React

1.

**Step-by-Step Implementation:**

**1. Set up the App Component:**

javascript

CopyEdit

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

const TodoApp = () => {

// State to hold tasks

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

// State to hold the new task input value

const [newTask, setNewTask] = useState("");

// Load tasks from local storage on initial render

useEffect(() => {

const storedTasks = JSON.parse(localStorage.getItem("todos"));

if (storedTasks) {

setTasks(storedTasks);

}

}, []);

// Sync tasks with local storage whenever they change

useEffect(() => {

localStorage.setItem("todos", JSON.stringify(tasks));

}, [tasks]);

// Add a new task

const addTask = () => {

if (newTask.trim()) {

setTasks([

...tasks,

{ id: Date.now(), text: newTask, completed: false }

]);

setNewTask(""); // Clear input field

}

};

// Delete a task

const deleteTask = (id) => {

setTasks(tasks.filter(task => task.id !== id));

};

// Toggle the completion status of a task

const toggleCompletion = (id) => {

setTasks(tasks.map(task =>

task.id === id ? { ...task, completed: !task.completed } : task

));

};

// Edit a task's text

const editTask = (id, newText) => {

setTasks(tasks.map(task =>

task.id === id ? { ...task, text: newText } : task

));

};

return (

<div className="todo-app">

<h1>Todo List</h1>

<input

type="text"

value={newTask}

onChange={(e) => setNewTask(e.target.value)}

placeholder="Enter a new task"

/>

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

<ul>

{tasks.map((task) => (

<li key={task.id} className={task.completed ? "completed" : ""}>

<span

onClick={() => toggleCompletion(task.id)}

style={{ textDecoration: task.completed ? "line-through" : "none" }}

>

{task.text}

</span>

<button onClick={() => editTask(task.id, prompt("Edit task", task.text))}>Edit</button>

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

</li>

))}

</ul>

</div>

);

};

export default TodoApp;

**Explanation:**

1. **State Management (useState)**:
   * tasks: Holds the list of todo tasks.
   * newTask: Holds the input value for new tasks.
2. **LocalStorage Sync (useEffect)**:
   * On initial render (useEffect with empty dependency array), we check if there’s any data in local storage and load it into the tasks state.
   * Another useEffect ensures that whenever tasks changes, it is saved back to local storage.
3. **Add Task (addTask)**:
   * This function adds a new task with a unique ID (using Date.now()) and a completed flag initialized to false.
4. **Delete Task (deleteTask)**:
   * This function removes a task based on its id.
5. **Toggle Completion (toggleCompletion)**:
   * It toggles the completion status of the task (crossing out the text).
6. **Edit Task (editTask)**:
   * The user can edit the task's text using a simple prompt.

**Styling (optional):**

You can style the todo list as follows:

css

CopyEdit

.todo-app {

font-family: Arial, sans-serif;

max-width: 500px;

margin: 0 auto;

padding: 20px;

}

input[type="text"] {

padding: 10px;

margin-bottom: 10px;

width: 80%;

border: 1px solid #ccc;

}

button {

padding: 10px;

background-color: #4CAF50;

color: white;

border: none;

cursor: pointer;

margin-left: 10px;

}

button:hover {

background-color: #45a049;

}

ul {

list-style-type: none;

padding: 0;

}

li {

padding: 10px;

border-bottom: 1px solid #ddd;

}

li.completed span {

text-decoration: line-through;

color: grey;

}

button {

background-color: red;

padding: 5px 10px;

margin-left: 10px;

}

button:hover {

background-color: darkred;

}

2.

**Step-by-Step Implementation:**

**1. Set up the Dynamic Form Component:**

javascript

CopyEdit

import React, { useState } from "react";

const DynamicForm = () => {

// State to store the form fields

const [fields, setFields] = useState([

{ id: Date.now(), name: "", email: "", age: "" } // Initial empty field object

]);

// Handle input change in a specific field

const handleInputChange = (index, event) => {

const newFields = [...fields];

newFields[index][event.target.name] = event.target.value; // Update the specific field

setFields(newFields); // Update state with the modified field

};

// Handle adding a new field

const handleAddField = () => {

setFields([

...fields,

{ id: Date.now(), name: "", email: "", age: "" } // Add a new empty field object

]);

};

// Handle removing a field

const handleRemoveField = (index) => {

const newFields = fields.filter((\_, i) => i !== index); // Remove the field at the specific index

setFields(newFields); // Update state with the remaining fields

};

// Handle form submission

const handleSubmit = (event) => {

event.preventDefault();

console.log("Form Data Submitted: ", fields);

// Send the fields data to the backend (API call can be added here)

};

return (

<div className="dynamic-form">

<h1>Dynamic Form</h1>

<form onSubmit={handleSubmit}>

{fields.map((field, index) => (

<div key={field.id} className="field">

<input

type="text"

name="name"

placeholder="Name"

value={field.name}

onChange={(event) => handleInputChange(index, event)}

/>

<input

type="email"

name="email"

placeholder="Email"

value={field.email}

onChange={(event) => handleInputChange(index, event)}

/>

<input

type="number"

name="age"

placeholder="Age"

value={field.age}

onChange={(event) => handleInputChange(index, event)}

/>

<button type="button" onClick={() => handleRemoveField(index)}>

Remove

</button>

</div>

))}

<button type="button" onClick={handleAddField}>

Add New Field

</button>

<button type="submit">Submit</button>

</form>

</div>

);

};

export default DynamicForm;

**Explanation:**

1. **State (useState)**:
   * fields: This state holds an array of field objects. Initially, it contains one empty object with keys (name, email, age) for the first form field.
2. **Dynamic Form Rendering**:
   * The fields.map() function is used to loop through each field and render the corresponding form fields (name, email, age).
   * Each form field is controlled, meaning its value is tied to the state (field.name, field.email, and field.age).
3. **Adding a Field (handleAddField)**:
   * When the "Add New Field" button is clicked, a new field object is added to the state, and the form re-renders with the new input fields.
4. **Removing a Field (handleRemoveField)**:
   * Clicking "Remove" next to a form field removes that specific field from the fields array using filter().
5. **Handling Input Change (handleInputChange)**:
   * Each input field has a name attribute, and when the input changes, the corresponding value in the fields array is updated.
   * The handleInputChange function ensures that only the specific field's value (name, email, or age) is updated based on the index.
6. **Form Submission (handleSubmit)**:
   * When the form is submitted, the fields data is logged to the console. You can replace this with an API call to send the data to the backend.

**Optional Styling (CSS):**

css

CopyEdit

.dynamic-form {

font-family: Arial, sans-serif;

max-width: 600px;

margin: 20px auto;

padding: 20px;

border: 1px solid #ddd;

border-radius: 10px;

}

input {

padding: 10px;

margin: 5px;

width: 100%;

margin-bottom: 10px;

border: 1px solid #ccc;

border-radius: 5px;

}

button {

padding: 10px 15px;

background-color: #4CAF50;

color: white;

border: none;

cursor: pointer;

border-radius: 5px;

margin-top: 10px;

}

button[type="button"] {

background-color: #f44336;

}

button:hover {

background-color: #45a049;

}

button[type="button"]:hover {

background-color: #d32f2f;

}

3.

**Step-by-Step Implementation:**

**1. Define State for Loading, Success, and Error:**

We need three pieces of state:

* **loading**: To track whether the data is still being fetched.
* **data**: To store the fetched data once it’s successfully retrieved.
* **error**: To store any error message if the fetch request fails.

**2. Use useEffect to Fetch Data:**

The **useEffect** hook will trigger the API call when the component mounts. You can fetch data asynchronously using **fetch** or **axios**. We’ll handle loading and error states based on the response.

**Example Implementation:**

javascript

CopyEdit

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

const UserList = () => {

// State to manage the API response, loading, and error state

const [users, setUsers] = useState([]);

const [loading, setLoading] = useState(true);

const [error, setError] = useState(null);

// useEffect to fetch data when component mounts

useEffect(() => {

// Initiate API request

fetch("https://jsonplaceholder.typicode.com/users")

.then((response) => {

// Check if the response is successful

if (!response.ok) {

throw new Error("Something went wrong while fetching data!");

}

return response.json();

})

.then((data) => {

setUsers(data); // Update state with the fetched data

setLoading(false); // Data has been fetched, so set loading to false

})

.catch((error) => {

setError(error.message); // If there is an error, set the error state

setLoading(false); // Stop the loading indicator

});

}, []); // Empty dependency array, so this runs only once when the component mounts

// Render loading, error, or the data

if (loading) {

return <div>Loading...</div>; // Show loading indicator

}

if (error) {

return <div>Error: {error}</div>; // Show error message

}

return (

<div>

<h1>User List</h1>

<ul>

{users.map((user) => (

<li key={user.id}>

{user.name} - {user.email}

</li>

))}

</ul>

</div>

);

};

export default UserList;

**Explanation:**

1. **State Variables**:
   * users: This holds the list of users fetched from the API.
   * loading: Initially set to true, it indicates that data is being fetched.
   * error: Stores any error message if the fetch request fails.
2. **useEffect Hook**:
   * The useEffect hook triggers when the component mounts (since we passed an empty dependency array []).
   * Inside the useEffect, we use the fetch API to get data from the remote API (https://jsonplaceholder.typicode.com/users in this case).
   * If the fetch request is successful, we set the users state with the fetched data and set loading to false.
   * If there's an error (e.g., network error, or bad response), we catch the error and set the error state with the error message. We also set loading to false in case of failure.
3. **Conditional Rendering**:
   * While the data is being fetched, the component renders the "Loading..." message.
   * If there's an error, it displays the error message.
   * If the data is successfully fetched, it maps through the users array and displays the user names and emails.

