* Component:

Header

Root component (Main component)

Index.html

footer

Note: In react root component is known as or named as app component.

Components are like functions that return HTML elements.

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML.

* Components come in two types:

1. Class components

In React, a class component is a type of component defined using JavaScript's class syntax. Class components extend from React.Component and require a render() method, which returns JSX to render the component's UI. This type of component is mainly used when you need to manage state or use lifecycle methods, although function components with hooks are now more commonly used for similar purposes.

Ex:

File 🡪Welcome.js

Import React, {Component} from “react”;

class Welcome extends Component

{

Render()

{

return <h1>something</h1>

}

}

Export default Welcome;

// import this where you want to use

1. Function Components

A function component in React is a simpler way to create components using JavaScript functions rather than classes. Function components are easy to read and write, and they work well with React hooks, allowing you to manage state and lifecycle events in a functional style.

EX: function App()

{

return (

<div>

<h1>Ankita</h1>

<p>Hello</p>

</div>

)

}

export default App;

explanation:

create a function with same js file name

// return <h1>something</h1> for single element

in this function write return keyword which will return only one statement for more statements we will use div inside return with braces

to access or import this function at last we will use export default with function name.

* Component inside root component

Header.js

Function Header()

{

Return <h1> Hello AP</h1>

// here also you can add component

}

export default Header;

App.js

Import Header from “./component/Header.js” 🡪path

Function App()

{

return(

<div>

<Header></Header>

</div>

)

}

There is another way to export component

One is with default and another one is

Export const header= function Header()

{ rest of the code}

* Styling React using CSS

There are three methods

1. **Inline Styling**

In React, inline styling is a way to apply styles directly to an element within the component's JSX. This is similar to HTML inline styles but follows JavaScript syntax. Inline styles in React are defined as JavaScript objects and applied using the style attribute.

**Syntax**:

* Inline styles are written as objects, with each CSS property written in camelCase instead of the usual kebab-case.
* You use the style attribute and pass an object with style properties.

**Defining Styles**:

* Each CSS property becomes a key in the object, and the value is a string or a variable.
* Ex

<div style={{ color: 'blue', fontSize: '20px' }}>Hello, World!</div>

**Dynamic Styling**:

* Inline styles allow you to use JavaScript expressions to set values dynamically.
* Ex:

function MyComponent() {

const isPrimary = true;

return (

<button style={{ backgroundColor: isPrimary ? 'blue' : 'gray' }}>

Click Me

</button>

);

}

**Complex Styles**:

* You can also define the style object outside of the JSX and reference it for cleaner

Ex

const buttonStyle = {

backgroundColor: 'blue',

padding: '10px',

color: 'white'

};

function MyComponent() {

return <button style={buttonStyle}>Click Me</button>;

}

**Using Variables in Styles**:

Since inline styles are JavaScript objects, you can use variables for values

Ex

const fontSize = '20px';

<div style={{ color: 'red', fontSize }}>Hello, World!</div>

1. **Css Stylesheet**

In React, using an external CSS stylesheet is a common and efficient way to manage styles across components.

**1. Creating a CSS Stylesheet**

* First, create a CSS file (e.g., App.css or MyComponent.css) with your styles.
* Define classes and styles just like in a regular CSS file. For example:

css

/\* MyComponent.css \*/

.container {

background-color: #f0f0f0;

padding: 20px;

border-radius: 5px;

}

.text {

color: blue;

font-size: 18px;

}

**2. Importing the Stylesheet in a Component**

* To use the stylesheet in a React component, import the CSS file at the top of your component file.

jsx

import React from 'react';

import './MyComponent.css'; // Importing the CSS file

function MyComponent() {

return (

<div className="container">

<p className="text">Hello, this is a styled component!</p>

</div>

);

}

export default MyComponent;

**3. Applying Classes with the className Attribute**

* In React, you use the className attribute (instead of class in HTML) to apply CSS classes to elements.
* The class names in JSX should match the names in your CSS file exactly.
* Here’s how it works in the example above:

jsx

<div className="container">

<p className="text">Hello, this is a styled component!</p>

</div>

**4. Global vs. Component-Level Stylesheets**

* You can have a global stylesheet (like App.css) that is imported in the main App.js component and affects the entire application.
* For specific component styling, you can create separate CSS files for each component (e.g., MyComponent.css). This keeps the styles modular and easier to maintain.

