**Index**

1. **create a new React application**
2. **React features**
3. **React Fundamentals (with Prerequisite -JavaScript Fundamental )**
4. **Functional component**
5. **Functional component -props**
6. **Functional component –useState – literal and for object**
7. **Class Component**
8. **Class Component props and state**
9. **Class component t lifecycle**
10. **UseEffect Hooks**
11. **Ref, CreateRef , useRef**
12. **List & keys in-Depth**
13. **Map() method**
14. **String Array Rendering**
15. **Keys**
16. **Fixed stable keys**
17. **List with a component**
18. **Embedding map() with Jsx**
19. **Props & Jsx in-Depth**
    1. **Expression & string Literals**
    2. **Props default to true**
    3. **Spread Attributes**
    4. **Props.children property**
    5. **<React.Fragment>**
20. **Event Handling in-Depth**
21. **Quick Essential recap**
22. **SyntheticEvent**
23. **Form -PreventDefault-preventing default behaviour**
24. **<form> with multiple States**
25. **<form> states as object**
26. **<form> Form Single Event handler using dynamic key**
27. **<form> state Functional update –The Correct Way**
28. **<form> Checkbox or Radio – Input fields**
29. **Conditional Rendering – In-Depth**
30. **Conditional rendering with if…else…**
31. **Inline if-else [condition ? true:false]**
32. **Switch-case**
33. **Logical && Operator**
34. **React Router Version 6 (SPA)**
35. **React Router Dom Setup**
36. **Defining Routes**
37. **Menu Formatting**
38. **URL vs Routes & Default Page Setup**
39. **Dynamic Route – useParams() hook**
40. **Nested Routes - <Outlets>**
41. **Not Found Page (404)**
42. **Index Attribute – Reffering the parent Route**
43. **Dynamic Product Data useState() and Route Configuration**
44. **Context & useOutletContext() passing data to child component**
45. **State and useLocation() hook – The correct Way of Data Sharing**
46. **useRoutes() Hook Javascript instead of Jsx**
47. **NavLink – for BetterStyling**
48. **<Navigate> to Redirect**
49. **useNavigate() hook**
50. **Context API**

**9.1 Props Drilling**

**9.2 First Step- All in App.js**

**9.3 Creating Cart & Total Component**

**9.4 Context ApI Implementation**

**9.5 UseContext() – New and better**

1. **Redux**
2. **Asynchrous Redux**
3. **Redux toolkit**
4. **Handling Http Requests**
5. **Project guidelines**
6. **create a new React application**

To create a new React application, you can use the create-react-app command-line tool. Here's the step-by-step process:

First, make sure you have Node.js and npm installed on your machine. You can check this by running node -v and npm -v in your terminal. If these commands return a version number, you're good to go. If not, you'll need to install Node.js and npm first.

Install create-react-app globally on your machine. This is a package provided by Facebook to help bootstrap new React applications.

Use create-react-app to create a new React application.

# Step 1: Check Node.js and npm versions

node -v

npm -v

# Step 2: Install create-react-app

npm install -g create-react-app

# Step 3: Create a new React application

npx create-react-app my-app

Replace my-app with the name you want for your new application. After running these commands, you'll have a new directory with the same name as your application, filled with a basic React application structure. You can start the application by navigating into the directory and running npm start.

**2- React features**

1. Hot Reload

2.

1. **React Prerequisite -JavaScript Fundamental**

* Function Declarations and Arrow Functions
* Template Literals
* Short Conditionals: &&, ||, Ternary Operator
* Array Methods: .map(), .filter(), .reduce()
* Object Tricks: Property Shorthand, Destructuring, Spread Operator
* Promises + Async/Await Syntax
* ES Modules + Import / Export syntax
* Spread Operator + Rest Parameters
* Optional Chaining + Nullish Coalescing

**1. Function Declarations and Arrow Functions**

The basis of any React application is the component. In React, components are defined with both JavaScript functions. In React components return JSX elements .

// JavaScript function: returns any valid JavaScript type

function javascriptFunction() {

return "Hello world";

}

// React function component: returns JSX

function ReactComponent(props) {

return <h1>{props.content}</h1>;

}

==> Arrow function

// Arrow function syntax

const MyComponent = (props) => {

return <div>{props.content}</div>;

};

// Arrow function syntax (shorthand)

const MyComponent = (props) => <div>{props.content}</div>;

/\*

In the last example we are using several shorthands that arrow functions allow:

1. No parentheses around a single parameter

2. Implicit return (as compared to using the "return" keyword)

3. No curly braces for function body

\*/

**2. 2. Template Literals**

With the addition of ES6, we were given a newer form of string called a template literal,

which consists of two back ticks `` instead of single or double quotes.

Instead of having to use the + operator,

we can connect strings by putting a JavaScript expression within a special ${} syntax:

xpression (such as a variable) within a special ${} syntax:

/\*

Concatenating strings prior to ES6.Notice the awkward space after the word Hello?

\*/

function sayHello(text) {

return "Hello " + text + "!";

}

sayHello("React"); // Hello React!

/\*

Concatenating strings using template literals.

See how much more readable and predictable this code is?

\*/

function sayHelloAgain(text) {

return `Hello again, ${text}!`;

}

sayHelloAgain("React"); // Hello again, React!

**2.3. Short Conditionals: &&, ||, Ternary Operator**

conditionally show (or hide) JSX elements using simple if statements .

const isLoggedIn = true;

if (isLoggedIn) {

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

}

return <div>Who are you?</div>;

using Ternary operator

return isLoggedIn ? <div>Welcome back!</div> : <div>Who are you?</div>;

**2.4 Three Array Methods: .map(), .filter(), .reduce()**

map

const programmers = ["Reed", "John", "Jane"];

return (

<ul>

{programmers.map((programmer) => (

<li>{programmer}</li>

))}

</ul>

);

**JSX ( Js + html ) in depth**

**alternative of React.CreateElement you don’t need to import React for jsx**

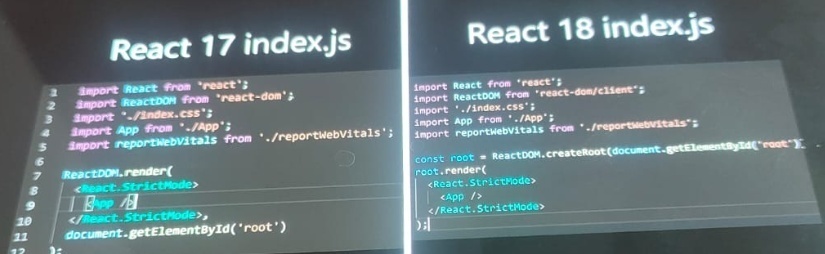
**HTML =>** <h1 class =”abc”>

**JSX =>** <h1 className =”abc”>

React 18

1. Dom Rendering

ReactDOM.render() is no longer supported . However it work better to switch CreateRoot() API .