**Techniques Used:**

1. **useEffect**: This is used for side effects like fetching data when the component mounts. It runs the fetch request only once (when the component is first rendered) due to the empty dependency array [].
2. **State Management (useState)**:
   * We use three separate pieces of state: loading, data (i.e., users), and error.
   * These states are updated based on the results of the API request.
3. **Error Handling**:
   * We use .catch() to catch any errors in the fetch request, such as network errors or issues with the response status.
   * The error message is displayed when there's an issue with the fetch request.
4. **Conditional Rendering**:
   * We render different UI based on the states of loading, error, and data.

**Alternative Using axios:**

You can use axios for data fetching instead of the fetch API. Axios has a simpler syntax and automatically handles some things like JSON parsing.

javascript

CopyEdit

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

import axios from "axios";

const UserList = () => {

const [users, setUsers] = useState([]);

const [loading, setLoading] = useState(true);

const [error, setError] = useState(null);

useEffect(() => {

axios

.get("https://jsonplaceholder.typicode.com/users")

.then((response) => {

setUsers(response.data);

setLoading(false);

})

.catch((error) => {

setError(error.message);

setLoading(false);

});

}, []);

if (loading) {

return <div>Loading...</div>;

}

if (error) {

return <div>Error: {error}</div>;

}

return (

<div>

<h1>User List</h1>

<ul>

{users.map((user) => (

<li key={user.id}>

{user.name} - {user.email}

</li>

))}

</ul>

</div>

);

};

export default UserList;

**Note**: When using axios, it handles JSON parsing automatically, so you don’t need to call .json()

4.

**Steps to Implement Authentication:**

1. **Create a Login Form**:
   * Build a form with input fields for username and password.
   * Handle form submission to send the credentials to the API for validation.
2. **Manage Authentication State**:
   * Use useState to manage the form data (username, password) and the authentication state (whether the user is logged in or not).
   * You can use useContext or a global state management tool (like **Redux**) to share the authentication status across the app.
3. **API Request to Validate Credentials**:
   * Use **fetch** or **axios** to send the login request to your backend API.
   * Handle the response and redirect the user to the dashboard on successful login.
   * Show an error message if the credentials are incorrect.
4. **Store Authentication Token**:
   * If the login is successful, store the authentication token (e.g., JWT) in localStorage or sessionStorage.
   * Use this token for subsequent API requests to authenticate the user.
5. **Conditional Rendering for Protected Routes**:
   * Redirect unauthenticated users to the login page if they try to access protected routes.
   * Use **React Router** to handle the route navigation and manage the protected routes.

**Example Implementation:**

Here’s how you can implement user authentication in React:

**1. Login Form Component:**

javascript

CopyEdit

import React, { useState } from "react";

import { useHistory } from "react-router-dom"; // React Router for navigation

const Login = () => {

const [username, setUsername] = useState("");

const [password, setPassword] = useState("");

const [error, setError] = useState(""); // To store error message

const history = useHistory(); // To programmatically navigate to the dashboard

const handleSubmit = async (e) => {

e.preventDefault();

// Perform API request to validate credentials

try {

const response = await fetch("https://yourapi.com/login", {

method: "POST",

headers: {

"Content-Type": "application/json",

},

body: JSON.stringify({

username,

password,

}),

});

if (!response.ok) {

throw new Error("Invalid credentials");

}

const data = await response.json();

const { token } = data; // Assuming the API returns a JWT token

// Store the token in localStorage or sessionStorage

localStorage.setItem("authToken", token);

// Redirect to dashboard on successful login

history.push("/dashboard");

} catch (err) {

setError(err.message); // Show error message if login fails

}

};

return (

<div>

<h2>Login</h2>

<form onSubmit={handleSubmit}>

<div>

<label>Username</label>

<input

type="text"

value={username}

onChange={(e) => setUsername(e.target.value)}

required

/>

</div>

<div>

<label>Password</label>

<input

type="password"

value={password}

onChange={(e) => setPassword(e.target.value)}

required

/>

</div>

{error && <div style={{ color: "red" }}>{error}</div>} {/\* Display error message \*/}

<button type="submit">Login</button>

</form>

</div>

);

};

export default Login;

**2. Protected Route (Dashboard) Component:**

javascript

CopyEdit

import React from "react";

import { Redirect } from "react-router-dom";

const Dashboard = () => {

// Check if the user is authenticated by checking the token in localStorage

const token = localStorage.getItem("authToken");

if (!token) {

// Redirect to login page if user is not authenticated

return <Redirect to="/login" />;

}

return (

<div>

<h2>Welcome to the Dashboard</h2>

{/\* Other dashboard content \*/}

</div>

);

};

export default Dashboard;

**3. App Component with React Router:**

javascript

CopyEdit

import React from "react";

import { BrowserRouter as Router, Route, Switch } from "react-router-dom";

import Login from "./Login";

import Dashboard from "./Dashboard";

const App = () => {

return (

<Router>

<Switch>

<Route path="/login" component={Login} />

<Route path="/dashboard" component={Dashboard} />

<Redirect from="/" to="/login" />

</Switch>

</Router>

);

};

export default App;

5.

**Steps to Create a Reusable Modal Component:**

1. **Create the Modal Component**:
   * The modal should accept title, content, and onClose as props.
   * It should include the content (text, images, forms, etc.) and a close button to hide the modal.
2. **Show/Hide the Modal**:
   * Use state to control whether the modal is visible or not. You can use useState in the parent component to manage the modal's visibility.
   * Pass a function (onClose) to the modal to update the state and close the modal.
3. **Implement Modal Styling**:
   * Style the modal with CSS to make it appear as an overlay on top of the page content.
   * You can use position: fixed to position it centrally, and give it a background overlay to dim the background.
4. **Close Modal on Background Click**:
   * Add functionality so the modal can be closed when the user clicks outside the modal (on the background).
5. **Reusability**:
   * Design the modal component in a way that it can be reused anywhere by passing different content and titles dynamically.

**Example Implementation of a Reusable Modal Component:**

**1. Modal Component:**

javascript

CopyEdit

import React from 'react';

import './Modal.css'; // For modal styling

const Modal = ({ isVisible, title, content, onClose }) => {

if (!isVisible) return null; // If modal is not visible, don't render anything

return (

<div className="modal-overlay" onClick={onClose}> {/\* Modal background \*/}

<div className="modal-content" onClick={(e) => e.stopPropagation()}> {/\* Prevent click on modal content from closing \*/}

<div className="modal-header">

<h2>{title}</h2>

<button className="close-btn" onClick={onClose}>X</button>

</div>

<div className="modal-body">

{content} {/\* Dynamic content \*/}

</div>

</div>

</div>

);

};

export default Modal;

**2. Modal Styling (CSS):**

css

CopyEdit

/\* Modal.css \*/

.modal-overlay {

position: fixed;

top: 0;

left: 0;

width: 100%;

height: 100%;

background-color: rgba(0, 0, 0, 0.5); /\* Semi-transparent background \*/

display: flex;

justify-content: center;

align-items: center;

z-index: 1000; /\* Ensures modal is above other content \*/

}

.modal-content {

background-color: white;

padding: 20px;

border-radius: 5px;

width: 400px;

max-width: 90%;

box-shadow: 0px 2px 10px rgba(0, 0, 0, 0.2);

}

.modal-header {

display: flex;

justify-content: space-between;

align-items: center;

}

.modal-header h2 {

margin: 0;

}

.close-btn {

background: none;

border: none;

font-size: 1.5rem;

cursor: pointer;

}

.modal-body {

margin-top: 20px;

}

**3. Parent Component to Use the Modal:**

javascript

CopyEdit

import React, { useState } from 'react';

import Modal from './Modal'; // Import the Modal component

const App = () => {

const [isModalVisible, setModalVisible] = useState(false); // State to control modal visibility

const toggleModal = () => {

setModalVisible(!isModalVisible); // Toggle modal visibility

};

return (

<div>

<h1>Reusable Modal Example</h1>

<button onClick={toggleModal}>Open Modal</button>

{/\* Pass modal visibility, title, content, and close function as props \*/}

<Modal

isVisible={isModalVisible}

title="Sample Modal"

content={<p>This is a reusable modal component in React.</p>}

onClose={toggleModal} // Close modal when the background or close button is clicked

/>

</div>

);

};

export default App;

6.

