React.js

Introduction

What is React?

- A JS library used to develop Single Page Application
- Single Page Application
  - Contains only one html page
  - Gets loaded only once when user visits the website
  - Once loaded, it sends the request to the server only to get the data
  - Executed only on the client side (inside a browser)
- React is used to develop client side applications
- React features
  - Has a component-driven architecture
  - Uses virtual DOM for improving the application performance
  - Eco-system: React Router, React Redux Toolkit, React Native

React: A Brief History

- **Created by**: Jordan Walke at Facebook (2011)
- **First used**: Facebook News Feed (2011)
- **Open-sourced**: May 2013 (React Conf 2015 marked major adoption)
- **Current maintainers**: Meta (Facebook) + community (OpenJS Foundation)

Key Milestones:

- 2013: Initial release
- 2015: React Native launched
- 2016: React Fiber architecture (v16+)
- 2018: Hooks introduced (v16.8)

- 2022: React 18 with concurrent features

Why React?

Declarative UI

```
// WHAT to render (not HOW to render it)
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}
```

Component-Based Architecture

```
// Reusable components
function App() {
  return (
    <Layout>
      <Header />
      <Sidebar />
      <Content />
    </Layout>
  );
}
```

Performance Benefits

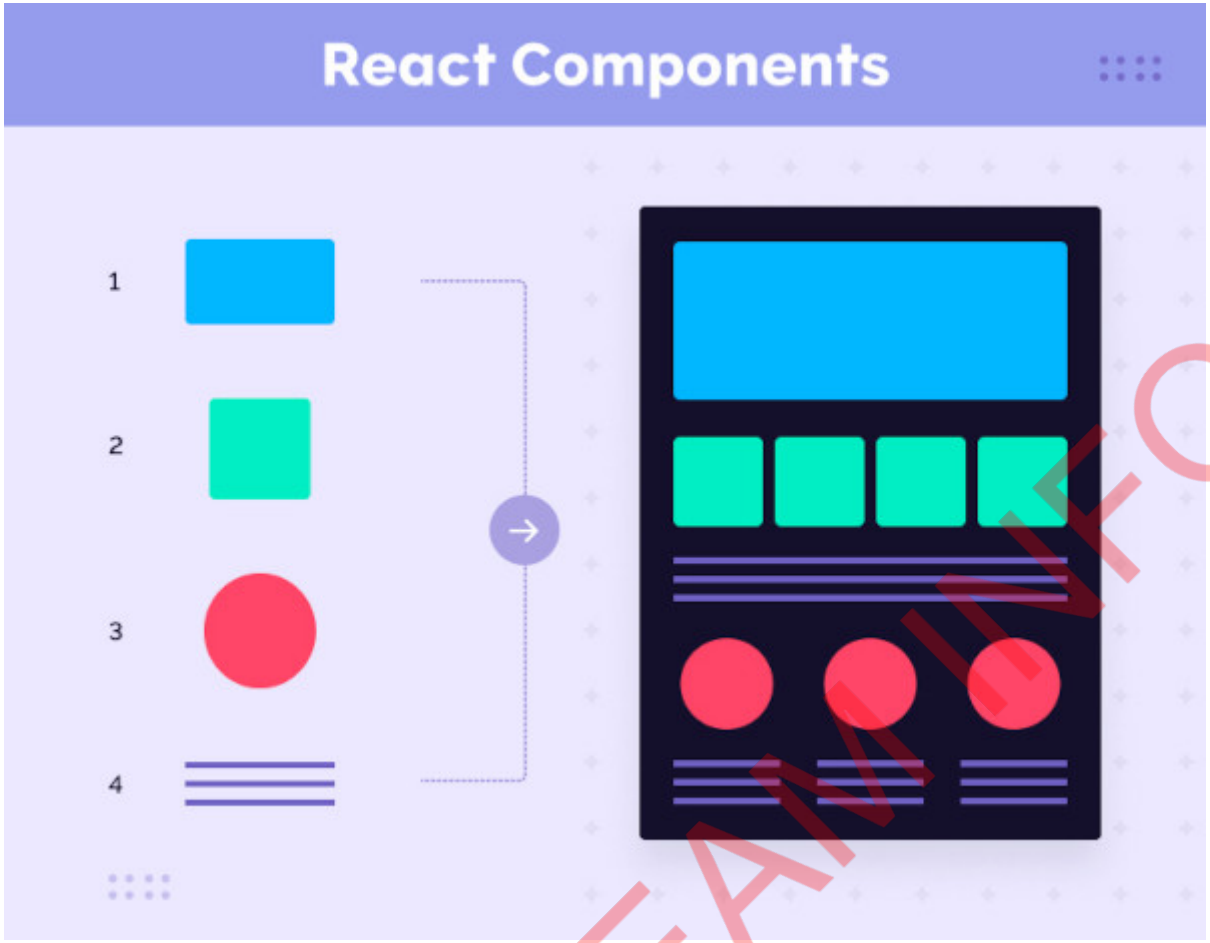
- Virtual DOM minimizes browser reflows
- Efficient reconciliation algorithm
- Code splitting and lazy loading support

React Ecosystem

- **State Management**: Redux, MobX, Context API
- **Routing**: React Router
- **UI Libraries**: Material-UI, Ant Design
- **Static Site Generators**: Next.js, Gatsby

React's Core Philosophy

1. **Component-Based Architecture:**
    - The first react core principle is component-based architecture.
    - It promotes a modular approach to building UIs via reusable components.
    - Moreover, each set of components encapsulates its logic and state, hence making it easier to manage and maintain complex user interfaces.



2. **Composition Over Inheritance**

```
// Combine small components
function Card({ children }) {
  return <div className="card">{children}</div>;
}

function Profile() {
  return (
```

```
      <Card>
        <Avatar />
        <UserDetails />
      </Card>
    );
  }
```

3. **Learn Once, Write Anywhere**
    - Web (React DOM)
    - Mobile (React Native)
    - VR (React 360)
    - Desktop (Electron + React)

Key Concepts

Component Types

Type	Description	When to Use
Functional	JavaScript functions with hooks	Modern React (95%+ cases)
Class	ES6 classes with lifecycle methods	Legacy codebases

Core React Features

1. **JSX**: HTML-like syntax in JavaScript
2. **State**: Component memory (useState, useReducer)
3. **Props**: Data passed to components
4. **Hooks**: Reusable stateful logic (useEffect, useContext)
5. **Context**: Global state without prop drilling

React uses

React's Limitations

1. **Steep Learning Curve**
    - JSX, state management, hooks
    - Build tools (Webpack, Babel)
2. **SEO Challenges**
    - Requires SSR (Next.js) for optimal SEO
3. **Rapid Changes**
    - Frequent updates (though backward-compatible)
4. **Tooling Overhead**
    - Typically needs transpilation (Babel) and bundling (Webpack)

React Alternatives

Library	Key Difference	Best For
Vue	More HTML-centric templates	Progressive adoption
Angular	Full MVC framework	Enterprise apps
Svelte	Compiles to vanilla JS	Performance-critical apps
SolidJS	React-like syntax, no VDOM	High-performance UIs

Choose React or Not?