**Why are Components Important in React?**

So, why are components such a big deal in React? There are a few reasons:

**Modularity and Reusability**

By breaking down your UI into components, you create a modular structure. This means you can develop, test, and maintain each piece of your app separately. Plus, once you've created a component, you can reuse it throughout your app, saving you time and effort.

### Efficiency in Development

Components encourage a more efficient development process. You can have different team members working on different components simultaneously without stepping on each other's toes. This speeds up the development process and encourages collaboration.

### Maintainability

Imagine you need to update the styling of a button that appears in multiple places in your app. With components, you only need to update the styling in one place, the button's component, and it will automatically reflect wherever it's used.

## Types of React Components

There are two main types of components in React: functional components and class components.

### functional components

This is the simplest way to define components in React. They are basically JavaScript functions that take in **props** (input data) and return **JSX** (JavaScript Syntax Extension) elements.

const MyComponent = () => {

    return (

      <div>

        <h1>Hello, World!</h1>

      </div>

    );

  };

  export default MyComponent;

Functional component with props [**Pass Data to Components using Props]**

function PersonInfo(props: { name: string; age: number }): JSX.Element {

  return (

    <div>

      <p>Name: {props.name}</p>

      <p>Age: {props.age}</p>

    </div>

  );

}

export default PersonInfo;

call from parent (app.tsx) **render with props**

<PersonInfo name="John" age={25} />

<PersonInfo name="Jane" age={22} />

**Props destructing**

Destructuring was introduced in ES6. It’s a JavaScript feature that allows us to extract multiple pieces of data from an array or object and assign them to their own variables.

const person = {

  firstName: "Lindsay",

  lastName: "Criswell",

  city: "NYC"

}

Before ES6, you had to access each property individually:

console.log(person.firstName) // Lindsay

console.log(person.lastName) // Criswell

console.log(person.city) // NYC

const { firstName, lastName, city } = person;

is equivalent to

const firstName = person.firstName

const lastName = person.lastName

const city = person.city

<div className = "App">

                <Greet active="KAPIL GARG" activeStatus = "CSE"/>

    </div>

const Greet = props =>{

    // Destructuring

    const {active, activeStatus} = props;

    return (

        <div>

              <h3> {active} </h3>

              <h1>{activeStatus}</h1>

        </div>

        )

  }

**Concept of state and useState() hook**

Without useState() react will not track the variable state and updated value will not reflect in dom .

export default function Counter() {

  const [count, setCount] = useState(0);

  function handleClick() {

    setCount(count + 1);

  }

  return (

    <button onClick={handleClick}>

      You pressed me {count} times

    </button>

  );

Update for obj literal

import React, { useState } from "react";

const CounterwithObjLitral = () => {

  const [state, setState] = useState({ cnt: 0, flag: false });

  const incrementCount = () => {

    setState((prevState) => ({

      ...prevState,

      cnt: prevState.cnt + 1,

    }));

  };

  const decrementCount = () => {

    setState((prevState) => ({

      ...prevState,

      cnt: prevState.cnt - 1,

    }));

  };

  const toggleFlag = () => {

    setState((prevState) => ({

      ...prevState,

      flag: !prevState.flag,

    }));

  };

  return (

    <div>

      <p>Count: {state.cnt}</p>

      <p>Flag: {state.flag ? "true" : "false"}</p>

      <button onClick={incrementCount}>Increment Count</button>

      <button onClick={decrementCount}>decrement Count</button>

      <button onClick={toggleFlag}>Toggle Flag</button>

    </div>

  );

};

export default CounterwithObjLitral;

**Lazy initial state/ load**

Normal

export default function Counter() {

  function initialState() {

    console.log('Initial state called -Expensive process' + Date.now());

    return 0;

  }

  const [count, setCount] = useState(initialState());

  function handleClick() {

    setCount(count + 1);

  }

  return (

    <button onClick={handleClick}>

      You pressed me {count} times

    </button>

  );

}

With Lazy initialization / load

export default function Counter() {

  function initialState() {

    console.log('Initial state called -Expensive process' + Date.now());

    return 0;

  }

  const [count, setCount] = useState(()=>initialState());

  function handleClick() {

    setCount(count + 1);

  }

  return (

    <button onClick={handleClick}>

      You pressed me {count} times

    </button>

  );

**Basic event Handling & Parameter passing**

Use fat arrow or return function rather calling

     <button onClick={() => setCount(count + 1)}>Increment</button>

      <button onClick={() => setCount(count - 1)}>Decrement</button>

**Apply internal and external css**

**External Stylesheet**

import "./styles.css";

use directly using className

**Inline CSS**

To implement inline CSS, you can create an object containing style references, which can be then called using the style attribute. For example:

const styles = {

  section: {

    fontSize: "18px",

    color: "#292b2c",

    backgroundColor: "#fff",

    padding: "0 20px"

  },

  wrapper: {

    textAlign: "center",

    margin: "0 auto",

    marginTop: "50px"

  }

}

It is then added to an element like this:

<section style={styles.section}>

  <div style={styles.wrapper}>

  </div>

</section>

**Primitive Types & Object Literal with useState()**

**Primitive Types with useState()**

import React, { useState } from "react";

const Counter = () => {

  const [count, setCount] = useState(0);

  const styles = {

    section: {

      fontSize: "18px",

      color: "#292b2c",

      backgroundColor: "#fff",

      padding: "0 20px",

    },

    wrapper: {

      margin: "0 auto",

      marginTop: "50px",

    },

  };

  // const increment = () => {

  //     setCount(count + 1);

  // };

  // const decrement = () => {

  //     setCount(count - 1);

  // };

  return (

    <>

      <section style={styles.section}>

        <div style={styles.wrapper}>

          <h1>Counter: {count}</h1>

          {/\* <button onClick={increment}>Increment</button>

            <button onClick={decrement}>Decrement</button> \*/}

          <button onClick={() => setCount(count + 1)}>Increment</button>

          <button onClick={() => setCount(count - 1)}>Decrement</button>

        </div>

      </section>

    </>

  );

};

export default Counter;

**Object Literal with useState()**

import React, { useState } from "react";

const CounterwithObjLitral = () => {

  const [state, setState] = useState({ cnt: 0, flag: false });

  const incrementCount = () => {

    setState((prevState) => ({

      ...prevState,

      cnt: prevState.cnt + 1,

    }));

  };

  const decrementCount = () => {

    setState((prevState) => ({

      ...prevState,

      cnt: prevState.cnt - 1,

    }));

  };

  const toggleFlag = () => {

    setState((prevState) => ({

      ...prevState,

      flag: !prevState.flag,

    }));

  };

  return (

    <div>

      <p>Count: {state.cnt}</p>

      <p>Flag: {state.flag ? "true" : "false"}</p>

      <button onClick={incrementCount}>Increment Count</button>

      <button onClick={decrementCount}>decrement Count</button>

      <button onClick={toggleFlag}>Toggle Flag</button>

    </div>

  );

};

export default CounterwithObjLitral;

**Class component**

The React.Component needs to extended to create a class component

Instead of return a render() method .

