React Introduction:-

React is a **front-end JavaScript library** designed for building dynamic and interactive user interfaces. It primarily works with a **Virtual DOM** for efficient rendering.

Why Use React?

- **Component Reusability:** Build once, reuse everywhere.
- **Fast Updates:** Uses a diffing algorithm for minimal DOM updates.
- **Ecosystem:** Rich third-party libraries and tools.

---

2. React Upgrade

Keeping React updated ensures access to the latest features, bug fixes, and performance improvements.

Example: Using New React Features

- React 16 introduced **Hooks** (e.g., `useState` and `useEffect`).
- React 18 introduced **Concurrent Rendering** and automatic batching.

```
bash
Copy code
npm install react@latest react-dom@latest
```

---

3. ES6 in React

Modern JavaScript (ES6+) enables concise, powerful coding patterns in React.

Key Features in React Context

- **Default Parameters:**

  ```
  jsx
  Copy code
  function greet(name = "User") {
    return `Hello, ${name}!`;
  }
  ```

- **Modules:**

  ```
  jsx
  Copy code
  import React from "react";
  export const MyComponent = () => <div>Hello!</div>;
  ```

- **Template Literals:**

```
jsx
Copy code
const title = `Welcome to React`;
```

---

4. Render HTML

Rendering HTML is the first step in building any React app. React uses **ReactDOM** to bind components to the DOM.

ReactDOM Methods:

- `ReactDOM.render()`: Renders elements into the DOM (pre-React 18).
- `ReactDOM.createRoot()`: Initializes concurrent rendering (React 18+).

```
jsx
Copy code
import ReactDOM from "react-dom/client";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<h1>Hello, React!</h1>);
```

---

5. JSX

JSX allows embedding HTML-like syntax directly in JavaScript. JSX makes code more readable but must be transpiled using **Babel**.

Key Points About JSX:

- Must return a **single parent element**.
- Can include **JavaScript expressions** inside `{}`.
- Attributes use **camelCase** (e.g., `className` instead of `class`).

```
jsx
Copy code
const element = (
  <div>
    <h1>Welcome</h1>
    <p>This is JSX in action.</p>
  </div>
);
```

---

6. Components

React components represent independent, reusable pieces of UI.

Functional Components:

- Simple, stateless components.

- Can use Hooks for state and lifecycle methods.

```
jsx
Copy code
function Greeting() {
  return <h1>Hello!</h1>;
}
```

Class Components:

- Stateful and require lifecycle methods like `componentDidMount`.

```
jsx
Copy code
import React, { Component } from "react";

class Greeting extends Component {
  render() {
    return <h1>Hello!</h1>;
  }
}
```

---

7. Props

Props (**properties**) allow data to be passed between components. Props are **read-only** and immutable.

Default Props

```
jsx
Copy code
function Welcome({ name = "User" }) {
  return <h1>Welcome, {name}!</h1>;
}
```

Prop Types Validation

```
bash
Copy code
npm install prop-types
jsx
Copy code
import PropTypes from "prop-types";

Welcome.propTypes = {
  name: PropTypes.string.isRequired,
};
```

---

8. Events

React events are based on W3C events but written in camelCase. The event object is passed as a parameter.

Binding `this` in Class Components

```
jsx
Copy code
class Button extends Component {
  handleClick = () => {
    alert("Button clicked!");
  };

  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

---

9. Conditionals

React supports various ways to conditionally render elements.

1. Using `if` Statements

```
jsx
Copy code
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h1>Welcome Back!</h1>;
  }
  return <h1>Please Sign In.</h1>;
}
```

2. Ternary Operator

```
jsx
Copy code
const message = isLoggedIn ? "Welcome!" : "Sign In";
```

---

10. Lists

Lists are dynamically rendered using JavaScript's `map()` method.

Key Points:

- Each item requires a **unique `key`** for efficient DOM updates.
- Keys should be stable, unique, and predictable.