- ☑ **Complex, interactive UIs**
- ☑ **Large-scale applications**
- ☑ **Cross-platform needs (web + mobile)**
- ☑ **Teams valuing component reusability**

❌ **Simple static websites**
❌ **Teams preferring HTML templates**
❌ **Projects requiring zero build step**

Industry Adoption

- **Facebook** (Original creator)
- **Instagram** (Web + Mobile)
- **Netflix** (UI performance)
- **Airbnb** (Component-driven design)
- **WhatsApp Web** (Real-time updates)

Pure React (Without JSX)

Core Concepts

- Direct manipulation of Virtual DOM using React's JavaScript API
- Uses `React.createElement()` to construct UI elements
- Manual DOM mounting with `ReactDOM.createRoot()`

Implementation Example

```
<!DOCTYPE html>
<html>
  <head>
    <script
      crossorigin
      src="https://unpkg.com/react@18/umd/react.development.js"
    ></script>
    <script
      crossorigin
      src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
    ></script>
  </head>
```

```
  <body>
    <div id="root1"></div>
    <div id="root2"></div>

    <script>
      // Simple element creation
      const root1 = ReactDOM.createRoot(document.getElementById("root1"));
      const h1 = React.createElement("h1", null, "Hello React");
      root1.render(h1);

      // Complex element hierarchy
      const root2 = ReactDOM.createRoot(document.getElementById("root2"));
      const h4 = React.createElement(
        "h4",
        { key: "h4" },
        "The library for web and native user interfaces"
      );
      const p = React.createElement(
        "p",
        { key: "p" },
        "React is a free and open-source front-end JavaScript library that aims to make building user interfaces based on
components more seamless."
      );
      const container = React.createElement("div", null, [h4, p]);
      root2.render(container);
    </script>
  </body>
</html>
```

Key Observations

1. **Verbose Syntax**: Each element requires explicit `createElement` call
2. **Manual Structure**: Nested elements become hard to visualize
3. **No Components**: Just direct Virtual DOM node creation
4. **Performance**: Still uses Virtual DOM diffing

Babel + JSX

Babel's Role

- **Transpiler**: Converts modern JS/JSX to browser-compatible code
- **JSX Transformation**: Converts HTML-like syntax to `React.createElement()` calls
- **Polyfills**: Adds support for newer JavaScript features

Enhanced Implementation

```
<!DOCTYPE html>
<html>
  <head>
    <script
      crossorigin
      src="https://unpkg.com/react@18/umd/react.development.js"
    ></script>
    <script
      crossorigin
      src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
    ></script>
    <script
      crossorigin
      src="https://unpkg.com/@babel/standalone/babel.min.js"
    ></script>
  </head>
  <body>
    <div id="root1"></div>
    <div id="root2"></div>

    <!-- Babel-processed scripts -->
    <script type="text/babel">
      // Component-based approach
      const root1 = ReactDOM.createRoot(document.getElementById("root1"));
```

```
      const Hdr1 = () => {
        return <h1>Hello React</h1>;
      };

      root1.render(<Hdr1 />);
    </script>

    <script type="text/babel">
      const root2 = ReactDOM.createRoot(document.getElementById("root2"));

      const Container = () => {
        return (
          <div>
            <h4>The library for web and native user interfaces</h4>
            <p>
              React is a free and open-source front-end JavaScript library that
              aims to make building user interfaces based on components more
              seamless.
            </p>
          </div>
        );
      };

      root2.render(<Container />);
    </script>
  </body>
</html>
```

Key Improvements

1. **JSX Syntax**: HTML-like templating

```
// Transforms to:
// React.createElement("h1", null, "Hello React")
```

```
<h1>Hello React</h1>
```

2. **Component Model**: Reusable UI units

```
function MyComponent() {
  return <div>Content</div>;
}
```

3. **Readability**: Visual hierarchy matches DOM structure

4. **Developer Experience**: Faster prototyping

JSX Under the Hood

Compilation Process

- Test on Babel website (with classic runtime).
  - https://babeljs.io/repl

```
// Original JSX
<div className="header">
  <h1>Title</h1>
  <p>Content</p>
</div>;

// Transpiled to:
React.createElement(
  "div",
  { className: "header" },
  // next arguments are variable arg list that is collected as array [...]
  React.createElement("h1", null, "Title"),
```

```
  React.createElement("p", null, "Content")
);
```

Why JSX?

1. **Visual Clarity**: Easier to understand nested structures
2. **Developer Productivity**: Faster than manual `createElement` calls
3. **Type Safety**: Combined with TypeScript for better tooling
4. **Component Composition**: Natural expression of UI hierarchy

Modern React Development

Standard Toolchains

1. **Create React App (CRA)** (deprecated)

```
npx create-react-app my-app
```

2. **Vite** (recommended)

```
npm create vite@latest my-react-app --template react
```

3. **Next.js** (for SSR)

```
npx create-next-app@latest
```

Key Evolution Milestones

1. **Class Components (2013-2018)**

```
class MyComponent extends React.Component {
  render() {
    return <div>Hello</div>;
  }
}
```

2. **Functional Components + Hooks (2018+)**

```
function MyComponent() {
  const [state, setState] = useState();
  return <div>Hello</div>;
}
```

3. **Server Components (2023+)**

```
// Next.js example
async function ServerComponent() {
  const data = await fetchData();
  return <div>{data}</div>;
}
```

Comparison Table

Aspect	Pure React API	JSX + Babel	Modern Toolchains
Syntax	`React.createElement()`	HTML-like JSX	JSX with TypeScript
Readability	Low	High	Highest

Aspect	Pure React API	JSX + Babel	Modern Toolchains
Setup	Script tags	Babel standalone	Bundlers (Webpack/Vite)
Components	Not native	Basic functions	Hooks, Server Components
Tooling	None	Basic transpilation	Full dev environment

React Project using Vite

Step-by-Step Process

```
# Initiate project creation
cmd> npm create vite@latest
# OR > yarn create vite myapp

# Follow prompts:
| ◇ Project name: hello-world
| ◇ Select framework: React
| ◇ Select variant: JavaScript

# After creation:
cmd> cd hello-world
cmd> npm install    # Install dependencies
# OR > yarn install
cmd> npm run dev    # Start development server
# OR > yarn run dev
```

Key Features of Vite

- **Vite**: is an application manager to create react app and manage (execute, build, ...) it.
- **Instant Server Start**: Uses native ES modules (no bundling during dev)
- **Lightning Fast HMR**: Hot Module Replacement (<100ms updates)

- **Optimized Build**: Rollup-based production bundling
- **Framework Agnostic**: Supports React, Vue, Svelte, etc.

Project Structure

```
hello-world/
├── node_modules/     # After npm/yarn install
├── public/           # Static assets
├── src/
│   ├── assets/       # Images/SVGs
│   ├── App.css       # Component styles
│   ├── App.jsx       # Main component
│   ├── main.jsx      # Entry point
│   └── index.css     # Global styles
├── index.html        # Root HTML
└── vite.config.js    # Build configuration
```

- `node_modules`
  - contains all the dependencies
  - don't modify this folder
  - gets created when you run the command: `npm install` or `yarn`
- `public`
  - contains the static files
  - used to store the images, fonts, audio, video files etc.
- `src`
  - `assets`
    - contains the assets (resources) like images, audio, video files
  - `App.css`
    - contains the css rules for the App component
  - `App.jsx`
    - contains the startup component named App