**5. CSS Modules for Scoped Styles (Optional)**

* To avoid class name collisions, you can use CSS Modules, which allow scoping styles to a specific component.
* Create a CSS Module file with the extension .module.css (e.g., MyComponent.module.css).
* Import it like this:

jsx

import styles from './MyComponent.module.css';

function MyComponent() {

return (

<div className={styles.container}>

<p className={styles.text}>Hello, this is a styled component with CSS Modules!</p>

</div>

);

}

* In this case, class names are accessed through the styles object and are locally scoped to the component, preventing name clashes.

1. **CSS Module**

CSS Modules in React offer a way to write CSS that is scoped locally to a specific component, helping to avoid class name conflicts and making styles easier to manage in larger applications.

**1. Creating a CSS Module File**

* To create a CSS Module, name your CSS file with a .module.css extension (e.g., MyComponent.module.css).
* Write your styles as usual, using regular CSS syntax. Example:

css

/\* MyComponent.module.css \*/

.container {

background-color: #f0f0f0;

padding: 20px;

border-radius: 5px;

}

.text {

color: blue;

font-size: 18px;

}

**2. Importing the CSS Module in a Component**

* Import the CSS Module file into your React component. Instead of a regular import, import it as an object, which will allow you to access each class name as a property on that object.

jsx

import React from 'react';

import styles from './MyComponent.module.css'; // Importing as 'styles'

function MyComponent() {

return (

<div className={styles.container}>

<p className={styles.text}>Hello, this is a component with CSS Modules!</p>

</div>

);

}

export default MyComponent;

**3. Applying Scoped Classes with className**

* When using CSS Modules, the className attribute takes the form styles.className, where className corresponds to the name of a class in your CSS Module.
* Each CSS class defined in the module will be automatically scoped to the component, ensuring no class name conflicts with styles in other components.

jsx

<div className={styles.container}>

<p className={styles.text}>Hello, this is a component with CSS Modules!</p>

</div>

**4. Understanding How CSS Modules Work**

* When you import a CSS Module, each class is converted to a unique identifier. This unique identifier is based on the component name, file path, and original class name, which avoids global conflicts.
* For instance, a .container class in MyComponent.module.css may become something like MyComponent\_container\_\_3jk4d, ensuring it won’t clash with .container classes in other CSS Modules.

**5. Conditional Classes with CSS Modules**

* If you need to apply classes conditionally, you can use template literals or libraries like classnames.
* Example using template literals:

jsx

<div className={`${styles.container} ${isActive ? styles.active : ''}`}>

Conditional Styling Example

</div>

* Example using the classnames library:

jsx

import classNames from 'classnames';

<div className={classNames(styles.container, { [styles.active]: isActive })}>

Conditional Styling Example

</div>

**6. CSS Module Variables (Optional)**

* Some projects support CSS variable imports, allowing you to use JavaScript variables directly in CSS Modules. This depends on your build setup and may require extra configuration or use of CSS-in-JS for complex dynamic styling.

**7. Benefits of Using CSS Modules in React**

* **Scoped Styling**: CSS Modules automatically scope styles to individual components, preventing global namespace pollution.
* **Automatic Class Naming**: Classes are generated dynamically, avoiding conflicts between classes in different modules.
* **Modularity and Maintainability**: CSS Modules promote modular code, making it easier to maintain styles in larger projects.

**Styling Approach Summary**

* **External CSS Stylesheets**: Great for global styling or when you want to keep styles separate from JavaScript.
* **CSS Modules**: Ideal when you need modular, component-specific styling to prevent conflicts.
* **Inline Styling**: Useful for dynamic or conditional styles that change based on component state but less scalable for large projects.
* React Props

In React, **props** (short for "properties") are a way of passing data from one component to another. They act like function arguments in JavaScript, allowing components to be dynamic and reusable by receiving information from their parent components.

**1. What Are Props?**

Props are objects that hold values or methods passed from a **parent component** to a **child component**. They make components more dynamic and configurable by enabling the passing of data from a parent component to a child component at the time of rendering.

For instance, if you have a UserProfile component, you might want to pass it a name and age as props, so the component displays personalized information based on the given data.