```
jsx
Copy code
function ItemList() {
  const items = ["Apple", "Banana", "Cherry"];
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
```

```
      ))}
    </ul>
  );
}
```

---

11. Forms

React forms use controlled components to manage form state.

Controlled Component Example

```
jsx
Copy code
function Form() {
  const [value, setValue] = React.useState("");

  return (
    <form>
      <input value={value} onChange={(e) => setValue(e.target.value)} />
    </form>
  );
}
```

Uncontrolled Component Example

```
jsx
Copy code
function Form() {
  const inputRef = React.useRef();

  const handleSubmit = () => {
    alert(inputRef.current.value);
  };

  return (
    <form>
      <input ref={inputRef} />
      <button onClick={handleSubmit}>Submit</button>
    </form>
  );
}
```

---

12. React Router

React Router enables navigation and routing in single-page applications.

Nested Routes

```
jsx
Copy code
import { Route, Routes } from "react-router-dom";

function App() {
```

```
  return (
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
    </Routes>
  );
}
```

---

13. React.memo

Optimizes functional components to prevent unnecessary re-renders.

Use Case

- Use `React.memo` for components that receive the same props repeatedly.

```
jsx
Copy code
const Greeting = React.memo(({ name }) => {
  console.log("Rendering");
  return <h1>Hello, {name}!</h1>;
});
```

---

14. CSS Styling

React supports multiple styling methods:

- **Inline Styles**:

  ```
  jsx
  Copy code
  const style = { color: "blue" };
  <h1 style={style}>Hello</h1>;
  ```

- **CSS-in-JS**: Styled-components, Emotion.

---

15. SASS Styling

SASS adds variables, nesting, and mixins to CSS.

Example

```
scss
Copy code
$primary: coral;

button {
  background: $primary;
```

```
  &:hover {
    background: darken($primary, 10%);
  }
}
```

---

1. React Introduction – Advanced:-

React is designed to handle the **view layer** of an application. It's not a framework like Angular; it's a library focused on building reusable UI components.

React Ecosystem Components:

- **State Management:** Redux, MobX, Context API.
- **Routing:** React Router.
- **Styling:** Styled Components, Emotion, Tailwind CSS.

Why Choose React?

- **Declarative Programming:** Focus on "what" to render rather than "how".
- **Unidirectional Data Flow:** State flows down from parent to child components, making debugging predictable.
- **Rich Ecosystem:** Thousands of libraries extend React's capabilities.

---

2. React Upgrade - Advanced Features

React continuously introduces **performance improvements** and **new features**:

- **React 18 Features:**
  - Concurrent Rendering.
  - Automatic Batching of Updates.
  - New `useTransition` Hook for UI transitions.

Example: Automatic Batching

Pre-React 18:

```
jsx
Copy code
setState1(newValue);
setState2(newValue); // Causes two renders.
```

React 18:

```
jsx
Copy code
setState1(newValue);
```

```
setState2(newValue); // Single render due to automatic batching.
```

---

3. ES6 in React - Advanced

Arrow Functions and `this` Binding:

Arrow functions automatically bind the context (`this`), reducing boilerplate in class components.

```
jsx
Copy code
class Button extends React.Component {
  handleClick = () => {
    console.log("Button clicked");
  };

  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

Rest/Spread Operators:

Easily handle state updates.

```
jsx
Copy code
const [user, setUser] = useState({ name: "Alice", age: 25 });
setUser({ ...user, age: 26 }); // Update age without modifying the entire
object.
```

---

4. Render HTML - Advanced

React allows rendering multiple components dynamically using **mapping** or **conditional logic**.

Dynamic Rendering Example:

```
jsx
Copy code
const items = ["Home", "About", "Contact"];

function Navbar() {
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
}
```

---

5. JSX - Advanced

JSX allows embedding complex logic into UI declarations.

Embedding Logic in JSX:

```
jsx
Copy code
function Status({ online }) {
  return <p>{online ? "Online" : "Offline"}</p>;
}
```

---

6. Components - Advanced

Component Lifecycle Methods in Class Components:

- **Mounting:** `componentDidMount`
- **Updating:** `componentDidUpdate`
- **Unmounting:** `componentWillUnmount`

```
jsx
Copy code
class Timer extends React.Component {
  componentDidMount() {
    this.interval = setInterval(() => console.log("Tick"), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return <h1>Timer Running</h1>;
  }
}
```

Functional Components + Hooks:

```
jsx
Copy code
function Timer() {
  useEffect(() => {
    const interval = setInterval(() => console.log("Tick"), 1000);
    return () => clearInterval(interval); // Cleanup on unmount.
  }, []);

  return <h1>Timer Running</h1>;
}
```

---

7. Props - Advanced

Props can be **objects, arrays, or even functions**.