        - when the application starts, it loads this component
    - `index.css`
        - contains the global css rules
        - all the css rules which can be shared across all the components
    - `main.jsx`
        - contains the startup function to load the first component
    - `screens` or `pages`
        - user created folder that contains the components which represent the pages
    - `services`
        - user created folder that contains the functions which are used to connect to the backend
    - `components`
        - user created folder that contains the reusable components
        - these components are shared across the pages
- `.gitignore`
    - used to ignore the files and folders which are not required to be pushed to the git repository
- `eslint.config.js`
    - contains the configuration about the ES Lint program
    - linter is a program to find out the syntax errors
- `index.html`
    - the only html file (SPA) in the project
    - this file loads all the react components
    - this file contains a div with id root which is considered to be the host for all the react components
- `package.json`
    - contains the configuration about the react application
    - scripts
        - contains commands which can be used with npm or yarn
    - dependencies
        - contains list of modules which will be compiled and added to the deployment package
        - the packages mentioned here are required to run the application
    - devDependencies
        - contains a list of modules which will be needed to develop the application

        - these modules will NOT be compiled into the deployable package
        - these packages are NOT required to run the application
- `vite.config.js`
    - configuration file for vite application package manager

Core Files Explanation

1. index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <!-- React root element -->
    <div id="root"></div>
    <!-- Entry point for React -->
    <script type="module" src="/src/main.jsx"></script>
  </head>
</html>
```

- **Key Point**: Vite handles JSX via native ES modules (`type="module"`)

2. main.jsx (Entry Point)

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import App from "./App.jsx";

createRoot(document.getElementById("root")).render(
  <StrictMode>
    <App />
  </StrictMode>
);
```

- **StrictMode**: Development tool for highlighting potential issues. It activates additional checks and warnings for its descendants, helping developers identify and address issues early in the development process. Strict Mode does not render any visible UI and has no impact on the production build. StrictMode helps with:
  - **Identifying components with unsafe lifecycles:** It warns about the usage of lifecycle methods that are prone to misuse or will be deprecated in future versions of React.
  - **Warning about legacy string ref API usage:** It helps identify and update legacy string refs to the recommended callback or createRef API.
  - **Warning about deprecated findDOMNode usage:** It flags the use of findDOMNode, which can lead to unexpected behavior in modern React components.
  - **Detecting unexpected side effects:** By intentionally invoking certain functions twice, it helps uncover components with impure rendering logic or unintended side effects.
  - **Detecting legacy context API:** It warns about the usage of the old context API, encouraging migration to the modern context API.
  - **Ensuring reusable state:** It can help identify issues related to state management and promote best practices for reusable state logic.
- **createRoot**: React 18+ concurrent rendering API.

3. App.jsx (Main Component)

```jsx
function App() {
  return (
    <div>
      <h1>Hello World!</h1>
    </div>
  );
}
export default App;
```

- **JSX Syntax**: HTML-like syntax that compiles to `React.createElement`
- **Component Convention**: PascalCase naming (`App` vs `app`)

Running the Application

Development Workflow

```
cmd> npm run dev

# Output:
  VITE v6.5.2  ready in 251 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
```

Production Build

```
cmd> npm run build

# Creates optimized files in /dist folder
# Deploy-ready static files (to be hosted in any web server like Apache, Tomcat, or serve as static files in express
application)
```

Application execution flow

- when the application starts (after `yarn dev`), vite starts a lite web server on port 5173
- web server loads the `index.html`
- the `index.html` file loads the `main.jsx` script
- in the `main.jsx`
    - finds the div having an id `root`
    - creates a root with the div for rendering the React components
    - loads the default/startup component named `App`
    - renders the `App` component into the div with 'root' id

Why Vite Over Create React App (CRA)?

Feature	Vite	CRA
Speed	Instant server start	Slow Webpack setup
HMR	<100ms updates	1-3s updates
Config	Pre-configured + flexible	Locked configuration
Modern	ES Modules native	Legacy module system
Bundle Size	Optimized output	Larger default bundles

React Core Concepts

1. Component Architecture

- Reusable Components

```
// Component = JavaScript function returning JSX
function Greeting() {
  return <h2>Welcome to React!</h2>;
}

// Component Composition
function App() {
  return (
    <div>
      <h1>Hello World</h1>
      <Greeting />
      <Greeting />
    </div>
  );
}
```

2. JSX Styling

- **Must return single root element**
- **Class → className**
- **Inline styles as objects**

```
<div className="container" style={{ color: "red" }}>
  <p>Styled content</p>
</div>
```

```
const style = {
    container: {
        color: "red"
    },
    content: {
        fontSize: "14px",
        color: "blue"
    }
}
<div className="container" style={style.container}>
    <p style={style.content}>Styled content</p>
</div>
```

- **Class vs className**

```
// Correct
<div className="book-card">...</div>
```

```
// Incorrect
<div class="book-card">...</div>
```

3. Embedding JavaScript in JSX/Data binding

- Using the variable value inside a html tag
- In react, it will be done using interpolation
    - used the {} brackets for loading the variables value inside the html tag

```
const myvar = 100
<h3>{myvar}</h3>
```

- To render an object, split the object into properties and use interpolation to render those properties

```
const jamesBond = {
  name: "James Bond",
  age: 65,
  address: "London",
  speciality: "Adventurous Spy",
};
```

```
// Variables in curly braces
<div>Name: {jamesBond.name}</div>
<div>Address: {jamesBond['address']}</div>

// Expressions work too
<div>Status: {age > 60 ? 'Veteran' : 'Active'}</div>
```

- to render an array, use the map function to iterate over the array and use interpolation to render the properties of the object

```
const heros = [
  {
    name: "Iron Man",
    age: 48,
    address: "Stark Tower, New York",
    speciality:
      "Genius-level intellect, Powered armor suit, Advanced technology",
  },
  {
    name: "Captain America",
    age: 105,
    address: "Brooklyn, New York",
    speciality: "Super strength, Enhanced agility, Vibranium shield combat",
  },
];

const MarvelHeros = () => {
  return (
    <div>
      {heros.map((hero) => (
        <div>
          <h3>Marvel Hero</h3>
          <div>Name: {hero.name}</div>
          <div>Address: {hero.address}</div>
          <div>Age: {hero.age}</div>
          <div>Speciality: {hero.speciality}</div>
        </div>
      ))}
    </div>
  );
};
export default MarvelHeros;
```

4. JSX Syntax Gotchas

Self-Closing Tags

```
// Invalid
<img src="..." alt="..." width="60px">

// Always required for void elements
<img src="..." alt="..." width="60px" />
```

Root Element Requirement

```
// Invalid
return (
  <h3>Title</h3>
  <p>Content</p>
)

// Valid
return (
  <div>
    <h3>Title</h3>
    <p>Content</p>
  </div>
)

// Valid (Non rendered - parent tag)
return (
  <>
    <h3>Title</h3>
    <p>Content</p>
```

```
    </>
)
```

5. Component Design

- Component is reusable entity which contains user interface defined in html code
- A component can be loaded using the component name as a tag (enclosed by < and >)
- Types
  - Functional component
    - Component created using a function
    - Before react 16, functional components were used only for stateless implementation (the component which does not require to maintain the state)
    - Since the react 16 introduced a concept called as a react hooks, it is possible now to create functional components to store the state
    - Hence the class components are not need anymore and by default we use a function to create component
    - A javascript function which returns a JSX code to create its user interface
  - Class component
    - Component create using a class
    - Before react 16, class components were used to create stateful components (a component which can maintain its state)

Single Responsibility Principle

```
// Good: Focused component
function BookPrice({ price }) {
  return <div>Price: Rs. {price}/-</div>;
}

// Bad: Mixed responsibilities
function BookDetails({ book }) {
  // Handles price, image, author...
}
```