**Class Component state**

import React, { Component } from "react";

class CounterClass extends Component {

  state: { counter: number } = {

    counter: 0,

  };

  increment = () => {

    let c = this.state.counter;

    c++;

    this.setState((prevState, props) => {

      return { counter: c };

    });

  };

  decrement = () => {

    let c = this.state.counter;

    c--;

    this.setState((prevState, props) => {

      return { counter: c };

    });

  };

  render() {

    return (

      <div>

        <h2>Count: {this.state.counter}</h2>

        <button onClick={this.increment}>Increment</button>

        <button onClick={this.decrement}>Decrement</button>

      </div>

    );

  }

}

export default CounterClass;

**Class Component props**

type CounterState = {

  count: number;

};

type CounterProps = {

  name: string;

};

class CounterWithProps extends React.Component<CounterProps, CounterState> {

  constructor(props: any) {

    super(props);

    console.log("props", props);

    this.state = {

      count: 0,

    };

  }

  handleClick = () => {

    this.setState((prevState: { count: number }, props: CounterProps) => ({

      count: prevState.count + 1,

    }));

  };

  render() {

    return (

      <div>

        <h1>Hello, {this.props.name}!</h1>

        <p>Count: {this.state.count}</p>

        <button onClick={this.handleClick}>Increment</button>

      </div>

    );

  }

}

export default CounterWithProps;

**Component life cycle methods**

|  |  |
| --- | --- |
| **Load/Create** |  |
| **Unload/Cleanup** |  |

**Sure, let's break down the React component lifecycle into simple terms with some easy-to-understand examples:**

**Mounting Phase**:

* **constructor()**: Think of it like setting up your room before guests arrive. You're getting things in order.

javascript

constructor(props) {

super(props);

this.state = { count: 0 };

}

* **render()**: It's like decorating your room with furniture and making it look presentable.

javascript

render() {

return <div>{this.state.count}</div>;

}

* + **componentDidMount()**: Similar to welcoming guests when they arrive at your home. It's triggered once after the component is rendered for the first time.

javascript

componentDidMount() {

console.log('Component mounted!');

}

The **componentDidMount()** method is called once the component has been mounted into the DOM. It is typically used to set up any necessary event listeners or timers, perform any necessary API calls or data fetching, and perform other initialization tasks that require access to the browser's DOM API.

1. **Updating Phase**:
   * **componentDidUpdate()**: This is like when your guests are already there, and you update them with any changes or information.

javascript

componentDidUpdate(prevProps, prevState) {

if (prevState.count !== this.state.count) {

console.log('Count updated!');

}

}

But this method is not recommended for updating the state, as it can cause an infinite loop of rendering. It is primarily used for tasks such as making API calls, updating the DOM, or preparing the component to receive new data **componentWillUpdate**()  is often used in conjunction with  **componentDidUpdate**()  to handle component updates.

* + **shouldComponentUpdate()**: You're deciding whether to update something in your room before your guests see it.

javascript

shouldComponentUpdate(nextProps, nextState) {

return nextState.count !== this.state.count;

}

1. **Unmounting Phase**:
   * **componentWillUnmount()**: When your guests leave, you clean up the room and say goodbye.

javascript

componentWillUnmount() {

console.log('Component will unmount!');

}

**UseEffect Hooks**

**if you know class component lifecycle method you can think of useEffect() hook as componentDidMount(),componentDidUpdate() and componentWillUnmount() combined .**

**Mutations ,subscriptions ,timer,logging and other side effect are not allowed inside the main body of functional component .so useEffect hook is suitable to implement these side effects .**

The useEffect hook in React is used to perform side effects in function components. Side effects are actions that happen outside of the usual flow of your application, such as data fetching, subscriptions, or manually changing the DOM.

Here's a simple breakdown:

1. **What it does**:
   * useEffect allows you to perform side effects in your function components.
2. **When it runs**:
   * It runs after every render by default, including the first render.
3. **Why it's useful**:
   * You can use it to fetch data, set up subscriptions, or perform other side effects.
4. **How to use it**:
   * You pass a function to useEffect that contains the code for your side effect.
5. **Dependencies**:
   * You can also specify dependencies, which are variables that, when changed, should trigger the effect to run again.
6. **Clean-up**:
   * Optionally, you can return a function from the effect, which will be run before the component is unmounted or before the effect runs again.

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

function MyComponent() {

    const [data, setData] = useState(null);

    useEffect(() => {

        // This runs after every render

        fetchData();

        // Optionally, return a cleanup function

        return () => {

            // Clean-up code here (if needed)

        };

    }, [/\* dependencies \*/]);

    async function fetchData() {

        const response = await fetch('https://api.example.com/data');

        const newData = await response.json();

        setData(newData);

    }

    return (

        <div>

            {data ? <p>{data}</p> : <p>Loading...</p>}

        </div>

    );

}

In this example:

* We're fetching data from an API after each render using **useEffect**.
* We specify an empty dependency array (**[]**) to indicate that this effect doesn't depend on any variables, so it only runs once (similar to **componentDidMount** in class components).
* If you have dependencies, you list them inside the dependency array, and the effect will rerun whenever any of those dependencies change.

**Digital clock component with useEffect() Hook**

const DigitalClock = (): JSX.Element => {

  const [time, setTime] = useState(new Date());

  useEffect(() => {

    const timer = setInterval(() => {

      setTime(new Date());

    }, 1000);

    return () => {

      clearInterval(timer);

    };

  }, []);

  return (

    <div>

      <h1>{time.toLocaleTimeString()}</h1>

    </div>

  );

};

export default DigitalClock;

**UseEffect() with multiple state (skip or run for certain states)**

useEffect( ()=>{

    //statements

} ,[state1,state2,……stateN] )

**So in dependency of state if changes only then useEffect code run .**

**Condition Rendering (componentWillUnmount)**

**Call this component conditional when component unloaded return() function will run**

useEffect( ()=>{

    //code

return()=>{

    //clean up code

    }

})

**Ref- the older way**

**“ref” the reference of any DOM element using the React .**

In React, a ref is like a "reference" to a specific element in the DOM (Document Object Model), or to a component instance. Think of it as a way to directly access or interact with a particular element or component in your React application.