Passing Functions as Props:

```
jsx
Copy code
function Child({ handleClick }) {
  return <button onClick={handleClick}>Click Me</button>;
}

function Parent() {
  const greet = () => alert("Hello from Parent!");
  return <Child handleClick={greet} />;
}
```

---

8. Events - Advanced

React wraps browser events in a **SyntheticEvent** to ensure cross-browser compatibility.

Prevent Default Behavior:

```
jsx
Copy code
function Form() {
  const handleSubmit = (e) => {
    e.preventDefault(); // Prevent page reload.
    console.log("Form submitted");
  };

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

---

9. Conditionals - Advanced

Short-Circuit Rendering:

Render components conditionally without `if` or ternary:

```
jsx
Copy code
function Message({ isLoggedIn }) {
  return isLoggedIn && <h1>Welcome Back!</h1>;
}
```

---

10. Lists - Advanced

Key Guidelines:

1. Avoid using array indices as keys when list order can change.
2. Use unique IDs for better performance.

---

11. Forms - Advanced

Controlled components are useful for managing complex form logic.

Controlled vs Uncontrolled Forms:

- **Controlled:** Form state is managed via React state.
- **Uncontrolled:** Use `ref` to directly access DOM elements.

---

12. React Router - Advanced

React Router offers **dynamic route matching** and **protected routes**.

Protected Routes Example:

```
jsx
Copy code
function ProtectedRoute({ isAuthenticated, children }) {
  return isAuthenticated ? children : <Navigate to="/login" />;
}
```

---

13. React.memo - Advanced

`React.memo` optimizes components by skipping re-renders if props haven't changed.

Comparison Logic:

You can provide a custom comparison function to control when re-renders occur.

```
jsx
Copy code
const Greeting = React.memo(
  ({ name }) => {
    console.log("Rendering Greeting");
    return <h1>Hello, {name}!</h1>;
  },
  (prevProps, nextProps) => prevProps.name === nextProps.name
);
```

---

14. CSS Styling in React - Advanced

Styled Components:

CSS-in-JS allows scoped styles for components.

```
bash
Copy code
npm install styled-components
jsx
Copy code
import styled from "styled-components";

const Button = styled.button`
  background: blue;
  color: white;
  padding: 10px;
`;

function App() {
  return <Button>Click Me</Button>;
}
```

---

15. SASS Styling - Advanced

SASS provides advanced features like **mixins** and **inheritance**.

Mixin Example:

```
scss
Copy code
@mixin button-styles($bg-color) {
  background-color: $bg-color;
  color: white;
  border: none;
}

button {
  @include button-styles(coral);
}
```

---

1. React Context API

The Context API enables sharing state across components without passing props manually at every level. It's ideal for **global state management** (e.g., user authentication, themes).

Steps to Use Context API:

1. Create a context using `React.createContext`.
2. Use a `Provider` to wrap components that need access to the context.
3. Use `useContext` to consume the context.

Example: Theme Context

```
jsx
Copy code
import React, { createContext, useContext, useState } from "react";

const ThemeContext = createContext();

function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");
  const toggleTheme = () => setTheme((prev) => (prev === "light" ? "dark" :
"light"));

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

function ThemedButton() {
  const { theme, toggleTheme } = useContext(ThemeContext);
  return (
    <button
      onClick={toggleTheme}
      style={{ background: theme === "light" ? "#fff" : "#333", color:
theme === "light" ? "#000" : "#fff" }}
    >
      Toggle Theme
    </button>
  );
}

export default function App() {
  return (
    <ThemeProvider>
      <ThemedButton />
    </ThemeProvider>
  );
}
```

---

2. Custom Hooks

Custom Hooks allow you to extract and reuse logic across components.

Example: Custom Hook for Fetching Data