**1. Install React Router:**

Ensure you've installed react-router-dom:

bash

CopyEdit

npm install react-router-dom

**2. Setting Up the Router and Pages:**

First, create the basic structure for your pages (e.g., Home, Contact, Profile, Login).

**HomePage.js**:

javascript

CopyEdit

import React from 'react';

const HomePage = () => {

return <h1>Home Page</h1>;

};

export default HomePage;

**ContactPage.js**:

javascript

CopyEdit

import React from 'react';

const ContactPage = () => {

return <h1>Contact Page</h1>;

};

export default ContactPage;

**ProfilePage.js**:

javascript

CopyEdit

import React from 'react';

const ProfilePage = () => {

return <h1>Profile Page (Protected)</h1>;

};

export default ProfilePage;

**LoginPage.js**:

javascript

CopyEdit

import React from 'react';

const LoginPage = () => {

return <h1>Login Page</h1>;

};

export default LoginPage;

**3. Create a Protected Route Component:**

In this component, we’ll check whether the user is authenticated. If not, we’ll redirect them to the login page.

**ProtectedRoute.js**:

javascript

CopyEdit

import React from 'react';

import { Route, Redirect } from 'react-router-dom'; // v5 syntax

// or in v6:

// import { Route, Navigate } from 'react-router-dom';

const ProtectedRoute = ({ component: Component, ...rest }) => {

const isAuthenticated = false; // Replace with actual authentication logic (e.g., check localStorage or context)

return (

<Route

{...rest}

render={(props) =>

isAuthenticated ? (

<Component {...props} />

) : (

<Redirect to="/login" /> // Redirect to login if not authenticated (v5)

)

}

/>

);

};

// In v6, this would be:

const ProtectedRouteV6 = ({ element: Element, ...rest }) => {

const isAuthenticated = false; // Replace with actual authentication logic

return (

<Route

{...rest}

element={isAuthenticated ? Element : <Navigate to="/login" />}

/>

);

};

export default ProtectedRoute;

**4. Set Up Routing in App.js:**

Now, set up routing using react-router-dom to handle navigation between the pages. You’ll include the ProtectedRoute for the Profile page.

javascript

CopyEdit

import React from 'react';

import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';

import HomePage from './HomePage';

import ContactPage from './ContactPage';

import ProfilePage from './ProfilePage';

import LoginPage from './LoginPage';

import ProtectedRoute from './ProtectedRoute';

const App = () => {

return (

<Router>

<Routes>

<Route path="/" element={<HomePage />} />

<Route path="/contact" element={<ContactPage />} />

<Route path="/login" element={<LoginPage />} />

{/\* Protected Route: ProfilePage \*/}

<Route

path="/profile"

element={<ProtectedRoute element={<ProfilePage />} />}

/>

</Routes>

</Router>

);

};

export default App;

**5. Using useNavigate (React Router v6) for Redirection:**

In React Router v6, useNavigate is used instead of useHistory for programmatic navigation. Here's how you'd implement it in the login page to redirect after login:

**LoginPage.js** (Using useNavigate):

javascript

CopyEdit

import React, { useState } from 'react';

import { useNavigate } from 'react-router-dom';

const LoginPage = () => {

const [username, setUsername] = useState('');

const [password, setPassword] = useState('');

const navigate = useNavigate();

const handleLogin = () => {

// Assuming you validate credentials here

if (username === 'user' && password === 'password') {

// Redirect to ProfilePage after successful login

navigate('/profile');

} else {

alert('Invalid credentials');

}

};

return (

<div>

<h1>Login</h1>

<input

type="text"

placeholder="Username"

value={username}

onChange={(e) => setUsername(e.target.value)}

/>

<input

type="password"

placeholder="Password"

value={password}

onChange={(e) => setPassword(e.target.value)}

/>

<button onClick={handleLogin}>Login</button>

</div>

);

};

export default LoginPage;

**6. Full Routing Flow:**

* The home, contact, and profile pages are defined as routes.
* The profile page is protected. If a user is not authenticated, they will be redirected to the login page.
* The LoginPage handles the logic for setting the user as authenticated and redirects to the profile page after successful login.

7.

**1. Set Up Product Data**

Let's assume the product data is in the following format:

javascript

CopyEdit

const products = [

{ id: 1, name: "Product A", category: "Electronics", price: 300, rating: 4.5 },

{ id: 2, name: "Product B", category: "Clothing", price: 50, rating: 3.8 },

{ id: 3, name: "Product C", category: "Electronics", price: 150, rating: 4.0 },

{ id: 4, name: "Product D", category: "Clothing", price: 80, rating: 4.2 },

{ id: 5, name: "Product E", category: "Home", price: 120, rating: 3.5 },

];

**2. Setting Up State in the Parent Component:**

You need state for the products, selected filters (category, price, rating), and sort criteria.

javascript

CopyEdit

import React, { useState } from "react";

const ProductCatalog = () => {

// Product data (could be fetched from an API)

const products = [

{ id: 1, name: "Product A", category: "Electronics", price: 300, rating: 4.5 },

{ id: 2, name: "Product B", category: "Clothing", price: 50, rating: 3.8 },

{ id: 3, name: "Product C", category: "Electronics", price: 150, rating: 4.0 },

{ id: 4, name: "Product D", category: "Clothing", price: 80, rating: 4.2 },

{ id: 5, name: "Product E", category: "Home", price: 120, rating: 3.5 },

];

// State for the filters and sort options

const [categoryFilter, setCategoryFilter] = useState("");

const [priceSort, setPriceSort] = useState(""); // "asc" or "desc"

const [ratingSort, setRatingSort] = useState(""); // "asc" or "desc"

// Filter and sort products based on selected options

const filterProducts = () => {

let filteredProducts = [...products];

// Filter by category

if (categoryFilter) {

filteredProducts = filteredProducts.filter((product) =>

product.category.toLowerCase().includes(categoryFilter.toLowerCase())

);

}

// Sort by price

if (priceSort === "asc") {

filteredProducts.sort((a, b) => a.price - b.price);

} else if (priceSort === "desc") {

filteredProducts.sort((a, b) => b.price - a.price);

}

// Sort by rating

if (ratingSort === "asc") {

filteredProducts.sort((a, b) => a.rating - b.rating);

} else if (ratingSort === "desc") {

filteredProducts.sort((a, b) => b.rating - a.rating);

}

return filteredProducts;

};

// Get filtered and sorted products

const displayedProducts = filterProducts();

return (

<div>

<h1>Product Catalog</h1>

{/\* Filter Section \*/}

<div>

<h2>Filters</h2>

<select onChange={(e) => setCategoryFilter(e.target.value)}>

<option value="">All Categories</option>

<option value="Electronics">Electronics</option>

<option value="Clothing">Clothing</option>

<option value="Home">Home</option>

</select>

</div>

{/\* Sort Section \*/}

<div>

<h2>Sort By</h2>

<div>

<label>

<input

type="radio"

name="priceSort"

value="asc"

onChange={(e) => setPriceSort(e.target.value)}

/>

Price (Low to High)

</label>

<label>

<input

type="radio"

name="priceSort"

value="desc"

onChange={(e) => setPriceSort(e.target.value)}

/>

Price (High to Low)

</label>

</div>

<div>

<label>

<input

type="radio"

name="ratingSort"

value="asc"

onChange={(e) => setRatingSort(e.target.value)}

/>

Rating (Low to High)

</label>

<label>

<input

type="radio"

name="ratingSort"

value="desc"

onChange={(e) => setRatingSort(e.target.value)}

/>

Rating (High to Low)

</label>

</div>

</div>

{/\* Product List \*/}

<div>

{displayedProducts.map((product) => (

<div key={product.id}>

<h3>{product.name}</h3>

<p>Category: {product.category}</p>

<p>Price: ${product.price}</p>

<p>Rating: {product.rating}</p>

</div>

))}

</div>

</div>

);

};

export default ProductCatalog;

8.