1. **What it does:**
   * A ref allows you to access or interact with a specific element or component directly.
2. **Why it's useful:**
   * It's useful when you need to focus on an input field, measure the size or position of an element, or trigger imperative animations or actions.
3. **How to use it:**
   * You create a ref using React.createRef() or the useRef() hook.
   * You attach the ref to a React element using the ref attribute.
   * Then, you can access the element or component using the ref's current property.

function MyComponent() {

    const inputRef = useRef(null);

    function focusInput() {

        inputRef.current.focus();

    }

    return (

        <div>

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

            <button onClick={focusInput}>Focus Input</button>

        </div>

    );

}

**list of common use cases for using refs in React:**

1. **Managing Focus:**
   * Auto-focusing on an input field when a component mounts.
   * Setting focus on a specific element when navigating to a new section of a page.
2. **Accessing DOM Elements:**
   * Integrating with third-party libraries that require direct access to DOM elements.
   * Manipulating the DOM directly for specific UI interactions or animations.
3. **Implementing Imperative Animations:**
   * Controlling animations imperatively, such as triggering animations on scroll or hover.
   * Animating elements that are not directly controlled by React's state or props.
4. **Interacting with Embedded Media:**
   * Accessing and controlling embedded media players, such as video or audio players.
   * Implementing custom controls or interactions for embedded media elements.
5. **Integrating with External APIs:**
   * Interacting with external APIs or libraries that require direct access to DOM elements.
   * Managing interactions with browser APIs like localStorage, navigator, etc.
6. **Handling Form Inputs:**
   * Accessing form input values directly for form validation or submission.
   * Implementing custom form controls that require direct interaction with the DOM.
7. **Managing Third-Party Components:**
   * Integrating and controlling third-party UI components that require direct access to DOM elements.
   * Implementing custom behavior or interactions with third-party components.
8. **Implementing Custom Hooks:**
   * Building custom hooks that encapsulate complex logic and may require access to DOM elements.
   * Managing state or behavior that spans multiple components using refs within custom hooks.
9. **Scrolling Behavior:**
   * Implementing custom scrolling behavior, such as smooth scrolling or infinite scrolling.
   * Handling scroll-based animations or effects.
10. **Managing Selections and Cursors:**
    * Implementing custom text selection behavior or managing cursor positions in text editors.
    * Building custom selection controls or interactions.

**CreateRef (class component)**

**In React 18, createRef is a function that allows you to create a reference to a React element or component.**

**Here's a simpler breakdown:**

1. Ref: Think of a "ref" as a way to reference a specific element or component in your React application.
2. createRef(): This is a function provided by React that you can use to create a new ref.

**So, when you use createRef(), you're basically making a "placeholder" that you can later use to refer to a specific element or component in your React code. This can be useful when you need to directly interact with a DOM element or a React component instance.**

import React, { Component } from 'react';

class MyComponent extends Component {

  constructor(props) {

    super(props);

    // Create a ref using createRef()

    this.myRef = React.createRef();

  }

  componentDidMount() {

    // Accessing the DOM node using the ref

    this.myRef.current.focus();

  }

  render() {

    return (

      <input

        type="text"

        ref={this.myRef} // Attaching the ref to the input element

      />

    );

  }

}

export default MyComponent;

1. Inside the constructor, we create a ref using **React.createRef()**. This creates a reference called **myRef**.
2. In the **render()** method, we attach this ref to an **<input>** element using the **ref** attribute.
3. In the **componentDidMount()** lifecycle method, we use the ref to focus on the input element when the component mounts.

**UseRef (functional component)**

In React 18, useRef() is a Hook that provides a way to create a mutable object that persists across renders. This object can hold a reference to a DOM element or a value that you want to persist between renders.

import React, { useRef } from 'react';

function MyComponent() {

  // Creating a ref using useRef()

  const myRef = useRef(null);

  // Using the ref to focus on an input element

  const focusInput = () => {

    myRef.current.focus();

  };

  return (

    <div>

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

      <button onClick={focusInput}>Focus Input</button>

    </div>

  );

}

export default MyComponent;

1. We import **useRef** from **react**.
2. Inside the functional component **MyComponent**, we call **useRef()** to create a ref. We initialize it with **null**.
3. We attach this ref to an **<input>** element using the **ref** attribute.
4. We define a function **focusInput** that uses **myRef.current** to focus on the input element.
5. When the button is clicked, the **focusInput** function is called, which focuses on the input element.

# List & keys

**This section is for collection of data and rendering on screen(dom).**

**Primitive type of array of number ,string, Boolean and object ,we loop through this type of data with properly rendering**

**Map() method**

the **map()** method is used to iterate over an array and perform a transformation on each element of the array, returning a new array with the transformed elements.

// Define an array

const numbers = [1, 2, 3, 4, 5];

// Use map() to double each number in the array

const doubledNumbers = numbers.map(number => number \* 2);

console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]

The **map()** method **does not mutate the original array**; instead, it returns a new array with the results of calling the provided function on every element in the array.

const fruits = [

  { id: 1, name: 'Apple' },

  { id: 2, name: 'Banana' },

  { id: 3, name: 'Orange' },

  { id: 4, name: 'Grape' },

  // Add more fruits as needed

];

// Using map() to retrieve only the fruit names

const fruitNames = fruits.map(fruit => fruit.name);

console.log(fruitNames); // Output: ['Apple', 'Banana', 'Orange', 'Grape']

**string Array Rendering**

import React from "react";

import { JsxElement } from "typescript";

const FruitList = (): JSX.Element => {

  const fruits = [

    { id: 1, name: "Apple" },

    { id: 2, name: "Banana" },

    { id: 3, name: "Orange" },

    { id: 4, name: "Grape" },

    // Add more fruits as needed

  ];

  return (

    <div>

      {fruits.map((fruit) => (

        <p>{fruit.name}</p>

      ))}

    </div>

  );

};

export default FruitList;

**Keys**

**Should have stable key (array index is not preferred way as items can add/remove)**

**Updated code as follow**

      {fruits.map((fruit,index) => (

        <p key={index}>{fruit.name}</p>

      ))}

**Fixed Stable key**

**Updated code as follow**

   {fruits.map((fruit) => (

        <p key={fruit.id}>{fruit.name}</p>

      ))}

**Proper Way (create a ListItem component and call) - improved architecture for react app**

    {fruits.map((fruit) => (

        <ListItem key={fruit.id.toString()} name={fruit.name} />

      ))}

const ListItem = ({ name }: { name: string }): JSX.Element => {

  return <p>{name}</p>;

};

export default ListItem;

**Embedding map() in Jsx - Improved way to write bigger react App**

import React from "react";

import { JsxElement } from "typescript";

import ListItem from "./ListItem";

const FruitList = (): JSX.Element => {

  const fruits = [

    { id: 1, name: "Apple" },

    { id: 2, name: "Banana" },

    { id: 3, name: "Orange" },

    { id: 4, name: "Grape" },

    // Add more fruits as needed

  ];

  const fruitsJsx = fruits.map((fruit) => (

    <ListItem key={fruit.id.toString()} name={fruit.name} />

  ));

  return <div>{fruitsJsx}</div>;

};

export default FruitList;

# Props & Jsx in Depth

In this section we will try to dive deeper with various combinations & tweaks to deal with props & JSX.

We will also learn how the content of component is passed to “props” as children property.

**Expression & string Literals**

In React, you often use expressions and string literals within JSX to dynamically render content based on the state of your components or other data. JSX allows you to embed JavaScript expressions inside curly braces **{}**.