```
jsx
Copy code
import { useState, useEffect } from "react";

function useFetch(url) {
```

```
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch(url)
      .then((response) => response.json())
      .then((data) => {
        setData(data);
        setLoading(false);
      });
  }, [url]);

  return { data, loading };
}

function App() {
  const { data, loading } = useFetch("https://api.example.com/items");

  if (loading) return <p>Loading...</p>;
  return <ul>{data.map((item) => <li key={item.id}>{item.name}</li>)}</ul>;
}
```

---

3. React Suspense and Lazy Loading

Suspense lets you wait for something (like code or data) to load before rendering the component.

Lazy Loading Components

```
jsx
Copy code
import React, { lazy, Suspense } from "react";

const LazyComponent = lazy(() => import("./LazyComponent"));

function App() {
  return (
    <Suspense fallback={<p>Loading...</p>}>
      <LazyComponent />
    </Suspense>
  );
}
```

---

4. Performance Optimization

React provides tools like `useMemo`, `useCallback`, and `React.memo` to optimize performance.

`useMemo`

Prevents unnecessary recalculations of expensive operations.

```
jsx
Copy code
function ExpensiveCalculationComponent({ number }) {
```

```
  const expensiveValue = useMemo(() => {
    console.log("Calculating...");
    return number * 2; // Simulate an expensive calculation.
  }, [number]);

  return <p>{expensiveValue}</p>;
}
```

`useCallback`

Memoizes a function so that it doesn't get recreated on every render.

```
jsx
Copy code
function Counter() {
  const [count, setCount] = useState(0);

  const increment = useCallback(() => {
    setCount((prev) => prev + 1);
  }, []);

  return <button onClick={increment}>Count: {count}</button>;
}
```

React.memo

Prevents re-rendering of functional components if props haven’t changed.

```
jsx
Copy code
const Child = React.memo(({ name }) => {
  console.log("Rendering Child");
  return <p>{name}</p>;
});

function Parent() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <Child name="React" />
    </div>
  );
}
```

---

5. Error Boundaries

Error boundaries catch JavaScript errors in a component tree and display a fallback UI.

Example: Creating an Error Boundary

```
jsx
Copy code
```

```
import React, { Component } from "react";

class ErrorBoundary extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError() {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    console.log(error, info);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}

function BuggyComponent() {
  throw new Error("Crash!");
}

export default function App() {
  return (
    <ErrorBoundary>
      <BuggyComponent />
    </ErrorBoundary>
  );
}
```

---

6. Higher-Order Components (HOC)

An HOC is a function that takes a component and returns a new component, adding additional functionality.

Example: Logging Props with HOC

```
jsx
Copy code
function withLogging(WrappedComponent) {
  return function EnhancedComponent(props) {
    console.log("Props:", props);
    return <WrappedComponent {...props} />;
  };
}

const ButtonWithLogging = withLogging(({ label }) =>
<button>{label}</button>);

export default function App() {
  return <ButtonWithLogging label="Click Me" />;
```

```
}
```

---

7. React Portals

Portals let you render components outside their parent DOM hierarchy.

Example: Rendering a Modal

```
jsx
Copy code
import ReactDOM from "react-dom";

function Modal({ children }) {
  return ReactDOM.createPortal(children, document.getElementById("modal-
root"));
}

function App() {
  return (
    <>
      <h1>Main App</h1>
      <Modal>
        <p>This is rendered in a portal.</p>
      </Modal>
    </>
  );
}
```

---

8. State Management (Redux)

Redux provides a centralized state management approach.

Example: Counter with Redux

1. **Reducer**:

   ```
   jsx
   Copy code
   function counterReducer(state = 0, action) {
     switch (action.type) {
       case "increment":
         return state + 1;
       case "decrement":
         return state - 1;
       default:
         return state;
     }
   }
   ```

2. **Store**:

   ```
   jsx
   Copy code
   ```

```
import { createStore } from "redux";

const store = createStore(counterReducer);
```

3. **Dispatch Actions**:

```
jsx
Copy code
store.dispatch({ type: "increment" });
console.log(store.getState()); // Output: 1
```

React Hooks are functions introduced in React 16.8 that let you use state and other React features in functional components. They simplify React development by allowing you to manage state, handle side effects, and more, without relying on class components.

---

1. Basic Hooks

useState

`useState` allows you to add state to functional components.

Example: Counter with `useState`

```
jsx
Copy code
import React, { useState } from "react";

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

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

---

useEffect

`useEffect` lets you perform side effects (e.g., data fetching, subscriptions) in functional components.