**2. How to Pass Props to a Component**

Props are passed to child components by adding them as attributes in the JSX tag when you render a component.

jsx

function App() {

return (

<UserProfile name="Ankit" age={21} />

);

}

Here, name="Ankit" and age={21} are props. In this example, UserProfile is the child component receiving the name and age props from its parent component App.

**3. Accessing Props in a Component**

Inside the UserProfile component, you can access the name and age props using the props object passed to the component.

For functional components:

jsx

function UserProfile(props) {

return (

<div>

<h1>Name: {props.name}</h1>

<p>Age: {props.age}</p>

</div>

);

}

Or, you can destructure the props for cleaner syntax:

jsx

function UserProfile({ name, age }) {

return (

<div>

<h1>Name: {name}</h1>

<p>Age: {age}</p>

</div>

);

}

**4. Props in Class Components**

In class-based components, props are accessed using this.props:

jsx

class UserProfile extends React.Component {

render() {

return (

<div>

<h1>Name: {this.props.name}</h1>

<p>Age: {this.props.age}</p>

</div>

);

}

}

**5. Passing Functions as Props**

You can also pass functions as props, which is helpful for handling events. For example, if you want the child component to trigger an action defined in the parent component:

jsx

function App() {

const handleButtonClick = () => {

alert("Button clicked!");

};

return <UserProfile onButtonClick={handleButtonClick} />;

}

function UserProfile({ onButtonClick }) {

return <button onClick={onButtonClick}>Click Me</button>;

}

In this case, onButtonClick is passed as a prop to UserProfile, allowing it to execute a function defined in App when the button is clicked.

**6. Props Are Read-Only**

Props are **immutable** within the component they are passed to, meaning a component cannot change its own props. If a component needs to maintain a state that changes, it should use **state** instead of props.

React enforces this immutability to ensure that the data flow remains **one-directional** (parent to child). The parent can re-render and pass new props, but the child cannot directly modify its own props.

**7. Default Props**

React allows you to set default values for props, which are used if no value is provided. This is useful for ensuring a component always has valid data to work with.

jsx

function UserProfile({ name = "Guest", age = 18 }) {

return (

<div>

<h1>Name: {name}</h1>

<p>Age: {age}</p>

</div>

);

}

In this example, if name or age is not provided when UserProfile is used, the component will default to "Guest" for name and 18 for age.

**8. Prop Types and Validation**

For larger projects, validating prop types can help prevent bugs. You can specify the expected type of each prop by using PropTypes:

jsx

import PropTypes from 'prop-types';

function UserProfile({ name, age }) {

return (

<div>

<h1>Name: {name}</h1>

<p>Age: {age}</p>

</div>

);

}

UserProfile.propTypes = {

name: PropTypes.string.isRequired,

age: PropTypes.number.isRequired,

};

Here, UserProfile expects name to be a string and age to be a number. If the parent component does not pass props of these types, React will log a warning in the console.

**Summary**

* **Props** are a mechanism to pass data from a parent component to a child component.
* They allow components to be **dynamic** and **reusable**.
* Props are **read-only** and should not be modified by the child component.
* **Default props** and **prop validation** with PropTypes can help manage component behavior and ensure type safety.

React conditionals

In React, conditionals allow you to render different components, elements, or content based on certain conditions, like whether data is loaded, a user is logged in, or specific user input. Since React uses JavaScript expressions, you can use JavaScript conditional logic in your components to control what gets displayed.

**1. Using if Statements**

The simplest way to handle conditional rendering is with if statements. This approach is usually used outside of JSX (JavaScript XML) and can control which component or element to render based on the condition.

**Example:**

jsx

function Greeting({ isLoggedIn }) {

if (isLoggedIn) {

return <h1>Welcome back!</h1>;

}

return <h1>Please sign up.</h1>;

}

Here, the Greeting component checks the isLoggedIn prop. If it’s true, it renders “Welcome back!”; otherwise, it renders “Please sign up.”

**2. Using Ternary Operators**

The ternary operator is a concise way to conditionally render content inside JSX. It’s ideal for simple conditions where you need to choose between two options.

**Syntax:**

jsx

condition ? <ComponentIfTrue /> : <ComponentIfFalse />

**Example:**

jsx

function Greeting({ isLoggedIn }) {

return (

<div>

{isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please sign up.</h1>}

</div>

);

}

This example does the same thing as the previous one but uses a ternary operator for inline conditional rendering.