**1. Create a Theme Context (Optional):**

For cleaner and reusable code, you can use React's Context API to manage the theme across the app, but here I'll stick with just the useState and localStorage for simplicity.

javascript

CopyEdit

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

const App = () => {

// Check localStorage for saved theme on app load

const savedTheme = localStorage.getItem("theme") || "light"; // Default to light theme

// State to track the current theme

const [theme, setTheme] = useState(savedTheme);

// Effect to apply the theme to the document body and save it in localStorage

useEffect(() => {

document.body.className = theme; // Apply theme class to body

localStorage.setItem("theme", theme); // Save theme in localStorage

}, [theme]);

// Toggle theme function

const toggleTheme = () => {

setTheme((prevTheme) => (prevTheme === "light" ? "dark" : "light"));

};

return (

<div>

<h1>React Dark Mode Example</h1>

<button onClick={toggleTheme}>

Switch to {theme === "light" ? "Dark" : "Light"} Mode

</button>

</div>

);

};

export default App;

**2. CSS for Light and Dark Themes:**

Define the styles for both themes in your CSS file. You can use classes on the body to switch between light and dark modes.

css

CopyEdit

/\* Light theme styles \*/

body.light {

background-color: #ffffff;

color: #000000;

}

button.light {

background-color: #f0f0f0;

color: #000000;

}

/\* Dark theme styles \*/

body.dark {

background-color: #2c2c2c;

color: #ffffff;

}

button.dark {

background-color: #444444;

color: #ffffff;

}

**3. Explanation:**

* **State Initialization (useState)**: The theme state (light or dark) is initialized from localStorage using localStorage.getItem("theme"), which retrieves the theme that was previously saved. If no theme is saved, it defaults to "light".
* **Applying the Theme**: Inside the useEffect hook, when the theme state changes, the theme is applied to the document.body element by setting the appropriate class (light or dark). This class applies the correct styles defined in the CSS file.
* **Toggling the Theme**: The toggleTheme function switches between "light" and "dark" by updating the theme state. The button displays the appropriate label based on the current theme.
* **Persistence Across Sessions**: The theme is saved in localStorage every time it changes (localStorage.setItem("theme", theme)). This ensures the selected theme persists across page reloads.

**4. Additional Enhancements:**

* **Customizing the Button**: You can add an icon (e.g., sun/moon) or a toggle switch instead of a button for a more modern UI.
* **Using CSS Variables**: You could use CSS variables for colors, making it easier to modify the color scheme dynamically.

css

CopyEdit

/\* Define CSS variables \*/

:root {

--background-light: #ffffff;

--background-dark: #2c2c2c;

--text-light: #000000;

--text-dark: #ffffff;

}

/\* Light theme \*/

body.light {

background-color: var(--background-light);

color: var(--text-light);

}

/\* Dark theme \*/

body.dark {

background-color: var(--background-dark);

color: var(--text-dark);

}

9.

import React, { useState, useEffect, useCallback } from "react";

// API mock (Replace this with your actual API call)

const fetchPosts = async (page) => {

const response = await fetch(`https://jsonplaceholder.typicode.com/posts?\_page=${page}&\_limit=10`);

return response.json();

};

const InfiniteScroll = () => {

const [posts, setPosts] = useState([]);

const [page, setPage] = useState(1);

const [loading, setLoading] = useState(false);

const [hasMore, setHasMore] = useState(true);

// Load more posts when the user scrolls to the bottom

const loadMorePosts = useCallback(async () => {

if (loading || !hasMore) return; // Prevent duplicate requests

setLoading(true);

try {

const newPosts = await fetchPosts(page);

if (newPosts.length > 0) {

setPosts((prevPosts) => [...prevPosts, ...newPosts]);

setPage((prevPage) => prevPage + 1);

} else {

setHasMore(false); // No more posts

}

} catch (error) {

console.error("Error fetching posts:", error);

} finally {

setLoading(false);

}

}, [page, loading, hasMore]);

// Detect when the user reaches the bottom of the page

const handleScroll = (event) => {

const bottom = event.target.scrollHeight === event.target.scrollTop + event.target.clientHeight;

if (bottom) {

loadMorePosts();

}

};

// Setup the scroll event listener on mount

useEffect(() => {

window.addEventListener("scroll", handleScroll);

return () => {

window.removeEventListener("scroll", handleScroll);

};

}, [loadMorePosts]);

// Initial data load

useEffect(() => {

loadMorePosts();

}, [loadMorePosts]);

return (

<div>

<h1>Infinite Scroll Posts</h1>

<div>

{posts.map((post) => (

<div key={post.id}>

<h2>{post.title}</h2>

<p>{post.body}</p>

</div>

))}

</div>

{loading && <div>Loading...</div>}

{!hasMore && <div>No more posts</div>}

</div>

);

};

export default InfiniteScroll;

10.

**Approach 1: Using WebSockets for Real-Time Messaging**

**1. Setting up WebSocket Server (Node.js example)**

You can use libraries like ws in Node.js to create a WebSocket server that handles incoming connections from clients and broadcasts messages to all connected clients.

1. **Install ws (WebSocket library for Node.js)**:

bash

CopyEdit

npm install ws

1. **Create a WebSocket server**:

javascript

CopyEdit

const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', (ws) => {

console.log('User connected');

ws.on('message', (message) => {

console.log('received: %s', message);

// Broadcast the message to all connected clients

wss.clients.forEach((client) => {

if (client !== ws && client.readyState === WebSocket.OPEN) {

client.send(message);

}

});

});

ws.on('close', () => {

console.log('User disconnected');

});

});

This WebSocket server listens for messages from any connected client and broadcasts them to all other clients.

**2. React Client (WebSocket in React)**

1. **Set up WebSocket client in React**:

You can use the useEffect hook to establish a WebSocket connection when the component mounts and listen for incoming messages. The useState hook will be used to manage the chat history and the current message.

javascript

CopyEdit

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

const ChatApp = () => {

const [messages, setMessages] = useState([]);

const [message, setMessage] = useState('');

const [ws, setWs] = useState(null);

useEffect(() => {

// Establish WebSocket connection

const socket = new WebSocket('ws://localhost:8080');

// Set up WebSocket listeners

socket.onopen = () => {

console.log('Connected to WebSocket server');

};

socket.onmessage = (event) => {

setMessages((prevMessages) => [...prevMessages, event.data]);

};

socket.onclose = () => {

console.log('Disconnected from WebSocket server');

};

setWs(socket);

return () => {

if (ws) ws.close();

};

}, []);

const sendMessage = () => {

if (ws && message) {

ws.send(message);

setMessage('');

}

};

return (

<div>

<div style={{ height: '300px', overflowY: 'scroll' }}>

{messages.map((msg, index) => (

<div key={index}>{msg}</div>

))}

</div>

<input

type="text"

value={message}

onChange={(e) => setMessage(e.target.value)}

placeholder="Type a message"

/>

<button onClick={sendMessage}>Send</button>

</div>

);

};

export default ChatApp;

**Explanation:**

* **WebSocket Setup**: The useEffect hook establishes a WebSocket connection to the server when the component mounts. It listens for incoming messages and appends them to the messages state.
* **Sending Messages**: When the user types a message and presses the "Send" button, the message is sent over the WebSocket connection to the server, which then broadcasts it to other connected clients.
* **Message Display**: The messages are displayed in a scrollable div, and every time a new message is received, it's added to the chat history.

**Approach 2: Using Firebase Realtime Database or Firestore**

Firebase provides an easier-to-use solution for real-time communication by automatically syncing data across all connected clients.

**1. Setting up Firebase**

1. **Install Firebase SDK**:

bash

CopyEdit

npm install firebase

1. **Configure Firebase**:

Create a firebase.js file to initialize Firebase:

javascript

CopyEdit

import firebase from 'firebase/app';

import 'firebase/database'; // or 'firebase/firestore'

const firebaseConfig = {

apiKey: 'YOUR\_API\_KEY',

authDomain: 'YOUR\_AUTH\_DOMAIN',

databaseURL: 'YOUR\_DATABASE\_URL',

projectId: 'YOUR\_PROJECT\_ID',

storageBucket: 'YOUR\_STORAGE\_BUCKET',

messagingSenderId: 'YOUR\_MESSAGING\_SENDER\_ID',

appId: 'YOUR\_APP\_ID'

};

firebase.initializeApp(firebaseConfig);

const database = firebase.database(); // or firebase.firestore()

export default database;

**2. React Client (Firebase Realtime Database)**

1. **Set up Firebase messaging in React**:

javascript

CopyEdit

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

import database from './firebase'; // import Firebase config

const ChatApp = () => {

const [messages, setMessages] = useState([]);

const [message, setMessage] = useState('');

useEffect(() => {

const messagesRef = database.ref('messages');

// Listen for changes to the messages data

messagesRef.on('child\_added', (snapshot) => {

const msg = snapshot.val();

setMessages((prevMessages) => [...prevMessages, msg]);

});

// Cleanup listener on unmount

return () => {

messagesRef.off();

};

}, []);

const sendMessage = () => {

if (message) {

const messagesRef = database.ref('messages');

messagesRef.push({ text: message, timestamp: Date.now() });

setMessage('');

}

};

return (

<div>

<div style={{ height: '300px', overflowY: 'scroll' }}>

{messages.map((msg, index) => (

<div key={index}>{msg.text}</div>

))}

</div>

<input

type="text"

value={message}

onChange={(e) => setMessage(e.target.value)}

placeholder="Type a message"

/>

<button onClick={sendMessage}>Send</button>

</div>

);

};

export default ChatApp;

11.