Component Event Handling

- Components should handle its events to make the component easily reusable.
- Event handling in React follows **camelCase** naming convention (`onClick` instead of `onclick`)
- Uses **Synthetic Events** - React's cross-browser wrapper around native DOM events
- Event handlers receive a **SyntheticEvent** object containing event details

Basic Event Handling

```
function App() {
  // 1. Define handler function
  const handleButtonClick = () => {
    alert("Button clicked!");
  };

  return (
    <div>
      {/* 2. Pass function reference (no parentheses!) */}
      <button onClick={handleButtonClick}>Click Me</button>
    </div>
  );
}
```

- ☑ **Pass function reference** - `onClick={handleClick}`
- ❌ **Avoid function call** - `onClick={handleClick()}` would execute immediately
- **SyntheticEvent** provides consistent interface across browsers

Handle user inputs

```
function App() {
  const handleInputChange = (event) => {
```

```
    // Access input value
    const userInput = event.target.value;
    console.log(`User input: ${userInput}`); // Correct string interpolation
  };

  return (
    <div>
      <label>Username:</label>
      <input
        type="text"
        onChange={handleInputChange}
        placeholder="Type something..."
      />
    </div>
  );
}
```

- Use `event.target.value` for input elements
- For forms, prefer **controlled components** (we'll cover later)
- Always use proper string interpolation with backticks: `` `Value: ${value}` ``

Common Event Types

Event	React Prop	DOM Equivalent
Click	`onClick`	`click`
Input Change	`onChange`	`input/change`
Form Submission	`onSubmit`	`submit`
Mouse Enter	`onMouseEnter`	`mouseenter`
Key Press	`onKeyDown`	`keydown`

Event Handler Best Practices

1. **Naming Convention**: Use `handle[Element][Event]` pattern

```
handleButtonClick, handleInputChange;
```

2. **Inline Handlers** (for simple logic):

```
<button onClick={() => console.log("Clicked")}>Log Click</button>
```

3. **Parameter Passing**:

```
<button onClick={(e) => handleClick(id, e)}>Delete Item</button>
```

SyntheticEvent Details

- **Properties**:

```
event.target; // DOM element that triggered event
event.currentTarget; // Element where handler is attached
event.type; // Event type ('click', 'change', etc.)
```

- **Methods**:

```
event.preventDefault(); // Cancel default behavior
event.stopPropagation(); // Stop event bubbling
```

Form submission Example

```
function FormComponent() {
  const handleSubmit = (event) => {
    event.preventDefault();
    console.log("Form submitted!");
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" onChange={(e) => console.log(e.target.value)} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Parent → Child Data Flow (props)

- "props" is an object containing all the properties sent by a parent component to a child component
- It is a readonly object (if child component modifies the props, the new values will NOT be available in the parent component)

```
// Person.jsx
export default function Person(props) {
  return (
    <div>
      <div>name: {props.name}</div>
      <div>address: {props.address}</div>
      <div>age: ${props.age}</div>
    </div>
  );
```

```
}

// App.jsx (in parent component)
<Person name="Nilesh Ghule" address="Pune" age="42" />;
```

```
// Object destructuring can make props more readable
// Person.jsx
export default function Person({ name, address, age }) {
  return (
    <div>
      <div>name: {name}</div>
      <div>address: {address}</div>
      <div>age: ${age}</div>
    </div>
  );
}
```

React Hooks

Core concept

- **Definition**: Special functions prefixed with use that let you "hook into" React features
- **Rules**:
    1. **Only Call Hooks at the Top Level**:
        - Never inside loops, conditions, or nested functions
        - Ensures consistent hook order between renders
    2. **Only Call Hooks from React Functions**:
        - Functional components
        - Other custom hooks

```

```

```
function Login() {
  // ☑ Correct - Top level of component
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");

  const onLogin = () => {
    // ❌ Wrong - Inside nested function
    // const [test, setTest] = useState('')
  };
}
```

Core React System Hooks

State Management Hooks

Hook	Purpose	Key Characteristics
`useState()`	Creates component state	Returns `[state, setState]` pair
`useReducer()`	Manages complex state logic	Alternative to `useState` for complex state objects
`useContext()`	Accesses shared context	Avoids prop drilling through component tree

Performance Optimization Hooks

Hook	Purpose	Key Characteristics
`useMemo()`	Memoizes expensive computations	Caches calculated values between renders
`useCallback()`	Memoizes callback functions	Prevents unnecessary re-renders of child components

Side Effect & DOM Hooks

Hook	Purpose	Key Characteristics
`useEffect()`	Handles side effects	Replaces lifecycle methods (`componentDidMount`, `componentDidUpdate`)
`useRef()`	Accesses DOM elements directly	Creates mutable object that persists across renders

React Router Hooks

Hook	Purpose	Usage Example
`useNavigate()`	Programmatic navigation	`const navigate = useNavigate(); navigate('/home');`
`useLocation()`	Accesses current route info	`const location = useLocation(); location.pathname`
`useParams()`	Reads route parameters	`const { id } = useParams();`

Redux Toolkit Hooks

Hook	Purpose	Key Characteristics
`useDispatch()`	Gets Redux store dispatcher	`const dispatch = useDispatch(); dispatch(action());`
`useSelector()`	Reads from Redux store	`const user = useSelector(state => state.auth.user);`

When to Use Each Hook

Scenario	Recommended Hook
Form input state	`useState`
Global theme settings	`useContext`
API data fetching	`useEffect` + `useState`
Complex state logic	`useReducer`
Optimizing expensive calculations	`useMemo`

Scenario	Recommended Hook
Preventing child re-renders	`useCallback`
Managing focus/animations	`useRef`
Routing navigation	`useNavigate`
Redux state management	`useSelector` + `useDispatch`

Best Practices

1. **Follow Rules of Hooks**:
    - Only call hooks at the top level
    - Only call from React functions
2. **Organize hooks logically**:
    - State hooks first
    - Effects next
    - Context/refs last
3. **Clean up effects**:
    - Always return cleanup functions from `useEffect` when needed

`useState()` Hook

Core Concept

- Adds reactive state to functional components
- Returns a stateful value and a function to update it

Syntax

```
const [state, setState] = useState(initialValue);
```

Part	Purpose
`state`	Current state value (read-only)
`setState`	Updater function (triggers re-render)
`initialValue`	Initial state (can be any type - number, string, object, etc.)

Usage Example

```
function Counter() {
  const [count, setCount] = useState(0); // Initialize with 0

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

Key Characteristics

1. **Asynchronous Updates**:

```
console.log(count); // May not reflect latest value immediately after setCount
```

2. **Functional Updates** (for derived state):

```
setCount((prevCount) => prevCount + 1);
```

3. **Object State**:
   - The modified state must be new JS object (or primitive value) - at new memory location - so that React sense the change in the state and re-render the component.
   - Modifying fields in existing object will NOT be considered as state update and will NOT trigger re-rendering of the component.

```
const [user, setUser] = useState({ name: "", age: 0 });
// Update with spread operator
setUser((prev) => ({ ...prev, name: "John" }));
```

Form Handling Example

- Note that, modified state must be a new object (not the changes in existing object). This example creates a new state object, copies old state using spread operator and append/update the modified field.

```
function LoginForm() {
  const [formData, setFormData] = useState({
    email: "",
    password: "",
  });

  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value,
    });
  };
```

```
  return <input name="email" value={formData.email} onChange={handleChange} />;
}
```

Important Points

- **Initialization**: The argument to `useState` is only used during the first render
- **Batching**: Multiple state updates may be batched together
- **Immutable Updates**: Always create new objects/arrays when updating state
- **Lazy Initialization**: For expensive initial state

```
const [data] = useState(() => computeExpensiveValue());
```

Props Drilling

Definition

Props drilling occurs when data is passed through multiple layers of components, even when intermediate components don't need that data.