**3. Using Logical && Operator**

The && operator is useful when you want to conditionally render something based on a single condition. If the condition is true, it renders the following element or component. If false, it renders nothing.

**Syntax:**

jsx

condition && <ComponentIfTrue />

**Example:**

jsx

function Notification({ hasNotifications }) {

return (

<div>

<h1>Your Dashboard</h1>

{hasNotifications && <p>You have new notifications!</p>}

</div>

);

}

In this example, if hasNotifications is true, the text "You have new notifications!" is rendered. If hasNotifications is false, nothing is rendered.

**4. Using if...else Statements with return**

For more complex conditional rendering with multiple conditions, you can use multiple if...else statements. In this case, you often return different elements based on each condition.

**Example:**

jsx

function StatusMessage({ status }) {

if (status === 'loading') {

return <p>Loading...</p>;

} else if (status === 'success') {

return <p>Data loaded successfully!</p>;

} else if (status === 'error') {

return <p>There was an error loading the data.</p>;

} else {

return <p>Unknown status.</p>;

}

}

Here, StatusMessage renders a different message based on the value of status.

**5. Conditional Rendering with switch Statements**

A switch statement can be useful for handling multiple cases, similar to multiple if...else conditions. It’s a good choice when you have multiple, mutually exclusive options to render.

**Example:**

jsx

function StatusMessage({ status }) {

switch (status) {

case 'loading':

return <p>Loading...</p>;

case 'success':

return <p>Data loaded successfully!</p>;

case 'error':

return <p>There was an error loading the data.</p>;

default:

return <p>Unknown status.</p>;

}

}

In this case, StatusMessage evaluates status and renders the appropriate message based on the matched case.

**6. Conditional Rendering with Inline Conditional Functions**

Sometimes, to keep code modular, it’s helpful to separate conditional logic into functions. You can then call these functions in JSX.

**Example:**

jsx

function Greeting({ isLoggedIn }) {

const getMessage = () => {

return isLoggedIn ? "Welcome back!" : "Please sign up.";

};

return <h1>{getMessage()}</h1>;

}

Here, the getMessage function handles the conditional logic, allowing the JSX to remain clean and focused on structure.

**7. Conditional CSS Classes and Styling**

In addition to conditional rendering of elements, you may want to apply CSS classes or styles conditionally based on some state or prop value. You can achieve this by using ternary operators or conditional statements directly in your className or style attributes.

**Example:**

jsx

function Button({ isPrimary }) {

return (

<button className={isPrimary ? "btn-primary" : "btn-secondary"}>

Click me

</button>

);

}

In this example, Button applies different CSS classes based on the value of isPrimary.

**Inline Styling Example:**

jsx

function Alert({ isError }) {

return (

<div style={{ color: isError ? "red" : "green" }}>

{isError ? "Error!" : "Success!"}

</div>

);

}

Here, the Alert component changes its text color based on whether isError is true or false.

**8. Short-Circuit Rendering in React**

In React, you can use the || (OR) operator to set fallback content if a particular value is null or undefined.

**Example:**

jsx

function UserProfile({ userName }) {

return (

<div>

<h1>Welcome, {userName || "Guest"}</h1>

</div>

);

}

If userName is not provided, the OR operator will fall back to rendering "Guest."

**Summary**

* **if statements**: Used outside JSX for simple conditional rendering with multiple return statements.
* **Ternary operators**: Commonly used inside JSX for concise conditional rendering.
* **Logical &&**: Used for single-condition rendering, ideal for optional elements.
* **if...else statements**: Useful for more complex conditions, especially with multiple return points.
* **Switch statements**: Helpful for rendering based on multiple distinct cases.
* **Conditional functions**: Enable modular code by extracting conditional logic into helper functions.
* **Conditional CSS classes/styles**: Control styling dynamically based on conditions.
* **Short-circuit rendering**: Use || for fallback values

React Lists

In React, **lists** are commonly used to display dynamic data in the form of arrays, such as a list of items, posts, or users. React provides powerful ways to render these arrays efficiently.

**1. Rendering Lists in React**

React lets you render lists by **mapping** over arrays, using JavaScript's .map() function. This method generates a new array of JSX elements based on the original array’s items.