**1. Install the necessary package:**

First, install react-beautiful-dnd:

bash

CopyEdit

npm install react-beautiful-dnd

**2. Basic Setup of Task Management App with Columns:**

Assume you have three columns: "To Do", "In Progress", and "Done". Each column will contain tasks, and you will allow users to drag tasks between the columns.

**3. Implementing the App Using react-beautiful-dnd:**

javascript

CopyEdit

import React, { useState } from 'react';

import { DragDropContext, Droppable, Draggable } from 'react-beautiful-dnd';

// Initial task data

const initialData = {

'to-do': [

{ id: '1', content: 'Task 1' },

{ id: '2', content: 'Task 2' },

],

'in-progress': [

{ id: '3', content: 'Task 3' },

],

'done': [

{ id: '4', content: 'Task 4' },

],

};

const App = () => {

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

// Handle the drag end event

const onDragEnd = (result) => {

const { destination, source, draggableId } = result;

// If dropped outside the droppable area

if (!destination) {

return;

}

// If the item is dropped at the same location (no change)

if (

destination.droppableId === source.droppableId &&

destination.index === source.index

) {

return;

}

// Reorder the tasks within the same column

const startColumn = tasks[source.droppableId];

const endColumn = tasks[destination.droppableId];

// If task is dropped into the same column

if (startColumn === endColumn) {

const newColumn = Array.from(startColumn);

newColumn.splice(source.index, 1);

newColumn.splice(destination.index, 0, startColumn[source.index]);

setTasks({

...tasks,

[source.droppableId]: newColumn,

});

} else {

// Move task from one column to another

const startColumnCopy = Array.from(startColumn);

const [removed] = startColumnCopy.splice(source.index, 1);

const endColumnCopy = Array.from(endColumn);

endColumnCopy.splice(destination.index, 0, removed);

setTasks({

...tasks,

[source.droppableId]: startColumnCopy,

[destination.droppableId]: endColumnCopy,

});

}

};

return (

<DragDropContext onDragEnd={onDragEnd}>

<div style={{ display: 'flex', justifyContent: 'space-around' }}>

{['to-do', 'in-progress', 'done'].map((columnId) => (

<Droppable droppableId={columnId} key={columnId}>

{(provided) => (

<div

ref={provided.innerRef}

{...provided.droppableProps}

style={{

padding: '10px',

width: '200px',

minHeight: '400px',

backgroundColor: '#f4f4f4',

borderRadius: '5px',

}}

>

<h3>{columnId.replace('-', ' ').toUpperCase()}</h3>

{tasks[columnId].map((task, index) => (

<Draggable draggableId={task.id} index={index} key={task.id}>

{(provided) => (

<div

ref={provided.innerRef}

{...provided.draggableProps}

{...provided.dragHandleProps}

style={{

...provided.draggableProps.style,

marginBottom: '10px',

padding: '10px',

backgroundColor: '#fff',

border: '1px solid #ccc',

borderRadius: '5px',

}}

>

{task.content}

</div>

)}

</Draggable>

))}

{provided.placeholder}

</div>

)}

</Droppable>

))}

</div>

</DragDropContext>

);

};

export default App;

12.

**1. Install Redux and React-Redux:**

First, install the necessary dependencies:

bash

CopyEdit

npm install redux react-redux

* redux is the state management library.
* react-redux connects Redux with React components.

**2. Define Actions:**

Actions represent the events that occur in the app. In this case, we'll need actions to add an item to the cart, remove an item from the cart, and update the quantity of an item in the cart.

javascript

CopyEdit

// src/actions/cartActions.js

export const ADD\_ITEM = 'ADD\_ITEM';

export const REMOVE\_ITEM = 'REMOVE\_ITEM';

export const UPDATE\_ITEM\_QUANTITY = 'UPDATE\_ITEM\_QUANTITY';

// Action to add an item to the cart

export const addItem = (item) => ({

type: ADD\_ITEM,

payload: item,

});

// Action to remove an item from the cart

export const removeItem = (id) => ({

type: REMOVE\_ITEM,

payload: id,

});

// Action to update the quantity of an item

export const updateItemQuantity = (id, quantity) => ({

type: UPDATE\_ITEM\_QUANTITY,

payload: { id, quantity },

});

**3. Create Reducers:**

Reducers are functions that specify how the state changes in response to actions. In this case, we need a cartReducer to manage the shopping cart state.

javascript

CopyEdit

// src/reducers/cartReducer.js

import { ADD\_ITEM, REMOVE\_ITEM, UPDATE\_ITEM\_QUANTITY } from '../actions/cartActions';

const initialState = {

items: [],

};

const cartReducer = (state = initialState, action) => {

switch (action.type) {

case ADD\_ITEM:

return {

...state,

items: [...state.items, action.payload],

};

case REMOVE\_ITEM:

return {

...state,

items: state.items.filter((item) => item.id !== action.payload),

};

case UPDATE\_ITEM\_QUANTITY:

return {

...state,

items: state.items.map((item) =>

item.id === action.payload.id

? { ...item, quantity: action.payload.quantity }

: item

),

};

default:

return state;

}

};

export default cartReducer;

**4. Combine Reducers:**

If your app has multiple reducers, combine them using combineReducers. In this case, we have only one reducer (cartReducer), but here’s how you would combine it if you had more:

javascript

CopyEdit

// src/reducers/index.js

import { combineReducers } from 'redux';

import cartReducer from './cartReducer';

const rootReducer = combineReducers({

cart: cartReducer,

});

export default rootReducer;

**5. Configure the Store:**

Next, configure the Redux store by passing the rootReducer to createStore:

javascript

CopyEdit

// src/store.js

import { createStore } from 'redux';

import rootReducer from './reducers';

const store = createStore(

rootReducer,

window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_() // Enables Redux DevTools extension

);

export default store;

**6. Provide the Store to the Application:**

To make the Redux store available throughout the app, wrap your app in the Provider component from react-redux and pass it the store:

javascript

CopyEdit

// src/index.js

import React from 'react';

import ReactDOM from 'react-dom';

import { Provider } from 'react-redux';

import store from './store';

import App from './App';

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

);

**7. Connect Components to Redux Store:**

Now, we’ll connect the React components to Redux. We'll use the useSelector hook to access the state and the useDispatch hook to dispatch actions.

Here’s how we can implement a shopping cart component where users can add, remove, or update the quantity of items in the cart:

javascript

CopyEdit

// src/components/Cart.js

import React from 'react';

import { useSelector, useDispatch } from 'react-redux';

import { addItem, removeItem, updateItemQuantity } from '../actions/cartActions';

const Cart = () => {

const dispatch = useDispatch();

const cartItems = useSelector((state) => state.cart.items);

const handleAddItem = () => {

const newItem = {

id: Math.random().toString(36).substr(2, 9), // Generate a unique ID

name: 'Product Name',

price: 100,

quantity: 1,

};

dispatch(addItem(newItem));

};

const handleRemoveItem = (id) => {

dispatch(removeItem(id));

};

const handleUpdateQuantity = (id, quantity) => {

dispatch(updateItemQuantity(id, quantity));

};

return (

<div>

<h2>Shopping Cart</h2>

<button onClick={handleAddItem}>Add Item</button>

<ul>

{cartItems.map((item) => (

<li key={item.id}>

<div>{item.name}</div>

<div>{item.price}</div>

<div>

Quantity:

<button onClick={() => handleUpdateQuantity(item.id, item.quantity - 1)}>-</button>

{item.quantity}

<button onClick={() => handleUpdateQuantity(item.id, item.quantity + 1)}>+</button>

</div>

<button onClick={() => handleRemoveItem(item.id)}>Remove</button>

</li>

))}

</ul>

</div>

);

};

export default Cart;

**8. Implementing the Cart in the App:**

In your App.js, you can now use the Cart component.

javascript

CopyEdit

// src/App.js

import React from 'react';

import Cart from './components/Cart';

const App = () => {

return (

<div>

<h1>Redux Shopping Cart</h1>

<Cart />

</div>

);

};

export default App;

13.