1. **Using Expressions**
2. function MyComponent() {
3. const name = 'John';
4. const age = 30;
5. return (
6. <div>
7. <p>{name}</p> {/\* Embedding variable \*/}
8. <p>{age + 5}</p> {/\* Performing arithmetic operation \*/}
9. <p>{Math.random()}</p> {/\* Function call \*/}
10. </div>
11. );
12. }

**Using String Literals:**

You can include string literals directly within JSX by enclosing them in quotes.

function MyComponent() {

  return (

    <div>

      <p>Hello, World!</p> {/\* String literal \*/}

      <p>{"Hello, " + "World!"}</p> {/\* String concatenation using expressions \*/}

      <p>{`Hello, ${name}`}</p> {/\* String interpolation \*/}

    </div>

  );

}

**Using Ternary Operator:**

JSX also allows you to use the ternary operator for conditional rendering.

function MyComponent({ isLoggedIn }) {

  return (

    <div>

      {isLoggedIn ? <p>Welcome, User!</p> : <p>Please log in</p>}

    </div>

  );

}

**Using JavaScript Object Properties:**

You can use JavaScript object properties dynamically within JSX.

function MyComponent() {

  const user = {

    name: 'John',

    age: 30

  };

  return (

    <div>

      <p>Name: {user.name}</p>

      <p>Age: {user.age}</p>

    </div>

  );

}

**Props default to true**

In React, handling boolean props and setting default values for props are common tasks. Here's how you can handle boolean props and set default values to true:

1. **Handling Boolean Props:**

When passing boolean props to a component, you can simply use the prop name itself to represent true, and omitting the prop or passing false will be considered as false

function MyComponent({ isActive }) {

  return (

    <div>

      {isActive ? <p>Active</p> : <p>Inactive</p>}

    </div>

  );

}

**Spread Attributes (pass object as individual prop)**

The spread operator (**...**) in JavaScript can be used to pass props easily in React. It allows you to pass all the properties of an object as individual props.

In below example **ParentComponent** is rendering **ChildComponent** and passing props using the spread operator (**...person**). This means that all the properties of the **person** object (**name** and **age**) will be passed as individual props to the **ChildComponent**.

This way, you can easily pass multiple props without explicitly listing them one by one

import React from 'react';

function ChildComponent(props) {

  return (

    <div>

      <p>Name: {props.name}</p>

      <p>Age: {props.age}</p>

    </div>

  );

}

function ParentComponent() {

  const person = {

    name: 'John',

    age: 30

  };

  return (

    <div>

      {/\* Using spread operator to pass props \*/}

      <ChildComponent {...person} />

    </div>

  );

}

export default ParentComponent;

**Props.children property [pass content to child component ]**

In React, the **props.children** property allows components to pass children elements directly into their output. This is particularly useful when you want to wrap other components or content within a parent component.

// ParentComponent wraps its children with a div and adds a class

function ParentComponent({ children }) {

  return (

    <div className="parent-component">

      {children}

    </div>

  );

}

// App component where ParentComponent is used

function App() {

  return (

    <div>

      <ParentComponent>

        <h1>Hello from ParentComponent!</h1>

        <p>This is a child paragraph.</p>

      </ParentComponent>

    </div>

  );

}

**<React.Fragment>**

In React, a fragment is a way to group multiple children elements without adding extra nodes to the DOM. It's essentially a lightweight wrapper that allows you to return multiple elements from a component's render method without needing to create an additional HTML element (like div ).

<React.Fragment> … </React.Fragment> or simple use <>…</>

function MyComponent() {

  return (

    <React.Fragment>

      <h1>Hello</h1>

      <p>World</p>

    </React.Fragment>

  );

}

# Event Handling in-Depth

**Quick Essential recap**

Camel case without parenthesis

  const handleClick = () => {

    // Code to handle button click

  };

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

To pass parameter

  const handleClick = (inp:string) => {

    // Code to handle button click

  };

 <button onClick={()=>handleClick(“test”)}>Click me</button>

**SyntheticEvent**

**Input Event -TextBox**

const EventsComponent = (): JSX.Element => {

  const handleEvent = (e: React.SyntheticEvent) => {

     console.log(e);

  };

  return (

    <div>

      {/\* Your component JSX goes here \*/}

      <button onClick={handleEvent}>Click me</button>

    </div>

  );

};

**Form -PreventDefault-preventing default behavior**

In React, when you're handling form submissions, you may want to prevent the default behavior of the form submission, especially when using AJAX requests or performing client-side validation. To achieve this, you can use the **preventDefault()** method available on the event object.

**Example 1** - below login form that uses React hooks. It initializes a state variable called formData with empty values for username and password. It also defines a handleSubmit function that prevents the default form submission behavior and logs the form data when the form is submitted.

function Login() {

  const [formData, setFormData] = useState({

    // initialize form data state

    username: '',

    password: ''

  });

  const handleSubmit = (event:SyntheticEvent) => {

    event.preventDefault(); // Prevents the default form submission behavior

    // Your custom logic for handling form submission

    console.log('Form submitted with data:', formData);

    // Here you can proceed with any further logic such as AJAX requests, form validation, etc.

  };

  const handleChange = (event:SyntheticEvent) => {

    const { name, value } = event.target as HTMLInputElement;

    setFormData({

      ...formData,

      [name]: value

    });

  };

  return (

    <form onSubmit={handleSubmit}>

      <label>

        Username:

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

      </label>

      <br />

      <label>

        Password:

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

      </label>

      <br />

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

    </form>

  );

}

**Example 2**

const EventsPreventDefaults: React.FC = () => {

  const [input1, setInput1] = useState("");

  const [input2, setInput2] = useState("");

  const handleSubmit = (event: React.FormEvent) => {

    event.preventDefault();

    // Do something with the form data

    console.log("Input 1:", input1);

    console.log("Input 2:", input2);

  };

  return (

    <form onSubmit={handleSubmit}>

      <input

        type="text"

        value={input1}

        onChange={(e) => setInput1(e.target.value)}

      />

      <input

        type="text"

        value={input2}

        onChange={(e) => setInput2(e.target.value)}

      />

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

    </form>

  );

};

export default EventsPreventDefaults;

**<form> with multiple States**

const FormsWithMultipleState: React.FC = () => {

  const [firstName, setFirstName] = useState("");

  const [lastName, setLastName] = useState("");

  const [mobile, setMobile] = useState("");

  const handleSubmit = (e: React.FormEvent) => {

    e.preventDefault();

    // Display the form values

    console.log("First Name:", firstName);

    console.log("Last Name:", lastName);

    console.log("Mobile:", mobile);