**Basic Example**

If you have an array of items, you can render them as follows:

jsx

function ItemList() {

const items = ["Apple", "Banana", "Orange"];

return (

<ul>

{items.map((item) => (

<li>{item}</li>

))}

</ul>

);

}

In this example:

* items.map((item) => <li>{item}</li>) creates a new array of <li> elements.
* Each item in items array is displayed in a list.

**2. Using Unique Keys in Lists**

React uses **keys** to uniquely identify each element in a list. Keys help React detect changes in the list, such as added, removed, or reordered items, and update the DOM efficiently without re-rendering the entire list.

**Example with Keys**

A common approach is to use an item’s id or index as the key:

jsx

function ItemList() {

const items = [

{ id: 1, name: "Apple" },

{ id: 2, name: "Banana" },

{ id: 3, name: "Orange" },

];

return (

<ul>

{items.map((item) => (

<li key={item.id}>{item.name}</li>

))}

</ul>

);

}

In this example:

* Each <li> element is assigned a unique key using item.id.
* **Keys must be unique among siblings** (elements at the same level). This uniqueness helps React efficiently manage list updates.

**3. Avoiding Index as a Key**

While using the array index as a key is common, it’s generally not recommended, especially if the list might change dynamically (e.g., sorting, adding, or removing items). Using the index can cause unexpected behavior since React uses keys to identify elements; if the key changes, React may lose track of the correct element.

**Example (Using Index as Key)**

jsx

function ItemList() {

const items = ["Apple", "Banana", "Orange"];

return (

<ul>

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

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

))}

</ul>

);

}

Here, the index is used as the key. This is only advisable if:

* The list is static and won’t change.
* The order of items won’t change.

**4. Using Lists of Components**

Lists in React can contain other components, making it easier to organize and separate concerns. Each component in the list can receive props, such as item data and the key.

**Example**

jsx

function Item({ item }) {

return <li>{item.name}</li>;

}

function ItemList() {

const items = [

{ id: 1, name: "Apple" },

{ id: 2, name: "Banana" },

{ id: 3, name: "Orange" },

];

return (

<ul>

{items.map((item) => (

<Item key={item.id} item={item} />

))}

</ul>

);

}

In this example:

* The Item component is used to display each item in the list.
* Each Item component receives an item prop containing data for each list element.

**5. Handling Conditional Rendering in Lists**

You can conditionally render items in a list by filtering the data or adding conditional logic directly in the map function.

**Example: Filtered List Rendering**

jsx

function ItemList() {

const items = [

{ id: 1, name: "Apple", inStock: true },

{ id: 2, name: "Banana", inStock: false },

{ id: 3, name: "Orange", inStock: true },

];

return (

<ul>

{items

.filter((item) => item.inStock)

.map((item) => (

<li key={item.id}>{item.name}</li>

))}

</ul>

);

}

In this example:

* Only items that have inStock: true are rendered, as determined by .filter((item) => item.inStock).
* This approach is useful for rendering lists based on certain conditions.

**6. Adding and Removing Items from Lists**

React lists often come with features for **adding**, **removing**, and **updating** items. Typically, this requires maintaining the list data in the component’s **state**.

**Example**

jsx

import { useState } from "react";

function ItemList() {

const [items, setItems] = useState([

{ id: 1, name: "Apple" },

{ id: 2, name: "Banana" },

]);

const addItem = () => {

const newItem = { id: items.length + 1, name: "New Item" };

setItems([...items, newItem]);

};

const removeItem = (id) => {

setItems(items.filter((item) => item.id !== id));

};

return (

<div>

<ul>

{items.map((item) => (

<li key={item.id}>

{item.name}{" "}

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

</li>

))}

</ul>

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

</div>

);

}

In this example:

* addItem adds a new item to the list by updating the items state.
* removeItem removes an item based on its id.

**7. Rendering Nested Lists**

You can render lists within lists by mapping over arrays within arrays, which can be useful for displaying hierarchical or nested data.

**Example: Nested List**

jsx

function CategoryList() {

const categories = [

{ id: 1, name: "Fruits", items: ["Apple", "Banana", "Orange"] },

{ id: 2, name: "Vegetables", items: ["Carrot", "Peas", "Broccoli"] },

];

return (

<div>

{categories.map((category) => (

<div key={category.id}>

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

<ul>

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

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

))}

</ul>

</div>

))}

</div>

);

}

In this example:

* categories is a list of category objects, each containing a nested list of items.
* Each category and its items are displayed using nested .map() calls.