Key Characteristics

- **Top-down data flow**: From parent → child → grandchild
- **Intermediate components** act as "pass-through" components
- **Problem**: Creates unnecessary dependencies between components

Example

```
function Parent() {
  const [count, setCount] = useState(10);

  return (
    <>
```

```
      <h2>Parent: count = {count}</h2>
      <Child count={count} setCount={setCount} />
    </>
  );
}

function Child({ count, setCount }) {
  return (
    <>
      <h2>Child Component</h2>
      <GrandChild count={count} setCount={setCount} />
    </>
  );
}

function GrandChild({ count, setCount }) {
  return (
    <>
      <h2>GrandChild: count = {count}</h2>
      <button onClick={() => setCount(count + 1)}>Update</button>
    </>
  );
}
```

Problems in This Implementation

1. **Child component** doesn't use `count` or `setCount` but must accept and pass them
2. **Tight coupling**: Changing the state structure requires modifying all components
3. **Reduced maintainability**: Harder to track data flow through multiple layers

Solutions to Props Drilling

1. Context API

- We will discuss this later

2. State management libraries

- For more complex requirements/objects libraries like redux are used for state management
- We will discuss this later

3. Component Composition

```
function Parent() {
  const [count, setCount] = useState(10);

  return (
    <>
      <h2>Parent: count = {count}</h2>
      <Child>
        <GrandChild count={count} setCount={setCount} />
      </Child>
    </>
  );
}

function Child({ children }) {
  return (
    <>
      <h2>Child Component</h2>
      {children}
    </>
  );
}
```

When to Use Each Solution?

Solution	Best For	Pros	Cons
Context API	Global state (theme, auth)	Clean component tree	Overhead for simple cases
Component Composition	Local component hierarchies	No extra dependencies	Doesn't scale for deep nesting
State Management Libraries (Redux, Zustand)	Complex applications	Centralized state	Higher learning curve

Best Practices

1. **Avoid premature optimization**: Simple prop passing is fine for 2-3 levels
2. **Use Context judiciously**: Only for truly global data
3. **Consider component structure**: Flatten components when possible
4. **Document data flow**: Especially in large codebases

External packages

- Find popular react libraries here: https://github.com/brillout/awesome-react-components

1. react-toastify

- https://www.npmjs.com/package/react-toastify
- **step 1: install react-toastify**

```
yarn add react-toastify
```

- **step 2: example**

```
import React from "react";

import { ToastContainer, toast } from "react-toastify";

function App() {
  const notify = () => toast("Wow so easy!");
```

```
  return (
    <div>
      <button onClick={notify}>Notify!</button>
      <ToastContainer />
    </div>
  );
}
```

- **Refer documention for more options/styles**

2. bootstrap

- **step 1: Install bootstrap**

```
yarn add bootstrap
```

- **step 2: Add bootstrap css & js in root component (main.jsx or App.jsx)**

```
import "bootstrap/dist/css/bootstrap.min.css";
import "bootstrap/dist/js/bootstrap.bundle.js";
```

- **step 3: Start using bootstrap classes into all components**

```
function App() {
  return (
    <div className="container">
      <div className="row justify-content-center">
        <h1>Hello Bootstrap</h1>
      </div>
    </div>
```

```
    );
}
```

3. axios

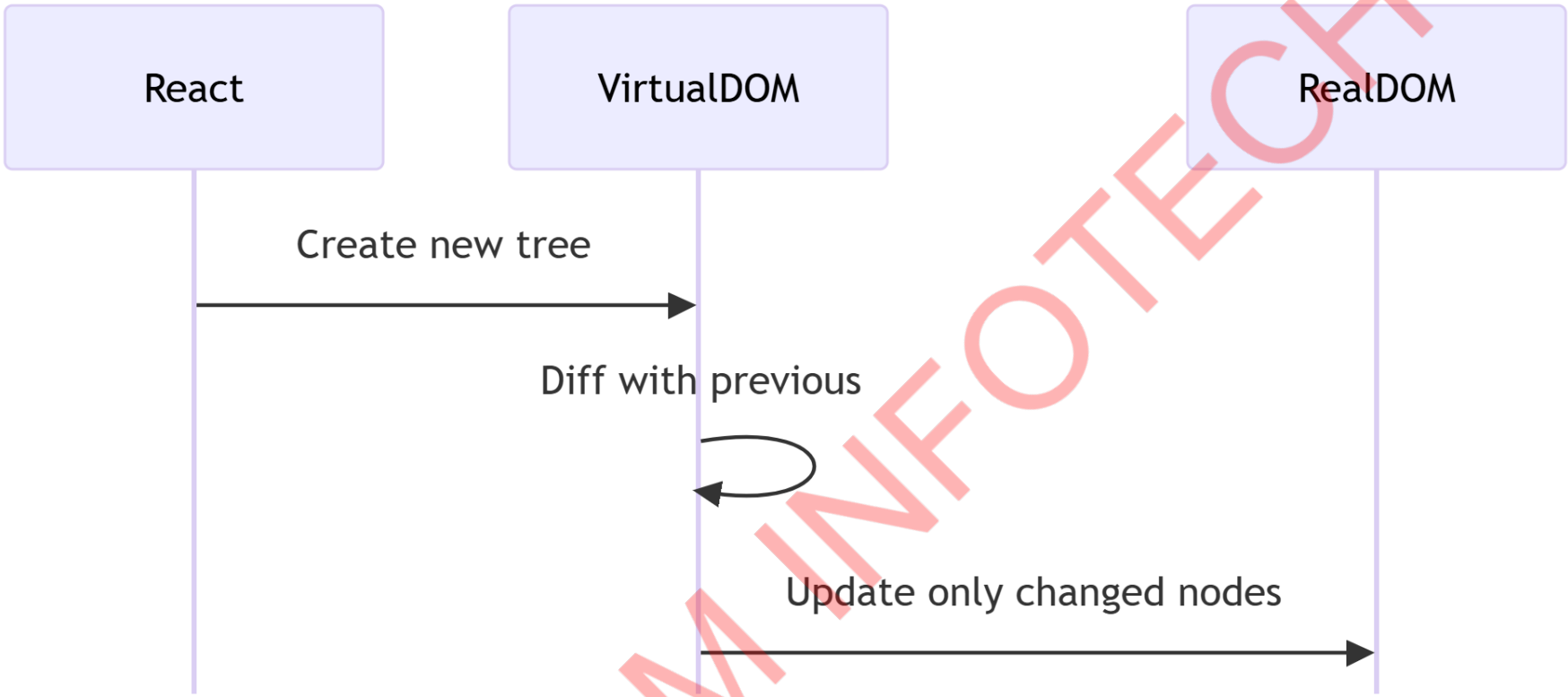
- Discussed later

React Core Concepts

6. Virtual DOM in Action

1. Initial render creates Virtual DOM tree
2. Subsequent changes create new Virtual DOM
3. React diffs trees and updates real DOM efficiently

How It Works with Components



Optimization Example

```
// Initial Render
<Book title="Chhava" price={450} />

// Price Update (Only changes price node)
<Book title="Chhava" price={480} />
```

Use Key Prop for Lists

```
// Essential for efficient list updates
{
  books.map((book) => <Book key={book.srno} {...book} />);
}
```

