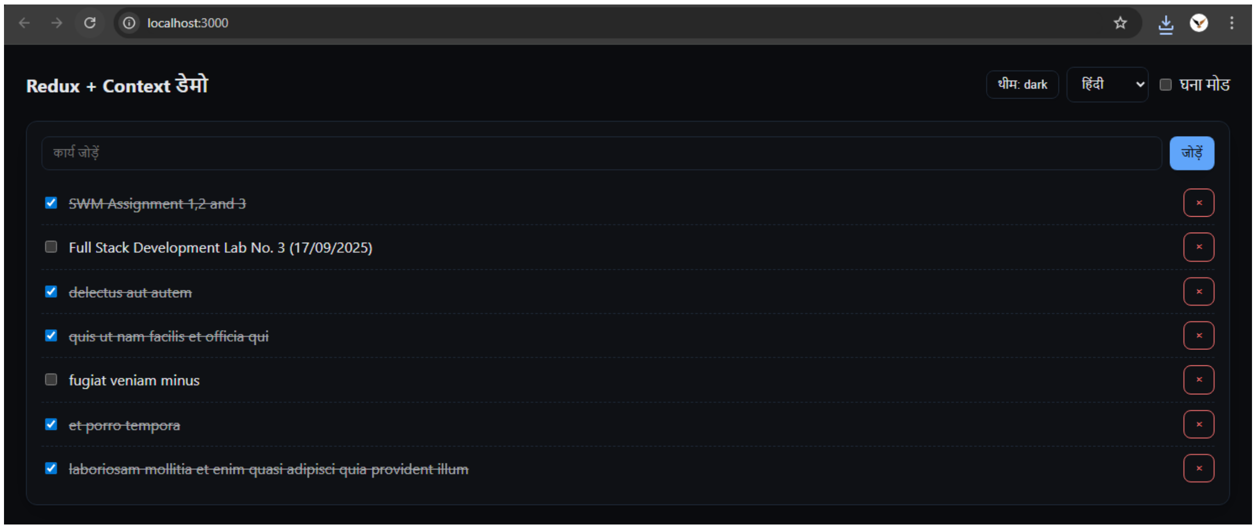
features > todos > todosSlice.js > fetchTodos

```
import { createSlice, createAsyncThunk, nanoid } from '@reduxjs/toolkit'

export const fetchTodos = createAsyncThunk('todos/fetchTodos', async () => {
  const res = await fetch('https://jsonplaceholder.typicode.com/todos?_limit=5')
  const data = await res.json()
  return data.map(t => ({ id: t.id, title: t.title, completed: t.completed }))
})
const todosSlice = createSlice({
  name: 'todos',
  initialState: {
    items: [],
    status: 'idle',
    error: null,
  },
  reducers: {
    addTodo: {
      reducer(state, action) {
        state.items.unshift(action.payload)
      },
      prepare(title) {
        return { payload: { id: nanoid(), title, completed: false } }
      },
    },
    toggleTodo(state, action) {
      const todo = state.items.find(t => t.id === action.payload)
      if (todo) todo.completed = !todo.completed
    },
    removeTodo(state, action) {
      state.items = state.items.filter(t => t.id !== action.payload)
    },
  },
```



```
    },
    toggleTodo(state, action) {
      const todo = state.items.find(t => t.id === action.payload)
      if (todo) todo.completed = !todo.completed
    },
    removeTodo(state, action) {
      state.items = state.items.filter(t => t.id !== action.payload)
    },
  },
  extraReducers: builder => {
    builder
      .addCase(fetchTodos.pending, state => {
        state.status = 'loading'
        state.error = null
      })
      .addCase(fetchTodos.fulfilled, (state, action) => {
        state.status = 'succeeded'
        state.items = action.payload
      })
      .addCase(fetchTodos.rejected, (state, action) => {
        state.status = 'failed'
        state.error = action.error.message || 'Failed to fetch todos'
      })
  },
})

export const { addTodo, toggleTodo, removeTodo } = todosSlice.actions
export default todosSlice.reducer
```