**1. Create the Layout Component:**

Let’s create a simple React component with a Flexbox container and child items (columns).

javascript

CopyEdit

// src/components/ResponsiveLayout.js

import React from 'react';

import './ResponsiveLayout.css'; // CSS for Flexbox styles

const ResponsiveLayout = () => {

return (

<div className="flex-container">

<div className="flex-item">Item 1</div>

<div className="flex-item">Item 2</div>

<div className="flex-item">Item 3</div>

<div className="flex-item">Item 4</div>

<div className="flex-item">Item 5</div>

<div className="flex-item">Item 6</div>

</div>

);

};

export default ResponsiveLayout;

**2. Define Flexbox Styles:**

In the ResponsiveLayout.css file, we define the Flexbox container and the responsive styles using media queries to adjust the number of columns based on the screen size.

css

CopyEdit

/\* src/components/ResponsiveLayout.css \*/

.flex-container {

display: flex;

flex-wrap: wrap; /\* Allows items to wrap to the next line \*/

gap: 16px; /\* Space between columns \*/

}

.flex-item {

background-color: #f2f2f2;

padding: 20px;

flex: 1; /\* Flex items will stretch to fill the container \*/

text-align: center;

border: 1px solid #ccc;

}

/\* Responsive Layout \*/

/\* For small screens (e.g., mobile), display 1 column \*/

@media (max-width: 600px) {

.flex-item {

flex: 1 1 100%; /\* 100% width for mobile \*/

}

}

/\* For medium screens (e.g., tablets), display 2 columns \*/

@media (min-width: 601px) and (max-width: 1024px) {

.flex-item {

flex: 1 1 calc(50% - 16px); /\* 50% width minus gap \*/

}

}

/\* For large screens (e.g., desktops), display 3 columns \*/

@media (min-width: 1025px) {

.flex-item {

flex: 1 1 calc(33.33% - 16px); /\* 33.33% width minus gap \*/

}

}

**3. Explanation of the Flexbox CSS:**

1. **.flex-container**:
   * display: flex; enables Flexbox on the container.
   * flex-wrap: wrap; ensures that items can wrap onto the next line if they exceed the available width of the container.
   * gap: 16px; adds space between each item.
2. **.flex-item**:
   * flex: 1; allows each item to grow and fill the container space evenly.
   * background-color, padding, and border are just for styling purposes.
   * text-align: center; centers the text inside each item.
3. **Media Queries**:
   * For small screens (e.g., mobile devices with a width of 600px or less), the flex items will take up 100% of the container width (flex: 1 1 100%), so only one column will be displayed.
   * For medium screens (e.g., tablets with a width between 601px and 1024px), the items will take up 50% of the container width (flex: 1 1 calc(50% - 16px)), so two columns will be displayed.
   * For large screens (e.g., desktops with a width of 1025px or more), the items will take up 33.33% of the container width (flex: 1 1 calc(33.33% - 16px)), so three columns will be displayed.

**4. Use the Component in Your App:**

Now, use the ResponsiveLayout component in your App.js file:

javascript

CopyEdit

// src/App.js

import React from 'react';

import ResponsiveLayout from './components/ResponsiveLayout';

const App = () => {

return (

<div>

<h1>Responsive Layout with Flexbox</h1>

<ResponsiveLayout />

</div>

);

};

export default App;

14.

import React, { useState } from 'react';

const FormWithValidation = () => {

// Step 1: Initialize state for the form fields and errors

const [formData, setFormData] = useState({

username: '',

email: '',

password: ''

});

const [errors, setErrors] = useState({

username: '',

email: '',

password: ''

});

// Step 2: Handle input changes and update state

const handleChange = (e) => {

const { name, value } = e.target;

setFormData((prevState) => ({

...prevState,

[name]: value

}));

};

// Step 3: Validation function

const validate = () => {

const newErrors = { username: '', email: '', password: '' };

let isValid = true;

// Username validation: must not be empty

if (!formData.username) {

newErrors.username = 'Username is required';

isValid = false;

}

// Email validation: must be a valid email

const emailRegex = /^[a-zA-Z0-9.\_-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;

if (!formData.email) {

newErrors.email = 'Email is required';

isValid = false;

} else if (!emailRegex.test(formData.email)) {

newErrors.email = 'Please enter a valid email';

isValid = false;

}

// Password validation: must have at least 6 characters

if (!formData.password) {

newErrors.password = 'Password is required';

isValid = false;

} else if (formData.password.length < 6) {

newErrors.password = 'Password must be at least 6 characters';

isValid = false;

}

setErrors(newErrors);

return isValid;

};

// Step 4: Handle form submission

const handleSubmit = (e) => {

e.preventDefault();

if (validate()) {

alert('Form submitted successfully');

// Perform submission logic here (e.g., send data to an API)

} else {

alert('Please fix the errors in the form');

}

};

return (

<form onSubmit={handleSubmit}>

<div>

<label htmlFor="username">Username:</label>

<input

type="text"

id="username"

name="username"

value={formData.username}

onChange={handleChange}

/>

{errors.username && <p className="error">{errors.username}</p>}

</div>

<div>

<label htmlFor="email">Email:</label>

<input

type="email"

id="email"

name="email"

value={formData.email}

onChange={handleChange}

/>

{errors.email && <p className="error">{errors.email}</p>}

</div>

<div>

<label htmlFor="password">Password:</label>

<input

type="password"

id="password"

name="password"

value={formData.password}

onChange={handleChange}

/>

{errors.password && <p className="error">{errors.password}</p>}

</div>

<button type="submit">Submit</button>

</form>

);

};

export default FormWithValidation;

Explanation:

State Management (useState):

formData stores the values for the username, email, and password fields.

errors stores any validation error messages for each field.

handleChange Function:

This function updates the formData state whenever the user types in the input fields. The input name (name attribute) is used to ensure that the correct state property is updated.

Validation (validate Function):

The validate function checks if the inputs meet the criteria:

Username: Cannot be empty.

Email: Must be a valid email.

Password: Must be at least 6 characters long.

If any of these checks fail, an error message is stored in the errors state.

Form Submission (handleSubmit):

When the form is submitted, the validate function is called. If validation passes, an alert is shown (you could replace this with actual form submission logic like sending data to an API). If validation fails, an alert prompts the user to fix the errors.

Error Display:

If any field has an error, the error message will be displayed below the corresponding field.

Styling the Error Messages:

You can style the error messages by adding some basic CSS, for example:

css

Copy

Edit

/\* src/App.css \*/

.error {

color: red;

font-size: 12px;

}

15.

**1. Create the Theme Context**

First, create a context that will hold the theme state and the function to toggle it.

javascript

CopyEdit

// src/contexts/ThemeContext.js

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

// Create a context

const ThemeContext = createContext();

// Create a provider component

const ThemeProvider = ({ children }) => {

// Set the initial theme based on localStorage or default to light

const [theme, setTheme] = useState(localStorage.getItem('theme') || 'light');

// Function to toggle theme

const toggleTheme = () => {

const newTheme = theme === 'light' ? 'dark' : 'light';

setTheme(newTheme);

localStorage.setItem('theme', newTheme); // Save the theme to localStorage

};

return (

<ThemeContext.Provider value={{ theme, toggleTheme }}>

{children}

</ThemeContext.Provider>

);

};

export { ThemeProvider, ThemeContext };

**2. Wrap Your Application with the ThemeProvider**

In your App.js or entry point of your application, wrap your components with the ThemeProvider so that all components within it can access the theme state.

javascript

CopyEdit

// src/App.js

import React from 'react';

import { ThemeProvider } from './contexts/ThemeContext';

import ThemeToggleButton from './components/ThemeToggleButton';

import './App.css';

function App() {

return (

<ThemeProvider>

<div className="App">

<ThemeToggleButton />

{/\* Other components go here \*/}

</div>

</ThemeProvider>

);

}

export default App;

**3. Create a Component to Toggle the Theme**

Now create a component (ThemeToggleButton) that will allow the user to toggle between light and dark modes.

javascript

CopyEdit

// src/components/ThemeToggleButton.js

import React, { useContext } from 'react';

import { ThemeContext } from '../contexts/ThemeContext';

const ThemeToggleButton = () => {

const { theme, toggleTheme } = useContext(ThemeContext);

return (

<button onClick={toggleTheme}>

Switch to {theme === 'light' ? 'dark' : 'light'} mode

</button>

);

};

export default ThemeToggleButton;

**4. Apply the Theme Globally**

You can apply the selected theme globally by modifying the body class or any other wrapper element in your app.