  };

  return (

    <div>

      <form onSubmit={handleSubmit}>

        <div>

          <label htmlFor="firstName">First Name:</label>

          <input

            type="text"

            id="firstName"

            value={firstName}

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

          />

        </div>

        <div>

          <label htmlFor="lastName">Last Name:</label>

          <input

            type="text"

            id="lastName"

            value={lastName}

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

          />

        </div>

        <div>

          <label htmlFor="mobile">Mobile:</label>

          <input

            type="text"

            id="mobile"

            value={mobile}

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

          />

        </div>

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

      </form>

      <div>

        <h2>Form Values:</h2>

        <p>First Name: {firstName}</p>

        <p>Last Name: {lastName}</p>

        <p>Mobile: {mobile}</p>

      </div>

    </div>

  );

};

**<form> states as object (with Single Event handler using dynamic key => same handler handleChange for all 3 input)**

const FormsWithCombineState: React.FC = () => {

  const [formData, setFormData] = useState({

    firstName: "",

    lastName: "",

    mobile: "",

  });

  const handleSubmit = (e: React.FormEvent) => {

    e.preventDefault();

    // Display the form values

    console.log("First Name:", formData.firstName);

    console.log("Last Name:", formData.lastName);

    console.log("Mobile:", formData.mobile);

  };

  const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {

    setFormData({

      ...formData,

      [e.target.id]: e.target.value,

    });

  };

  return (

    <div>

      <form onSubmit={handleSubmit}>

        <div>

          <label htmlFor="firstName">First Name:</label>

          <input

            type="text"

            id="firstName"

            value={formData.firstName}

            onChange={handleChange}

          />

        </div>

        <div>

          <label htmlFor="lastName">Last Name:</label>

          <input

            type="text"

            id="lastName"

            value={formData.lastName}

            onChange={handleChange}

          />

        </div>

        <div>

          <label htmlFor="mobile">Mobile:</label>

          <input

            type="text"

            id="mobile"

            value={formData.mobile}

            onChange={handleChange}

          />

        </div>

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

      </form>

      <div>

        <h2>Form Values:</h2>

        <p>First Name: {formData.firstName}</p>

        <p>Last Name: {formData.lastName}</p>

        <p>Mobile: {formData.mobile}</p>

      </div>

    </div>

  );

};

**Functional update –The Correct Way –**

**Update in above code.**

  setFormData((prevState) => {

      return {

        ...prevState,

        [e.target.id]: e.target.value,

      };

    });

**The main reason to do so React Does not update the state immediately as it has its own method or schedule to update the state .**

**Checkbox or Radio – Input fields**

**For checkbox we don’t have value property. We have to use checked property .**

const FormsWithRadioButtonAndCheckBox: React.FC = () => {

  const [formData, setFormData] = useState({

    firstName: "",

    lastName: "",

    mobile: "",

    gender: "",

    habits: {

      reading: false,

      writing: false,

      coding: false,

    },

  });

  const handleSubmit = (e: React.FormEvent) => {

    e.preventDefault();

    // Display the form values

    console.log("First Name:", formData.firstName);

    console.log("Last Name:", formData.lastName);

    console.log("Mobile:", formData.mobile);

    console.log("Gender:", formData.gender);

    console.log("Habits:", formData.habits);

  };

  const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {

    if (e.target.type === "checkbox") {

      setFormData((prevState) => {

        return {

          ...prevState,

          habits: {

            ...prevState.habits,

            [e.target.id]: e.target.checked,

          },

        };

      });

    } else {

      setFormData((prevState) => {

        return {

          ...prevState,

          [e.target.id]: e.target.value,

        };

      });

    }

  };

  return (

    <div>

      <form onSubmit={handleSubmit}>

        <div>

          <label htmlFor="firstName">First Name:</label>

          <input

            type="text"

            id="firstName"

            value={formData.firstName}

            onChange={handleChange}

          />

        </div>

        <div>

          <label htmlFor="lastName">Last Name:</label>

          <input

            type="text"

            id="lastName"

            value={formData.lastName}

            onChange={handleChange}

          />

        </div>

        <div>

          <label htmlFor="mobile">Mobile:</label>

          <input

            type="text"

            id="mobile"

            value={formData.mobile}

            onChange={handleChange}

          />

        </div>

        <div>

          <label htmlFor="gender">Gender:</label>

          <input

            type="radio"

            id="gender"

            name="gender"

            value="male"

            checked={formData.gender === "male"}

            onChange={handleChange}

          />

          <label htmlFor="male">Male</label>

          <input

            type="radio"

            id="gender"

            name="gender"

            value="female"

            checked={formData.gender === "female"}

            onChange={handleChange}

          />

          <label htmlFor="female">Female</label>

        </div>

        <div>

          <label htmlFor="reading">Reading:</label>

          <input

            type="checkbox"

            id="reading"

            checked={formData.habits.reading}

            onChange={handleChange}

          />

        </div>

        <div>

          <label htmlFor="writing">Writing:</label>

          <input

            type="checkbox"

            id="writing"

            checked={formData.habits.writing}

            onChange={handleChange}

          />

        </div>

        <div>

          <label htmlFor="coding">Coding:</label>

          <input

            type="checkbox"

            id="coding"

            checked={formData.habits.coding}

            onChange={handleChange}

          />

        </div>

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

      </form>

      <div>

        <h2>Form Values:</h2>

        <p>First Name: {formData.firstName}</p>

        <p>Last Name: {formData.lastName}</p>

        <p>Mobile: {formData.mobile}</p>

        <p>Gender: {formData.gender}</p>

        <p>Habits:</p>

        <ul>

          <li>Reading: {formData.habits.reading ? "Yes" : "No"}</li>

          <li>Writing: {formData.habits.writing ? "Yes" : "No"}</li>

          <li>Coding: {formData.habits.coding ? "Yes" : "No"}</li>

        </ul>

      </div>

    </div>

  );

};

export default FormsWithRadioButtonAndCheckBox;

# Conditional Rendering – In-Depth

**Conditional rendering with if…else…**

interface ConditionalIfElseProps {

    condition: boolean;

}

const ConditionalIfElse: React.FC<ConditionalIfElseProps> = ({ condition }) => {

    if (condition) {

        return (

            <div>

                <ProductComponent />

            </div>

        );

    } else {

        return (

            <div>

                <p>No product available.</p>

            </div>

        );

    }

};

export default ConditionalIfElse;

**Inline if-else [condition ? true:false] Ternary operator**

interface ConditionalIfElseProps {

  condition: boolean;

}

const ConditionalIfElse: React.FC<ConditionalIfElseProps> = ({ condition }) => {

  return (

    <div>

    { condition

            ?    <ProductComponent />

            : <p>No product available.</p>

    }

    </div>

  );

};

export default ConditionalIfElse;

**Switch-case**

type SwitchCaseProps = {

    value: string;

};

const SwitchCase: React.FC<SwitchCaseProps> = ({ value }) => {

    switch (value) {

        case 'case1':

            return <div>Case 1</div>;

        case 'case2':

            return <div>Case 2</div>;

        case 'case3':

            return <div>Case 3</div>;

        default:

            return <div>Default Case</div>;

    }

};

export default SwitchCase;

**Logical && Operator**

const LogicalAndOperator: React.FC = () => {

  const isLoggedIn = true;

  const hasPermission = true;

  const canAccessResource = isLoggedIn && hasPermission;

  return (

    <div>

      <h1>Logical AND Operator Example</h1>

      <p>Is user logged in? {isLoggedIn.toString()}</p>

      <p>Does user have permission? {hasPermission.toString()}</p>

      <p>Can access resource? {canAccessResource.toString()}</p>

    </div>

  );

};

export default LogicalAndOperator;

# Chapter-8 React Router Version 6 (SPA)

**SPA (single page App ) vs MPA (Multiple page App)**

**Single Page Application (SPA):**

* Example: A social media feed where you can scroll endlessly to see new posts without the page refreshing or loading a new URL.