**8. Optimizing Large Lists with react-window or react-virtualized**

For large lists, rendering every item can slow down performance. To handle this, libraries like react-window and react-virtualized provide virtualized lists that only render items in the viewport.

**Example of react-window**

jsx

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

function LargeItemList({ items }) {

return (

<List

height={300} // height of the list container

itemCount={items.length} // total items

itemSize={35} // height of each item

width={300} // width of the list container

>

{({ index, style }) => (

<div style={style}>{items[index]}</div>

)}

</List>

);

}

In this example, only visible items in the viewport are rendered, improving performance for long lists.

**Summary**

* **Basic List Rendering**: Use .map() to transform arrays into lists of elements.
* **Keys**: Assign unique keys to each item to help React track changes in the list.
* **Avoid Using Index as Key**: Use id or another unique identifier when possible.
* **Component Lists**: Create reusable components for each list item to separate structure and logic.
* **Conditional List Rendering**: Use .filter() or inline conditionals to selectively render items.
* **State for Dynamic Lists**: Use React state to add, remove, or update list items.
* **Nested Lists**: Render hierarchical data with nested .map() calls.
* **Optimization**: Use libraries like react-window for performance with large lists.

**React Forms**

React forms are essential for gathering input from users in React applications. React provides ways to handle forms both in controlled and uncontrolled manners.

**1. Controlled Components**

* **Concept**: A controlled component in React is one where form data is handled by the component's state. In other words, the state of the input field is the "single source of truth."
* **How It Works**:
  + When a user types into a form field, an onChange event is triggered.
  + This event updates the component's state, and then React re-renders the component with the updated state.
  + Since the state is what dictates the value of the form, the component's rendering reflects the updated input.
* **Example**:

javascript

import React, { useState } from 'react';

function ControlledForm() {

const [name, setName] = useState('');

const handleChange = (e) => {

setName(e.target.value);

};

const handleSubmit = (e) => {

e.preventDefault();

alert(`Submitted name: ${name}`);

};

return (

<form onSubmit={handleSubmit}>

<label>

Name:

<input type="text" value={name} onChange={handleChange} />

</label>

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

</form>

);

}

export default ControlledForm;

* **Advantages**:
  + Easier to validate and manipulate the input values as they are directly tied to the component’s state.
  + Better control over the form values and the component’s UI.

**2. Uncontrolled Components**

* **Concept**: Uncontrolled components let the DOM handle the form data directly instead of storing it in the React state. To access form values, we use Refs, which provide a way to access the DOM elements directly.
* **How It Works**:
  + Instead of setting the value through React state, we assign a ref to the input element.
  + When we need the value, we retrieve it directly from the DOM using the ref.
* **Example**:

javascript

import React, { useRef } from 'react';

function UncontrolledForm() {

const nameRef = useRef();

const handleSubmit = (e) => {

e.preventDefault();

alert(`Submitted name: ${nameRef.current.value}`);

};

return (

<form onSubmit={handleSubmit}>

<label>

Name:

<input type="text" ref={nameRef} />

</label>

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

</form>

);

}

export default UncontrolledForm;

* **Advantages**:
  + Sometimes simpler to implement when you do not need full control over form data.
  + Works better when integrating with third-party libraries that directly interact with the DOM.

**3. Handling Multiple Inputs**

* In cases where a form has multiple fields, we can store each input’s state in a single object and use the input’s name attribute to identify the field.
* **Example**:

javascript

import React, { useState } from 'react';

function MultiInputForm() {

const [formData, setFormData] = useState({ firstName: '', lastName: '' });

const handleChange = (e) => {

const { name, value } = e.target;

setFormData((prevData) => ({

...prevData,

[name]: value,

}));

};

const handleSubmit = (e) => {

e.preventDefault();

alert(`First Name: ${formData.firstName}, Last Name: ${formData.lastName}`);

};

return (

<form onSubmit={handleSubmit}>

<label>

First Name:

<input type="text" name="firstName" value={formData.firstName} onChange={handleChange} />

</label>

<label>

Last Name:

<input type="text" name="lastName" value={formData.lastName} onChange={handleChange} />

</label>

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

</form>

);

}

export default MultiInputForm;

**4. Form Validation**

* Validation is important to ensure that users input the correct data format.
* There are several ways to validate form data in React:
  + Using regular expressions and state-based checks.
  + Using libraries like Formik or React Hook Form, which provide built-in validation options.
  + Handling validation in onChange or onSubmit events.