For example, in your App.css or a similar global stylesheet:

css

CopyEdit

/\* src/App.css \*/

body {

transition: background-color 0.3s, color 0.3s;

}

body.light {

background-color: #fff;

color: #000;

}

body.dark {

background-color: #333;

color: #fff;

}

Then, modify your App.js or ThemeProvider to dynamically update the class on the body tag based on the current theme:

javascript

CopyEdit

// src/App.js

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

import { ThemeProvider, ThemeContext } from './contexts/ThemeContext';

import ThemeToggleButton from './components/ThemeToggleButton';

import './App.css';

function App() {

const { theme } = useContext(ThemeContext);

useEffect(() => {

// Apply the theme class to the body element

document.body.className = theme;

}, [theme]);

return (

<div className="App">

<ThemeToggleButton />

{/\* Other components \*/}

</div>

);

}

export default App;

**5. (Optional) Persisting the Theme**

In the example above, the selected theme is persisted in localStorage so that when the user refreshes the page, the theme remains the same. The useEffect hook updates the body class whenever the theme changes.

16.

import React, { useState } from 'react';

const FileUpload = () => {

const [image, setImage] = useState(null); // State to store the selected image

const [imagePreview, setImagePreview] = useState(null); // State to store the image preview URL

// Handle file input change

const handleFileChange = (event) => {

const file = event.target.files[0];

if (file) {

// Set the selected image in state

setImage(file);

// Create an image preview using URL.createObjectURL

setImagePreview(URL.createObjectURL(file));

}

};

// Remove the selected image

const removeImage = () => {

setImage(null);

setImagePreview(null);

};

// Handle the image upload

const handleUpload = () => {

// Logic for uploading the image (e.g., using FormData, Axios, etc.)

console.log('Uploading image: ', image);

};

return (

<div className="file-upload-container">

<input

type="file"

accept="image/\*"

onChange={handleFileChange}

/>

{/\* If an image preview exists, display it \*/}

{imagePreview && (

<div className="image-preview">

<img src={imagePreview} alt="Preview" style={{ width: '200px', height: '200px' }} />

<button onClick={removeImage}>Remove Image</button>

</div>

)}

{/\* Show upload button if an image is selected \*/}

{image && (

<div>

<button onClick={handleUpload}>Upload Image</button>

</div>

)}

</div>

);

};

export default FileUpload;

Explanation of the Code

State Management:

image: This state holds the selected image file.

imagePreview: This state holds the URL for the image preview, generated by URL.createObjectURL(). This will allow us to display a preview of the selected image.

File Input:

The file input field is set to accept only image files (accept="image/\*").

The handleFileChange function is triggered when the user selects a file. It updates the image state with the selected file and the imagePreview state with a preview URL.

Image Preview:

If an image is selected (imagePreview is not null), an image preview is displayed using the img tag. The image is displayed with a width and height of 200px, but this can be adjusted as needed.

Remove Image:

The removeImage function resets both the image and imagePreview states to null, effectively removing the image preview and resetting the input.

Upload Button:

If an image is selected, the "Upload Image" button is shown. When clicked, it calls the handleUpload function, where you can add the logic for uploading the image to a server or performing any other necessary actions (e.g., using FormData and an HTTP client like Axios).

Styling the Component (Optional)

You can add basic CSS for styling the upload container, image preview, and buttons:

css

Copy

Edit

/\* Example styles for the file upload component \*/

.file-upload-container {

text-align: center;

margin: 20px;

}

.image-preview {

margin-top: 10px;

display: inline-block;

text-align: center;

}

.image-preview img {

margin-bottom: 10px;

}

button {

padding: 8px 16px;

margin: 5px;

border: none;

cursor: pointer;

background-color: #4CAF50;

color: white;

}

button:hover {

background-color: #45a049;

}

Handling the Upload

If you need to upload the image to a server, here's a quick example of how you can implement the upload logic using FormData and fetch (or Axios):

javascript

Copy

Edit

const handleUpload = () => {

const formData = new FormData();

formData.append('file', image); // Append the selected file to the FormData

// Example of uploading the image using fetch

fetch('YOUR\_UPLOAD\_URL', {

method: 'POST',

body: formData,

})

.then(response => response.json())

.then(data => console.log('Image uploaded successfully:', data))

.catch(error => console.error('Error uploading image:', error));

};

17.

import React, { useState } from 'react';

const PaginationExample = () => {

// Sample data (could be fetched from an API)

const data = Array.from({ length: 100 }, (\_, index) => `Item ${index + 1}`);

const itemsPerPage = 10; // Number of items per page

const [currentPage, setCurrentPage] = useState(1); // Current page state

// Calculate total number of pages

const totalPages = Math.ceil(data.length / itemsPerPage);

// Get the subset of data to display on the current page

const currentData = data.slice(

(currentPage - 1) \* itemsPerPage,

currentPage \* itemsPerPage

);

// Handle page change

const handlePageChange = (pageNumber) => {

if (pageNumber > 0 && pageNumber <= totalPages) {

setCurrentPage(pageNumber);

}

};

return (

<div>

<h1>Paginated Data</h1>

{/\* Display current page data \*/}

<ul>

{currentData.map((item, index) => (

<li key={index}>{item}</li>

))}

</ul>

{/\* Pagination Controls \*/}

<div className="pagination-controls">

<button

onClick={() => handlePageChange(1)}

disabled={currentPage === 1}

>

First

</button>

<button

onClick={() => handlePageChange(currentPage - 1)}

disabled={currentPage === 1}

>

Previous

</button>

<span>Page {currentPage} of {totalPages}</span>

<button

onClick={() => handlePageChange(currentPage + 1)}

disabled={currentPage === totalPages}

>

Next

</button>

<button

onClick={() => handlePageChange(totalPages)}

disabled={currentPage === totalPages}

>

Last

</button>

</div>

</div>

);

};

export default PaginationExample;

Explanation of the Code:

Data Setup:

We create a sample dataset (data) with 100 items. In a real app, this data would likely come from an API.

State Management:

currentPage: Keeps track of the current page.

itemsPerPage: Defines how many items to show on each page.

Pagination Logic:

totalPages is calculated by dividing the total data length by itemsPerPage and rounding up using Math.ceil().

currentData is the subset of data that corresponds to the current page. This is done by slicing the data array using slice(), based on the currentPage and itemsPerPage.

Page Navigation:

The handlePageChange function updates the currentPage state when the user clicks a page navigation button.

The "First", "Previous", "Next", and "Last" buttons allow the user to navigate through the pages. These buttons are disabled if the user is already on the first or last page.

Display:

The data corresponding to the current page is displayed in a list (<ul>).

The current page number and total pages are displayed in the pagination controls.

Styling (Optional):

You can style the pagination buttons to make them look nicer. For example:

css

Copy

Edit

.pagination-controls {

margin-top: 20px;

display: flex;

justify-content: center;

gap: 10px;

}

.pagination-controls button {

padding: 10px;

background-color: #4CAF50;

color: white;

border: none;

cursor: pointer;

}

.pagination-controls button:disabled {

background-color: #ddd;

cursor: not-allowed;

}

.pagination-controls span {

align-self: center;

}

18.

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

const CountdownTimer = ({ targetDate }) => {

// Set the initial state to the difference between the target date and the current date

const calculateTimeLeft = () => {

const difference = targetDate - new Date();

return {

seconds: Math.floor((difference / 1000) % 60),

minutes: Math.floor((difference / 1000 / 60) % 60),

hours: Math.floor((difference / (1000 \* 60 \* 60)) % 24),

days: Math.floor(difference / (1000 \* 60 \* 60 \* 24)),

};

};

const [timeLeft, setTimeLeft] = useState(calculateTimeLeft());

useEffect(() => {

// If the countdown reaches zero, stop updating the time

if (timeLeft.days <= 0 && timeLeft.hours <= 0 && timeLeft.minutes <= 0 && timeLeft.seconds <= 0) {

return;

}

const interval = setInterval(() => {

setTimeLeft(calculateTimeLeft());

}, 1000);

// Cleanup the interval when the component unmounts or when the countdown ends

return () => clearInterval(interval);

}, [timeLeft]);

// If time is up, display a message

if (timeLeft.days <= 0 && timeLeft.hours <= 0 && timeLeft.minutes <= 0 && timeLeft.seconds <= 0) {

return <div>Time's up!</div>;

}

return (

<div>

<h2>Countdown Timer</h2>

<div>

{timeLeft.days}d {timeLeft.hours}h {timeLeft.minutes}m {timeLeft.seconds}s

</div>

</div>

);

};

export default CountdownTimer;

Explanation of the Code:

calculateTimeLeft():

This function calculates the remaining time by subtracting the current time from the targetDate.