Example: Fetching Data

```
jsx
Copy code
```

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

function DataFetcher() {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/posts")
      .then((response) => response.json())
      .then((data) => {
        setData(data);
        setLoading(false);
      });
  }, []); // Empty array means this effect runs once after the component
mounts.

  if (loading) return <p>Loading...</p>;

  return (
    <ul>
      {data.slice(0, 5).map((item) => (
        <li key={item.id}>{item.title}</li>
      ))}
    </ul>
  );
}
```

---

useContext

`useContext` accesses the value of a React context directly.

Example: Theme Toggle with Context

```
jsx
Copy code
import React, { createContext, useContext, useState } from "react";

const ThemeContext = createContext();

function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");
  const toggleTheme = () => setTheme((prev) => (prev === "light" ? "dark" :
"light"));

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

function ThemedComponent() {
  const { theme, toggleTheme } = useContext(ThemeContext);
  return (
    <div style={{ background: theme === "light" ? "#fff" : "#333", color:
theme === "light" ? "#000" : "#fff" }}>
      <p>Current Theme: {theme}</p>
      <button onClick={toggleTheme}>Toggle Theme</button>
```

```
    </div>
  );
}

function App() {
  return (
    <ThemeProvider>
      <ThemedComponent />
    </ThemeProvider>
  );
}
```

---

2. Additional Hooks

useReducer

`useReducer` is an alternative to `useState` for managing complex state logic.

Example: Counter with `useReducer`

```
jsx
Copy code
import React, { useReducer } from "react";

function reducer(state, action) {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: "increment"
})}>Increment</button>
      <button onClick={() => dispatch({ type: "decrement"
})}>Decrement</button>
    </div>
  );
}
```

---

useRef

`useRef` creates a mutable reference to a DOM element or a value.

Example: Accessing a DOM Element

```
jsx
Copy code
import React, { useRef } from "react";

function TextInput() {
  const inputRef = useRef();

  const focusInput = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
}
```

---

useMemo

`useMemo` memoizes a computed value to prevent unnecessary recalculations.

Example: Expensive Calculation

```
jsx
Copy code
import React, { useState, useMemo } from "react";

function ExpensiveCalculation({ num }) {
  const expensiveValue = useMemo(() => {
    console.log("Calculating...");
    return num * 2;
  }, [num]);

  return <p>Result: {expensiveValue}</p>;
}

function App() {
  const [num, setNum] = useState(1);
  return (
    <div>
      <button onClick={() => setNum(num + 1)}>Increase</button>
      <ExpensiveCalculation num={num} />
    </div>
  );
}
```

---

useCallback

`useCallback` memoizes a function to prevent unnecessary re-creations.

Example: Memoized Callback

```
jsx
```

```
Copy code
import React, { useState, useCallback } from "react";

function Child({ onClick }) {
  console.log("Rendering Child");
  return <button onClick={onClick}>Click Me</button>;
}

const MemoizedChild = React.memo(Child);

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

  const handleClick = useCallback(() => {
    console.log("Button clicked");
  }, []);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment Parent</button>
      <MemoizedChild onClick={handleClick} />
    </div>
  );
}
```

---

3. Custom Hooks

Custom Hooks are user-defined functions that encapsulate reusable logic.

Example: Custom Hook for Window Size

```
jsx
Copy code
import React, { useState, useEffect } from "react";

function useWindowSize() {
  const [size, setSize] = useState({ width: window.innerWidth, height:
window.innerHeight });

  useEffect(() => {
    const handleResize = () => setSize({ width: window.innerWidth, height:
window.innerHeight });
    window.addEventListener("resize", handleResize);
    return () => window.removeEventListener("resize", handleResize);
  }, []);

  return size;
}

function App() {
  const { width, height } = useWindowSize();

  return (
    <p>
      Width: {width}, Height: {height}
    </p>
  );
}
```

4. Rules of Hooks

- **Call Hooks at the Top Level:** Don’t call hooks inside loops, conditions, or nested functions.
- **Call Hooks from React Functions:** Only use hooks in functional components or custom hooks.