Reading

- https://www.geeksforgeeks.org/reactjs-reconciliation/

React `useEffect()` Hook

Core Concepts

- **Purpose**: Handles side effects (component lifecycle) in functional components
- **Replaces**: Class component lifecycle methods (`componentDidMount`, `componentDidUpdate`, `componentWillUnmount`)
- **Behavior**: Runs after render is committed to the screen

Syntax & Parameters

```
useEffect(() => {
  // Side effect logic
  return () => {
    /* Cleanup function */
  };
}, [dependencies]);
```

Parameter	Description	Required
Effect function	Contains side effect logic	Yes
Dependencies array	Controls when effect runs	No (but recommended)

Lifecycle Stages with `useEffect`

1. Component Mount (`componentDidMount`)

```
useEffect(() => {
  console.log("Component mounted - runs once");
  // API calls, subscriptions setup
}, []); // Empty dependency array
```

2. Component Unmount (`componentWillUnmount`)

```
useEffect(() => {
  console.log("Component mounted");

  return () => {
    console.log("Component unmounted - cleanup");
    // Cancel subscriptions, timers, etc.
  };
}, []);
```

3. State/Props Update (`componentDidUpdate`)

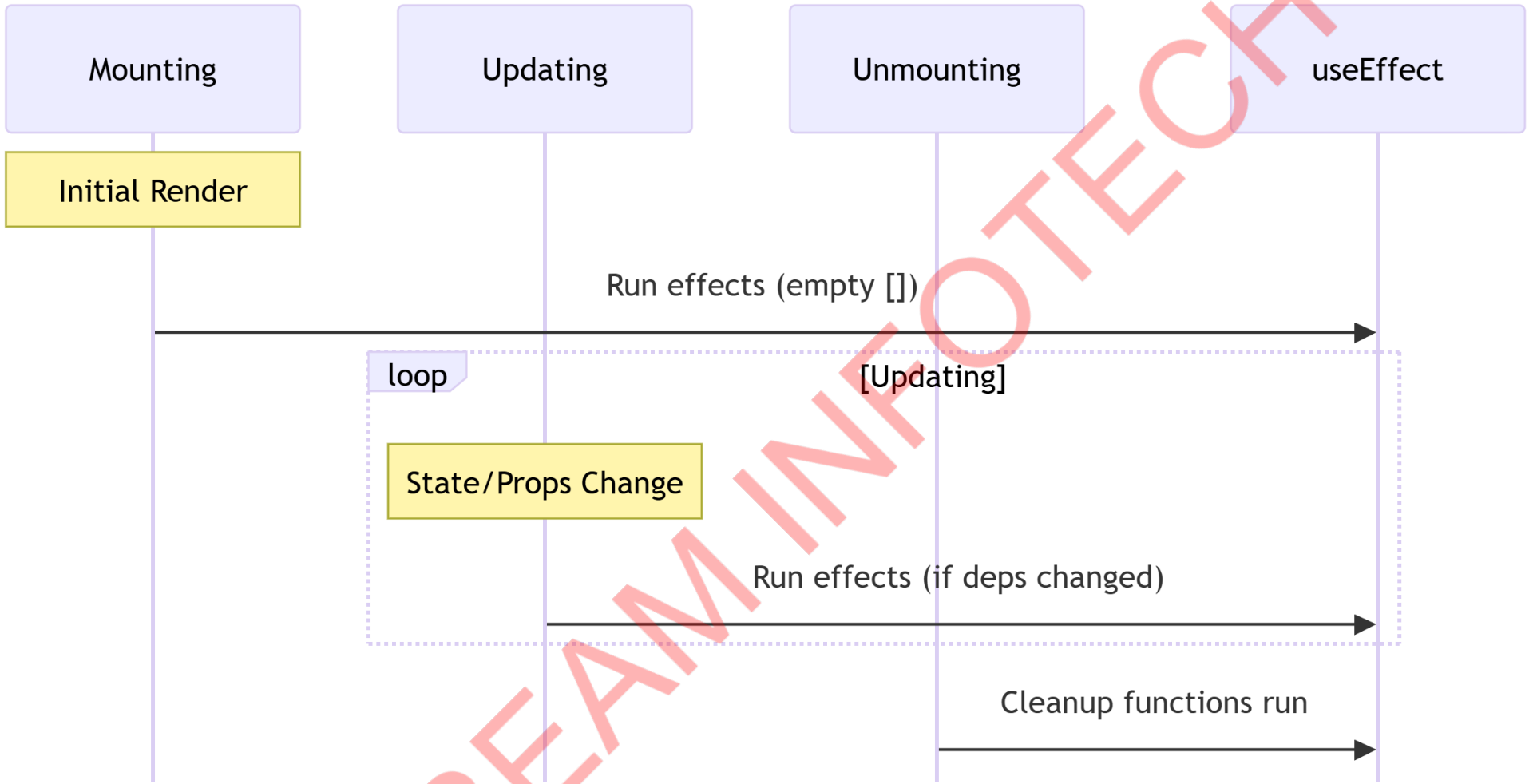
```
// Runs after every render (no dependency array)
useEffect(() => {
  console.log("Runs on mount AND every update");
});
```

4. Conditional Updates (Dependent on State/Props)

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

// Only runs when `count` changes
useEffect(() => {
  console.log(`Count changed to: ${count}`);
}, [count]); // Specific dependency
```

Diagram



Dependency Array (arg2) Behavior

Dependency Array	Behavior
`[]`	Runs once on mount (mount/unmount only)
`[dep1, dep2]`	Runs when any dependency changes

Dependency Array	Behavior
No array	Runs after every render (not recommended)

Multiple Effects Separation example

```
function Component() {
  // Separate concerns into different effects
  useEffect(() => {
    /* API call */
  }, []);
  useEffect(() => {
    /* Event listener */
  }, []);
  useEffect(() => {
    /* Analytics */
  });
}
```

Best Practices

1. **Always include dependencies** that your effect uses
2. **Clean up resources** to prevent memory leaks
3. **Split complex effects** into multiple `useEffect` calls
4. **Avoid infinite loops** by ensuring dependencies don't cause re-renders

Common Pitfalls

```
// 🚫 Infinite loop (missing dependency)
useEffect(() => {
  setCount(count + 1);
});
```

```
// 🚫 Missing cleanup
useEffect(() => {
  window.addEventListener("resize", handler);
});

// ☑️ Correct usage
useEffect(() => {
  const handler = () => {};
  window.addEventListener("resize", handler);
  return () => window.removeEventListener("resize", handler);
}, []);
```

React Component Lifecycle

Functional Counter Component with Hooks

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

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

  // Equivalent to componentDidMount
  useEffect(() => {
    console.log("Component mounted");
    return () => console.log("Component unmounted");
  }, []);

  // Equivalent to componentDidUpdate (count changes)
  useEffect(() => {
    console.log(`Count updated to: ${count}`);
  }, [count]);

  return (
```

```
    <div>
      <h2>Functional Counter: {count}</h2>
      <button onClick={() => setCount((c) => c + 1)}>Increment</button>
    </div>
  );
}
```

Class Counter Component Equivalent

```
import React from "react";

class ClassCounter extends React.Component {
  state = { count: 0 };

  // Mounting phase
  componentDidMount() {
    console.log("Component mounted");
  }

  // Updating phase
  componentDidUpdate(prevProps, prevState) {
    if (prevState.count !== this.state.count) {
      console.log(`Count updated to: ${this.state.count}`);
    }
  }

  // Unmounting phase
  componentWillUnmount() {
    console.log("Component unmounted");
  }

  render() {
    return (
      <div>
```

```
        <h2>Class Counter: {this.state.count}</h2>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Increment
        </button>
      </div>
    );
  }
}
```