**Multi-Page Application (MPA):**

* Example: An online shopping website where clicking on different categories (like "Electronics," "Clothing," etc.) takes you to separate pages where you can view products specific to that category.

**In Simple Terms:**

* **SPA** is like a single**, dynamic document** where everything happens within one "page" without the need to load separate pages.
* **MPA** is like having **multiple standalone documents** (pages) that are accessed individually, and each page reloads separately.

**Summary:**

* **SPA** keeps everything on a single web page, offering a more seamless and interactive experience.
* **MPA** consists of multiple separate pages, each with its own URL, typically resulting in full page reloads when navigating between them.

To make a single page application we have to implement client-side routing .

There are 2 libraries often used

1. React Router (we going to use **v6**)
2. Reach Router

**8.1 React Router Dom Setup**

**To install React Router DOM, you can follow these steps:**

1. **Navigate to Your Project Directory:** Open your terminal or command prompt and navigate to the directory of your React project.
2. **Install React Router DOM Package:** Run the following command to install the React Router DOM package using npm (Node Package Manager):

npm install react-router-dom@6

1. **Import BrowserRouter:** Import **BrowserRouter** from **react-router-dom** in your **index.tsx** file.

import React from 'react';

import ReactDOM from 'react-dom';

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

import App from './App';

ReactDOM.render(

  <BrowserRouter>

    <App />

  </BrowserRouter>,

  document.getElementById('root')

);

**Defining Routes**

<BrowserRouter> is imported from react-router-dom to enable routing in the application.

**BrowserRouter – All the routes and components will be wrapped inside <BrowserRouter> component , as it will store the clear URLs internally , using which you actually route through all the components or basically the entire application .**

**In index.js**

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

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

  <React.StrictMode>

    <BrowserRouter>

       <App />

    </BrowserRouter>

  </React.StrictMode>

);

**Add Routes and route**

import Home from "./Components/Home";

import Contact from "./Components/Contact";

import About from "./Components/About";

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

function App() {

  return (

    <div className="App">

      <ul>

        <li>

          <a href="/home">Home</a>

        </li>

        <li>

          <a href="/about">About</a>

        </li>

        <li>

          <a href="/contact">Contact</a>

        </li>

      </ul>

      <hr></hr>

      <Routes>

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

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

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

      </Routes>

    </div>

  );

}

**Update link to make Spa**

import Home from "./Components/Home";

import Contact from "./Components/Contact";

import About from "./Components/About";

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

function App() {

  return (

    <div className="App">

      <ul>

        <li>

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

        </li>

        <li>

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

        </li>

        <li>

          <Link to="/contact">Contact</Link>

        </li>

      </ul>

      <hr></hr>

      <Routes>

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

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

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

      </Routes>

    </div>

  );

}

**Menu Formatting**

   <ul>

        <li>

          <Link to="/home" style={{ textDecoration: "none" }}>

            Home

          </Link>

        </li>

        <li>

          <Link to="/about" style={{ textDecoration: "none" }}>

            About

          </Link>

        </li>

        <li>

          <Link to="/contact" style={{ textDecoration: "none" }}>

            Contact

          </Link>

        </li>

      </ul>

li {

  margin: 10px;

  display: inline-block;

  padding: 10px;

  border: 1px solid #ccc;

  border-radius: 5px;

  background-color: #f9f9f9;

  cursor: pointer;

}

**8.2 URL vs Routes & Default Page Setup**

Link component updating the path in browser Url ,once url got changed router picked up url path and compare with routes and render the matched route .

This is happening because BrowserRouter keep track of changes and navigating history of URL .

Update code as follow

<Link to="/" style={{ textDecoration: "none" }}>

            Home

          </Link>

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

**8.3 Dynamic Route – useParams() hook**

**Step1 : Create a Dynamic Route in Routes**

  <Route path="/ProductList/:id" element={<ProductDetails />} />

**Step 2: Create a Link to generate url for dynamic route**

      <Link

          to={"/ProductList/" + product.id}

          style={{ textDecoration: "none" }}

        >

          {product.name}

        </Link>

**Step 3:**

**Retrieve dynamic route parameter value**

const { id } = useParams();

**8.4 Nested Routes - <Outlets>**

**1. Create a nested route**

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

          <Route path=":id" element={<ProductDetails />} />

        </Route>

1. **Update the nested route link with their content rendering place with <Outlet>**

 const ProductList = ({ products }) => {

  return (

    <>

      <ul style={{ listStyle: "none" }}>

        <li style={{ display: "inline-block", textAlign: "left" }}>

          <ProductBox key={1} product={"Apples"} />

        </li>

        <li style={{ display: "inline-block", textAlign: "left" }}>

          <ProductBox key={1} product={"Banana"} />

        </li>

        <li style={{ display: "inline-block", textAlign: "left" }}>

          <ProductBox key={1} product={"Grapes"} />

        </li>

      </ul>

      <div>

        <Outlet />

      </div>

    </>

  );

};

**Not Found Page (404)**

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

* 1. **Index Attribute – Referring the parent Route**

The `index` prop in React Router v6 is used to specify a default route that will be rendered when no other child route matches.

In the provided code, `<Route index element={<Home />} />` means that the `Home` component will be rendered when the path is exactly `/`. If there are other nested routes, and none of them match, the `Home` component will still be rendered because it's marked as the `index` route.

      <Routes>

        <Route path="/">

          <Route index element={<Home />} />

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

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

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

            <Route path=":id" element={<ProductDetails />} />

          </Route>

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

        </Route>

      </Routes>

* 1. **Dynamic Product Data useState() and Route Configuration**

const ProductList = ({ products }) => {

  const [productsdata, setProductsdata] = useState([

    { id: 1, name: "Apple", price: 100 },

    { id: 2, name: "Banana", price: 200 },

    { id: 3, name: "Grapes", price: 300 },

  ]);

  return (

    <>

      <ul style={{ listStyle: "none" }}>

        {productsdata.map((product) => (

          <li style={{ display: "inline-block", textAlign: "left" }}>

            <ProductBox key={product.id} product={product} />

          </li>

        ))}

      </ul>

      <div>

        <Outlet />

      </div>

    </>

  );

};

* 1. **Context & useOutletContext() passing data to child component**

When it comes to sharing the data to child component there is attribute called “context” which you can use with <outlet> component.