**5. Third-Party Form Libraries**

* **Formik**: Formik simplifies forms in React by providing built-in functions to manage the form’s state, validation, and submission.
* **React Hook Form**: This library uses refs, making it efficient with uncontrolled components. It has a simpler syntax, reduces re-renders, and integrates well with TypeScript.

**React Router**

React Router is a popular library used in React applications to manage navigation and enable routing between different components or pages. It allows developers to build single-page applications (SPAs) that can have multiple "pages" or views by mapping URL paths to specific React components.

**1. Setting Up React Router**

* To use React Router, you need to install the react-router-dom package.
* Installation:

bash

npm install react-router-dom

* Import necessary components from react-router-dom:

javascript

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

**2. Basic Components of React Router**

**a. BrowserRouter**

* This is a wrapper component that enables routing within a React application. It should be placed at the top level of your component tree to provide routing functionality throughout the app.
* Typically, we name it Router for simplicity:

javascript

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

**b. Routes and Route**

* Routes is a container for all Route components. It enables React Router to render the right component based on the current URL path.
* Each Route defines a path and the component to render when the app’s URL matches that path.
* Example:

javascript

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

import Home from './Home';

import About from './About';

function App() {

return (

<Router>

<Routes>

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

<Route path="/about" element={<About />} />

</Routes>

</Router>

);

}

export default App;

* Here, navigating to / renders the Home component, and navigating to /about renders the About component.

**c. Link and NavLink**

* **Link**: Link is used to create navigation links between different routes without causing a full page reload. It replaces the traditional <a> tag in React applications.

javascript

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

function Navbar() {

return (

<nav>

<Link to="/">Home</Link>

<Link to="/about">About</Link>

</nav>

);

}

* **NavLink**: NavLink is similar to Link but adds styling to indicate the active link (the currently active route). You can apply styles conditionally based on whether the link is active.

javascript

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

function Navbar() {

return (

<nav>

<NavLink to="/" activeClassName="active">Home</NavLink>

<NavLink to="/about" activeClassName="active">About</NavLink>

</nav>

);

}

**3. Dynamic Routing with Route Parameters**

* React Router allows you to create dynamic routes with parameters, useful for displaying different content based on the URL.
* To define a dynamic route, use a colon : followed by the parameter name in the path.
* Example:

javascript

<Route path="/user/:userId" element={<UserProfile />} />

* In UserProfile component, you can access the route parameter using the useParams hook:

javascript

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

function UserProfile() {

const { userId } = useParams();

return <div>User ID: {userId}</div>;

}

**4. Programmatic Navigation**

* Sometimes, you may want to navigate to a different route programmatically (e.g., after a form submission or a button click).
* For this, React Router provides the useNavigate hook.
* Example:

javascript

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

function Home() {

const navigate = useNavigate();

const goToAbout = () => {

navigate('/about');

};

return <button onClick={goToAbout}>Go to About</button>;

}

**5. Nested Routes**

* React Router allows nesting routes, which is useful for structuring complex applications.
* In nested routes, a parent route can render child routes within a particular component.
* Example:

javascript

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

import Dashboard from './Dashboard';

import Profile from './Profile';

import Settings from './Settings';

function App() {

return (

<Routes>

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

<Route path="profile" element={<Profile />} />

<Route path="settings" element={<Settings />} />

</Route>

</Routes>

);

}

* Here, navigating to /dashboard/profile renders the Profile component inside Dashboard.

**6. Route Protection (Private Routes)**

* For secure routes, you may need to restrict access based on user authentication or other conditions.
* To create a private route, wrap the protected component in a conditional component that checks the user’s status.
* Example:

javascript

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

function PrivateRoute({ children, isAuthenticated }) {

return isAuthenticated ? children : <Navigate to="/login" />;

}

// Usage in Routes

<Route path="/dashboard" element={<PrivateRoute isAuthenticated={userLoggedIn}><Dashboard /></PrivateRoute>} />

**7. Handling 404 (Not Found) Pages**

* If a user navigates to a non-existent route, it’s common to show a 404 error page.
* To do this, add a Route without a specific path at the end of your Routes list:

javascript

<Route path="\*" element={<NotFound />} />

* The NotFound component will render if no other path matches.