Functional and Class Component Lifecycle Comparison

Lifecycle Stage	Functional Component	Class Component
Initialization	`useState()`	`constructor()`/`state = {}`
Mount	`useEffect(() => {}, [])`	`componentDidMount()`
Update	`useEffect(() => {}, [deps])`	`componentDidUpdate()`
Unmount	`useEffect(() => { return cleanup }, [])`	`componentWillUnmount()`
Render	Return JSX directly	`render()` method

Key Differences

Functional Components (with Hooks)

- **Simpler syntax**: No `this` keyword
- **Combined lifecycle**: Single `useEffect` hook handles all stages
- **Fine-grained control**: Multiple effects can be separated
- **No instance**: Better for optimization

Class Components

- **Explicit methods**: Separate methods for each lifecycle

- `this` **context**: Requires binding for event handlers
- **Legacy code**: Still found in older codebases
- **Error boundaries**: Can implement `componentDidCatch`

Parent Component - Controlling Counter Rendering

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

export default function LifecycleDemo() {
  const [showCounter, setShowCounter] = useState(true);

  return (
    <div>
      <h1>Lifecycle Demo</h1>
      <button onClick={() => setShowCounter(!showCounter)}>
        Toggle Counter
      </button>

      {showCounter && (
        <div style={{ display: "flex", gap: "2rem", marginTop: "2rem" }}>
          <FunctionalCounter />
          <ClassCounter />
        </div>
      )}
    </div>
  );
}
```

Here's your refined and expanded set of notes on **React Router** with improved structure, grammar, and technical clarity while following your requested sequence:

React Router

React Router enables **client-side routing** in React applications, allowing navigation between views without page reloads.

Installation

- Install the package via:

```
npm install react-router
# or
yarn add react-router
```

Using `BrowserRouter`, `Routes` and `Route`

Step 1: Wrap the App with `BrowserRouter`

- `BrowserRouter` syncs UI with the browser's URL using the HTML5 History API.
- Configure the router at the entry point (e.g., `main.jsx`):

```
import { BrowserRouter } from "react-router";

createRoot(document.getElementById("root")).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
);
```

Step 2: Define Routes in `App.jsx`

- Use `Routes` and `Route` to map paths to components:

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

function App() {
```

```
  return (
    <div>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/login" element={<LoginForm />} />
        <Route path="/register" element={<RegistrationForm />} />
        <Route path="/mytasks" element={<TaskList />} />
        <Route path="/newtask" element={<TaskForm />} />
      </Routes>
    </div>
  );
}
```

Key Notes:

- `Routes` acts as a container for all `Route` definitions.
- Each `Route` maps a `path` to a React component (`element`).
- Paths should start with `/` (e.g., `/login` instead of `login`) for consistency.

Navigation (`Link` vs `useNavigate`)

Static Navigation with `Link`

- Use for **declarative navigation** (e.g., menus, headers).
- Prevents full page reloads (SPA behavior).

```
import { Link } from "react-router";

function Navbar() {
  return (
    <nav>
      <Link to="/login">Login</Link>
      <Link to="/register">Register</Link>
```

```
      </nav>
    );
}
```

Dynamic Navigation with `useNavigate`

- Use for **programmatic navigation** (e.g., after form submission).
- `navigate(path, options)` accepts:
  - `path` (string): Target route.
  - `options` (optional): State, `replace` flag, etc.

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

function Register() {
  const navigate = useNavigate();

  const onRegister = () => {
    navigate("/login", { state: { from: "register" } }); // Optional metadata
  };

  return <button onClick={onRegister}>Register</button>;
}
```

- **navigate() options:**
  - `state` can pass data between routes (accessed via `useLocation` in next component).
  - `replace: true` replaces the current history entry instead of pushing.

Nested Routes

- Nested routes allow hierarchical UI layouts (e.g., dashboard with sub-views).

Example: User Home with Sub-Routes

```
<Routes>
  <Route path="/user" element={<UserLayout />}>
    <Route path="task-list" element={<TaskList />} /> // Matches
    `/user/task-list`
    <Route path="add-task" element={<AddTask />} /> // Matches `/user/add-task`
  </Route>
</Routes>
```

- Noe: Paths are **relative** (no leading `/`) inside nested routes.
- Parent route (`UserLayout`) must render an `<Outlet />` to display nested content (child route):

```
function UserLayout() {
  // get userName from auth info
  return (
    <div>
      <h1>User Layout (Show Navbar here)</h1>
      <h4>Hello, {userName}</h4>
      <Outlet /> {/* Renders child routes here */}
    </div>
  );
}
```

Advanced Hooks

`useParams()`

- Access dynamic URL parameters (e.g., `id` in `/tasks/:id`).

```
<Route path="/tasks/:id" element={<TaskDetail />} />;

// Inside TaskDetail.jsx:
function TaskDetail() {
  const { id } = useParams(); // Extracts `id` from URL
  return <div>Task ID: {id}</div>;
}
```

`useLocation()`

- Access current URL and state passed via `navigate()`.

```
function Login() {
  const location = useLocation();
  const from = location.state?.from || "/"; // Default to home
  return <div>Redirected from: {from}</div>;
}
```

Special uses

404 Handling: Add a catch-all route:

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

Index Routes: Render a default child route:

```
<Route path="/home" element={<Home />}>
  <Route index element={<Dashboard />} /> // Renders at `/home`
```

```
</Route>
```

Context API in React

- Context API provides a way to share data (state, functions) across components without manually passing props at every level.

Core Concepts

- `createContext()`: Creates a context object.
- `Provider`: Supplies the context value to its children.
- `useContext()`: Accesses the context value in child components.

Basic Example: Sharing a Counter

Parent Component (Creates Context)

```
import { createContext, useState } from "react";

// Step 1: Create Context
export const CounterContext = createContext();

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

    // Step 2: Wrap children with Provider
    return (
        <div>
            <CounterContext.Provider value={{ count, setCount }}>
                <Child /> {/* context provided to <Child/> and its all children ...*/}
            </CounterContext.Provider>

            <AnotherChild/> {/* context is NOT provided to <AnotherChild/>*/}
        <div>
```

```
    );
}
```

Grandchild Component (Modifies State)

```
import { useContext } from "react";
import { CounterContext } from "./Parent";

function Grandchild() {
  // Step 3: Consume Context
  const { count, setCount } = useContext(CounterContext);

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

- **Note:**
  - The `count` state is modified in `Grandchild` without prop drilling.
  - All components under `Provider` can access the context.

Auth-Protected Routes Using Context

Step 1: Create Authenticaton context

```
// App.jsx
import { createContext, useState } from "react";
import { BrowserRouter, Routes, Route } from "react-router";
```

```
export const AuthContext = createContext();

function App() {
  const [user, setUser] = useState(null);

  return (
    <AuthContext.Provider value={{ user, setUser }}>
      <BrowserRouter>
        <Routes>
          {/* Protected Route */}
          <Route path="/dashboard" element={user ? <Dashboard /> : <Login />} />

          {/* Public Routes */}
          <Route path="/login" element={<Login />} />
          <Route path="/" element={<Home />} />
        </Routes>
      </BrowserRouter>
    </AuthContext.Provider>
  );
}
```

Step 2: Login Component (Updates Context)

```
import { useContext } from "react";
import { AuthContext } from "./App";
import { useNavigate } from "react-router";

function Login() {
  const { setUser } = useContext(AuthContext);
  const navigate = useNavigate();

  const handleLogin = () => {
    setUser("Nilesh G"); // Update context
```

```
    navigate("/dashboard"); // Redirect
  };

  return <button onClick={handleLogin}>Login</button>;
}
```

Key Notes:

- Combines `Context` + `Router` for auth flows.
- `useNavigate` redirects after updating context.