<Outlet context={object} />

With the “context” attribute ,you can pass the data as an Object literal .

The object is passed to the child component and to refer the passed data inside the child component you have the useOutletContext() hook .

  const data = useOutletContext();

* 1. **State and useLocation() hook – The correct Way of Data Sharing**

**Attach data in Link tag state attribute**

   <Link

          state={{ product }}

          to={"/ProductList/" + product.id}

          style={{ textDecoration: "none" }}

        >

          {product.name}

        </Link>

**Get the data using useLocation() hook**

 const prd = useLocation();

* 1. **useRoutes() Hook Javascript instead of Jsx**

In React, JSX is used to define the user interface of the application. It is a syntax extension to JavaScript that allows you to write HTML-like code within JavaScript.

Routes are a way to define different pages or views in your application. In JSX, you can use the <Route> component to define a route. The <Route> component takes a path prop, which is the URL path that will trigger the route. It also takes a component prop, which is the React component that will be rendered when the route is triggered.

For example, the following JSX code defines a route that will render the Home component when the user visits the / path:

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

You can also use JavaScript to define routes. To do this, you can use the useRoutes hook. The useRoutes hook takes an array of route objects as its argument. Each route object has a path property and a component property, just like the <Route> component.

const routes = [  
 {  
 path: "/",  
 component: Home,  
 },  
];  
  
const App = () => {  
 const router = useRoutes(routes);  
 return router;  
};

* 1. **NavLink – for BetterStyling**

Similar to <Link> but <NavLink> allow you can apply various changes to the <Link> as per state.

* 1. **<Navigate> to Redirect**

To redirect to your choice of Url , you can use the <Navigate> component

   <Navigate to={"/"}></Navigate>

import React from "react";

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

const NotFound = () => {

  return (

    <div>

      <Navigate to={"/"}></Navigate>

    </div>

  );

};

* 1. **useNavigate() hook**

using **<Navigate>** component is an easy way to navigate but in reality useNavigate() hook is preferred .

Reasons

Generally when you have form submissions , once data is filled and submitted .there is a need to navigate to the page you are trying to access .

Example => after login redirect to home/Dashboard page

import React from "react";

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

const NotFound = () => {

  const navigate = useNavigate();

  return (

    <div>

      <h1>404 Not Found</h1>

      <p>Sorry, the page you are looking for does not exist.</p>

      <button onClick={() => navigate("/")}>Go to Home</button>

    </div>

  );

};

export default NotFound;

suppose you just want move to previous page then you can also use

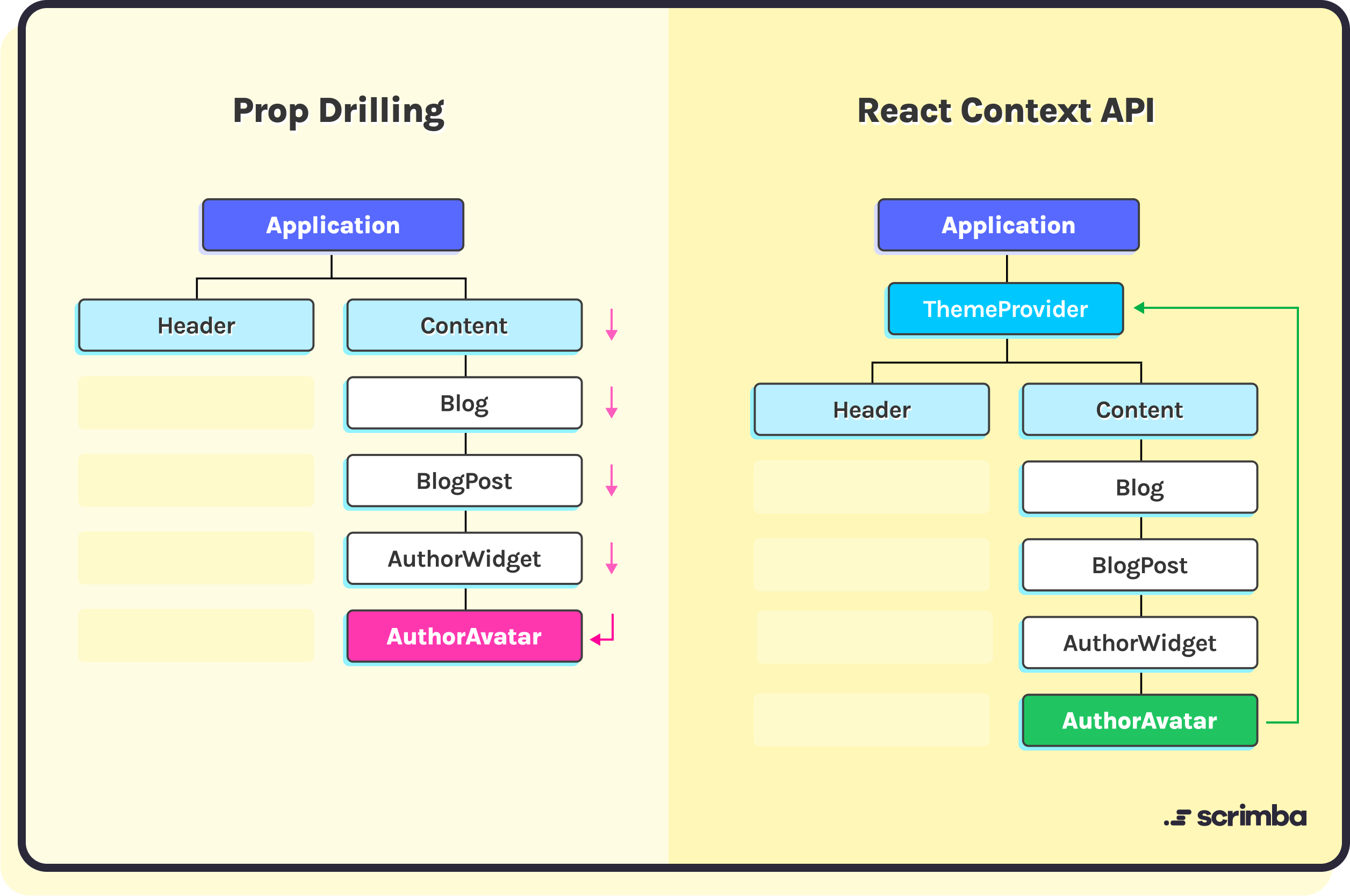
**navigate(-1)**

**Chapter 9 Context API**

**9.1 Props Drilling**

Props drilling, also known as prop passing or prop threading, refers to the process of passing data from one component to another by passing it down through multiple levels of nested components in a React application.

There is various ways like context API or redux to deal with state when you want it to be global or flexible across various components or an entire application.



**9.2 First Step- All in App.js**

**Purchase/Cart/Total component**

**9.3 Creating Cart & Total Component**

**9.4 Context ApI Implementation**

**9.5 UseContext() – New and better**