It returns an object with days, hours, minutes, and seconds.

useState():

We initialize the state (timeLeft) with the result of calculateTimeLeft(). The timeLeft state will hold the countdown values.

useEffect():

We use setInterval to update the timer every second by calling setTimeLeft(calculateTimeLeft()).

When the countdown reaches zero, the setInterval will stop updating, and the message "Time's up!" will appear.

We clear the interval when the component unmounts or the countdown finishes using the cleanup function returned by useEffect.

Timer Display:

The timer is displayed as a combination of days, hours, minutes, and seconds.

When the countdown ends, the message "Time's up!" will be displayed.

Usage Example:

You can pass the targetDate prop (which is the date and time when the event happens) to the CountdownTimer component.

javascript

Copy

Edit

const App = () => {

// Set the target date for the event (e.g., New Year's Eve)

const targetDate = new Date('2025-01-01T00:00:00');

return (

<div>

<h1>Event Countdown</h1>

<CountdownTimer targetDate={targetDate} />

</div>

);

};

export default App;

Styling (Optional):

You can style the timer to make it visually appealing.

css

Copy

Edit

div {

font-size: 24px;

font-weight: bold;

color: #333;

}

h1 {

text-align: center;

}

h2 {

text-align: center;

margin-top: 20px;

}

19.

import React, { useState } from 'react';

// Post Component

const Post = ({ post, onLike, onComment, onShare }) => {

return (

<div className="post">

<h3>{post.user}</h3>

<p>{post.content}</p>

<div className="interactions">

<button onClick={() => onLike(post.id)}>Like ({post.likes})</button>

<button onClick={() => onComment(post.id)}>Comment ({post.comments.length})</button>

<button onClick={() => onShare(post.id)}>Share ({post.shares})</button>

</div>

<div className="comments">

{post.comments.map((comment, index) => (

<p key={index}>{comment}</p>

))}

</div>

</div>

);

};

// News Feed Component

const NewsFeed = () => {

const [posts, setPosts] = useState([

{

id: 1,

user: 'John Doe',

content: 'This is a sample post.',

likes: 0,

comments: [],

shares: 0,

},

{

id: 2,

user: 'Jane Smith',

content: 'Here is another post.',

likes: 0,

comments: [],

shares: 0,

},

// More posts...

]);

// Like handler

const handleLike = (id) => {

setPosts(posts.map(post =>

post.id === id

? { ...post, likes: post.likes + 1 }

: post

));

};

// Comment handler

const handleComment = (id) => {

const comment = prompt('Enter your comment:');

if (comment) {

setPosts(posts.map(post =>

post.id === id

? { ...post, comments: [...post.comments, comment] }

: post

));

}

};

// Share handler

const handleShare = (id) => {

setPosts(posts.map(post =>

post.id === id

? { ...post, shares: post.shares + 1 }

: post

));

};

return (

<div className="news-feed">

{posts.map(post => (

<Post

key={post.id}

post={post}

onLike={handleLike}

onComment={handleComment}

onShare={handleShare}

/>

))}

</div>

);

};

export default NewsFeed;

Explanation of the Code:

Post Component:

The Post component takes a post object and three handlers (onLike, onComment, onShare) as props.

It displays the content of the post and the number of likes, comments, and shares.

The buttons trigger the corresponding interaction when clicked.

The comments are displayed under the post.

NewsFeed Component:

The NewsFeed component manages the state of the posts. The state is initialized with an array of posts, each having properties for likes, comments, and shares.

The handleLike, handleComment, and handleShare functions are responsible for updating the state when the user interacts with the post.

handleComment uses prompt() to get a comment from the user, but in a real-world scenario, you would use a more robust solution (e.g., a modal or input field).

The posts are mapped and displayed using the Post component.

Interaction Handlers:

Like: Each time a user clicks the "Like" button, the like count for the corresponding post increases by 1.

Comment: A prompt asks the user to enter a comment, and the comment is added to the post’s comments array.

Share: Each time a user clicks the "Share" button, the share count for the corresponding post increases by 1.

Styling (Optional):

You can add some basic styling to make the feed look more appealing.

css

Copy

Edit

.news-feed {

width: 80%;

margin: 0 auto;

}

.post {

border: 1px solid #ddd;

padding: 20px;

margin-bottom: 20px;

}

button {

margin-right: 10px;

}

.comments {

margin-top: 10px;

font-style: italic;

color: #555;

}

20.

**1. Virtualized Lists**

Virtualization is the most effective way to optimize large lists by rendering only the visible items (or a small set of items) at any given time. As the user scrolls, new items are rendered dynamically while off-screen items are removed from the DOM.

**Libraries**:

* **react-window**: A lightweight solution for rendering large lists by rendering only the visible items.
* **react-virtualized**: A more feature-rich library for virtualized lists, grids, and tables.

**Example with react-window:**

bash

CopyEdit

npm install react-window

javascript

CopyEdit

import React from 'react';

import { FixedSizeList as List } from 'react-window';

const MyList = () => {

const items = new Array(1000).fill('Item');

const Row = ({ index, style }) => (

<div style={style}>

{items[index]} #{index}

</div>

);

return (

<List

height={400} // Height of the container

itemCount={items.length} // Total number of items

itemSize={35} // Height of each item

width={300} // Width of the container

>

{Row}

</List>

);

};

export default MyList;

In this example, only the visible items are rendered, which improves performance significantly.

**2. Memoization with React.memo**

When rendering large lists, if individual list items don't change frequently, you can wrap those components with React.memo to prevent unnecessary re-renders. React.memo is a higher-order component that will only re-render if the props of a component change.

**Example:**

javascript

CopyEdit

const ListItem = React.memo(({ item }) => {

return <div>{item}</div>;

});

const List = ({ items }) => {

return (

<div>

{items.map((item, index) => (

<ListItem key={index} item={item} />

))}

</div>

);

};

In this example, ListItem will only re-render if the item prop changes, thus avoiding unnecessary renders of list items that have not changed.

**3. Use of useMemo for Expensive Computations**

If you are performing heavy computations or transformations on the list data, you can use the useMemo hook to memoize the result and avoid recalculating the list items on every render.

**Example:**

javascript

CopyEdit

const items = useMemo(() => processItems(rawItems), [rawItems]);

This ensures that processItems is only recalculated when rawItems changes.

**4. Windowing and Pagination**

Instead of rendering the entire list at once, consider implementing **pagination** or **windowing**. This reduces the number of items being rendered and improves performance by only showing a subset of the data.

* **Pagination**: Load and display a subset of items, and load more data as the user navigates between pages.
* **Windowing**: Only render a small portion of the list and update it as the user scrolls.

**5. Lazy Loading of Data**

If your list data is coming from a backend, consider **lazy loading** the data. This means loading the list in chunks instead of fetching all items at once. You can combine this with infinite scrolling or pagination.

**Example:**

javascript

CopyEdit

const fetchItems = async (page) => {

const response = await fetch(`https://api.example.com/items?page=${page}`);

const data = await response.json();

return data.items;

};

**6. Avoiding Inline Functions in render**

Inline functions within render can lead to unnecessary re-renders, as they create new instances of functions on each render. Instead, define event handlers outside of the render method or use useCallback to memoize event handlers.

**Example**:

javascript

CopyEdit

const handleClick = useCallback(() => {

// Handle the click event

}, []);

return <button onClick={handleClick}>Click me</button>;

**7. Batching State Updates**

React batches multiple state updates together to minimize re-renders. However, if you're making multiple state updates within the same event handler, try to batch them using functional updates.

**Example:**

javascript

CopyEdit

setItems((prevItems) => [...prevItems, newItem]);

This will ensure that React does not re-render after each state update but only once after all the updates are processed.

**8. Avoid Unnecessary Re-renders with shouldComponentUpdate or PureComponent**

If you're using class components, shouldComponentUpdate can help you control when the component should update. For functional components, React.memo serves a similar purpose.

**Example with shouldComponentUpdate:**

javascript

CopyEdit

class ListItem extends React.Component {

shouldComponentUpdate(nextProps) {

return nextProps.item !== this.props.item;

}

render() {

return <div>{this.props.item}</div>;

}

}

This ensures that the ListItem component will only re-render if the item prop changes.

**9. Using Web Workers for Heavy Computation**

If your list rendering involves heavy data computation (e.g., sorting, filtering, or transforming large datasets), you can offload the heavy computation to a **Web Worker**. Web Workers run in a separate thread and won't block the main thread, allowing the UI to remain responsive.

**10. Code Splitting and Lazy Loading Components**

For large lists with complex items, it may help to **lazy load** or **code split** the components, especially if they are large and not immediately necessary.

**Example**:

javascript

CopyEdit

const ListItem = React.lazy(() => import('./ListItem'));