Output:


localhost:3000

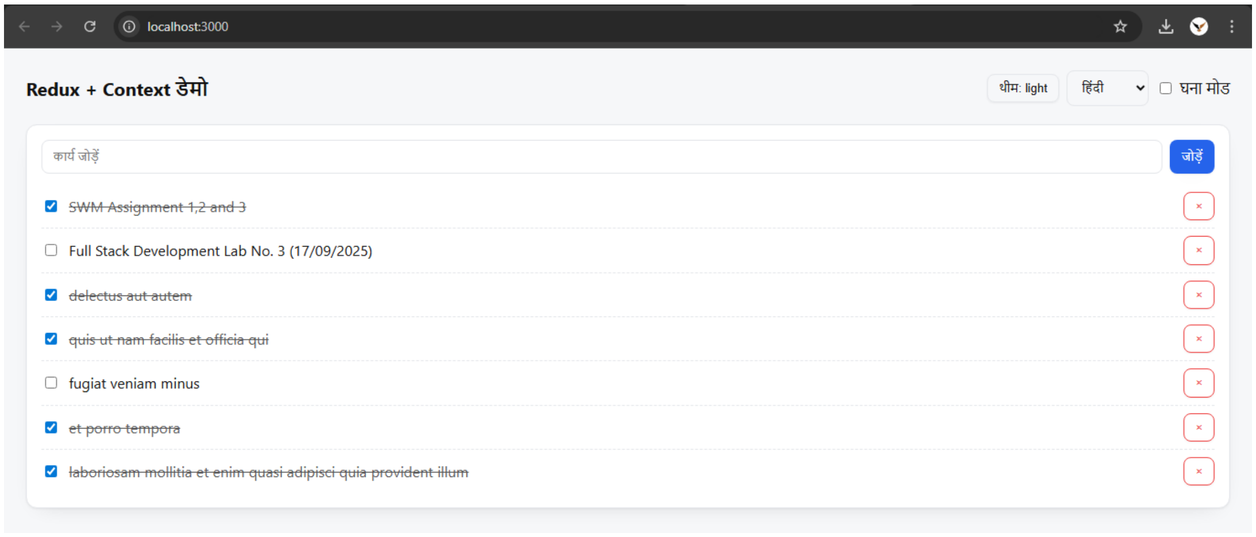
Redux + Context डेमो

थीम: dark

हिंदी

घना मोड

- [x] ~~SWM Assignment 1,2 and 3~~
- [ ] Full Stack Development Lab No. 3 (17/09/2025)
- [x] ~~delectus aut autem~~
- [x] ~~quis ut nam facilis et officia qui~~
- [ ] fugiat veniam minus
- [x] ~~et porro tempora~~
- [x] ~~laboriosam mollitia et enim quasi adipisci quia provident illum~~


localhost:3000

Redux + Context डेमो

थीम: light

हिंदी

घना मोड

- [x] ~~SWM Assignment 1,2 and 3~~
- [ ] Full Stack Development Lab No. 3 (17/09/2025)
- [x] ~~delectus aut autem~~
- [x] ~~quis ut nam facilis et officia qui~~
- [ ] fugiat veniam minus
- [x] ~~et porro tempora~~
- [x] ~~laboriosam mollitia et enim quasi adipisci quia provident illum~~

Conclusion:

In exploring Redux and API integration in React, we gain a deeper understanding of how to manage state and handle asynchronous data flows in modern web applications. Redux provides a robust framework for maintaining a predictable, centralized state through actions, reducers, and a single store, making it easier to track and debug state changes. When combined with API calls, typically managed via tools like Fetch or Axios within React's useEffect or lifecycle methods, Redux enables seamless handling of asynchronous operations through middleware like Redux Thunk or Saga. This combination ensures that data fetched from external services is efficiently integrated into the application's state, driving dynamic UI updates. By leveraging Redux for state management and carefully structuring API interactions, developers can build scalable, maintainable, and performant React applications. Understanding these tools and their interplay highlights the importance of thoughtful state management and asynchronous data handling for creating responsive and reliable user experiences.