Best Practices

1. **Avoid Overusing Context**
   - Use for **global** state (theme, auth, etc.).
   - For local state, prefer `useState` or `useReducer`.
2. **Optimize Performance**
   - Memoize context value to prevent unnecessary re-renders:

```
const value = useMemo(() => ({ user, setUser }), [user]);
return <AuthContext.Provider value={value}>...</AuthContext.Provider>;
```

3. **Multiple Contexts**
   - Split unrelated data into separate contexts (e.g., `ThemeContext`, `UserContext`).
4. **Default Values**
   - Provide sensible defaults when creating context:

```
const ThemeContext = createContext("light"); // Default: 'light'
```

When to Use Context vs Alternatives

Solution	Use Case
Context API	Medium-low frequency updates (theme, auth)
Redux	High-frequency updates (dashboards, real-time data)
Component State	Local component state (forms, UI toggles)

API Integration using Axios

- Axios is a popular HTTP client for making API requests (like fetch()).

Setup Axios

```
npm install axios
# or
yarn add axios
```

Basic API Requests

- Good practice to write API calls in separate module e.g. services/books.js.
- "export" all operations and consume them into react components.

GET Request (Fetch All Books)

```
import axios from "axios";

async function fetchBooks() {
  try {
    const response = await axios.get("http://localhost:3000/books");

    if (response.data.status === "success") {
      console.log("Books:", response.data.data);
      return response.data.data; // Array of books
    } else {
      console.error("Error:", response.data.message);
      throw new Error(response.data.message);
    }
  } catch (error) {
    console.error("API Error:", error.message);
    throw error;
  }
}
```

GET Request (Fetch Single Book)

```jsx
async function fetchBookById(bookId) {
  try {
    const response = await axios.get(`http://localhost:3000/books/${bookId}`);

    if (response.data.status === "success") {
      return response.data.data; // Single book object
    } else {
      throw new Error(response.data.message);
    }
  } catch (error) {
    console.error("Failed to fetch book:", error.message);
    throw error;
```

```
  }
}
```

POST Request (Add New Book with Image Upload)

- Using `FormData` to send body and file data ("multipart/form-data").

```
async function addNewBook(bookData, imageFile) {
  try {
    const formData = new FormData();
    formData.append("title", bookData.title);
    formData.append("author", bookData.author);
    formData.append("category", bookData.category);
    formData.append("price", bookData.price);
    formData.append("bookimg", imageFile); // File object

    const response = await axios.post("http://localhost:3000/books", formData, {
      headers: {
        "Content-Type": "multipart/form-data",
      },
    });

    if (response.data.status === "success") {
      console.log("Book added:", response.data.data);
      return response.data.data;
    } else {
      throw new Error(response.data.message);
    }
  } catch (error) {
    console.error("Failed to add book:", error.message);
    throw error;
  }
}
```

PUT Request (Update Book - JSON Body)

```
async function updateBook(bookId, updatedData) {
  try {
    const response = await axios.put(
      `http://localhost:3000/books/${bookId}`,
      updatedData // { title, author, category, price, image_name }
    );

    if (response.data.status === "success") {
      console.log("Updated book:", response.data.data);
      return response.data.data;
    } else {
      throw new Error(response.data.message);
    }
  } catch (error) {
    console.error("Failed to update book:", error.message);
    throw error;
  }
}
```

DELETE Request (Remove a Book)

```
async function deleteBook(bookId) {
  try {
    const response = await axios.delete(
      `http://localhost:3000/books/${bookId}`
    );

    if (response.data.status === "success") {
      console.log("Book deleted successfully");
```

```
      return true;
    } else {
      throw new Error(response.data.message);
    }
  } catch (error) {
    console.error("Failed to delete book:", error.message);
    throw error;
  }
}
```

Example Usage in React Component

```
import { useState, useEffect } from "react";
// import fetchBooks if written in separate module.

function BookList() {
  const [books, setBooks] = useState([]);

  useEffect(() => {
    async function loadBooks() {
      try {
        const data = await fetchBooks();
        setBooks(data);
      } catch (error) {
        alert("Failed to load books");
      }
    }
    loadBooks();
  }, []);

  return (
    <div>
      {books.map((book) => (
        <div key={book.id}>{book.title}</div>
```

```
        ))}
      </div>
    );
}
```

Best Practices

1. `axios.create()` – Configure a base URL and default headers.

```
// api.js
import axios from 'axios';

const API_BASE_URL = 'http://localhost:3000/books';

export const api = axios.create({
  baseURL: API_BASE_URL,
  headers: { 'Content-Type': 'application/json' },
});
```

```
// books.js

import {api} from './api';

async function fetchBookById(id) {
  try {
    const response = await api.get(`/${id}`);
    return response.data.data; // Single book object
  } catch (error) {
    console.error('Fetch failed:', error.response?.data?.message || error.message);
    return null;
```

```
  }
}
```

2. Interceptors – Globally handle requests/responses (e.g., adding auth tokens).

```
import axios from "axios";

const API = axios.create({
  baseURL: "http://localhost:3000/books", // Replace with your API endpoint
  timeout: 5000, // Request timeout
});

// Add a request interceptor
API.interceptors.request.use(
  function (config) {
    // Get authentication token from local storage
    const token = localStorage.getItem("token");

    // Add token to request headers if it exists
    if (token) {
      config.headers.Authorization = `Bearer ${token}`;
    }

    return config;
  },
  function (error) {
    // Handle request errors
    return Promise.reject(error);
  }
);
```

3. Error Handling Middleware – Centralize API error handling.

```
// utils/api.js
import axios from "axios";

const API = axios.create({
  baseURL: "http://localhost:3000/books", // Replace with your API endpoint
  timeout: 5000, // Request timeout
});

// Standardized response handling
API.interceptors.response.use(
  (response) => {
    if (response.data.status === "error") {
      return Promise.reject(response.data.message);
    }
    return response.data.data; // Directly return the `data` field for success
  },
  (error) => {
    return Promise.reject(error.response?.data?.message || "Network Error");
  }
);

export default API;
```

```
import API from "./utils/api";

async function fetchBooks() {
  try {
    const books = await API.get("/");
    console.log("Books:", books); // books = [{ id: 1, title: "Atlas Shrugged", ... }]
    return books;
  } catch (error) {
    console.error("GET Error:", error);
    throw error;
```

```
    }
  }
```

4. Cancellation – Use `AbortController` to cancel pending requests.

```
useEffect(() => {
  const abortController = new AbortController();
  const fetchData = async () => {
    try {
      const data = await API.get("/", { signal: abortController.signal });
      setBooks(data);
    } catch (err) {
      if (!abortController.signal.aborted) setError(err.message);
    }
  };
  fetchData();
  return () => abortController.abort();
}, []);
```

5. Environment Variables - Store API base URLs in .env

```
REACT_APP_API_BASE_URL=http://localhost:3000
```

- Access via process.env.REACT_APP_API_BASE_URL.