ReactJS Comprehensive Guide for Beginners

Introduction to ReactJS

What is React?

React is a **JavaScript library** used for building fast and interactive user interfaces, especially for **single-page applications (SPAs)**. It was developed by **Facebook (Meta)** and has become one of the most popular front-end frameworks.

Why Use React?

- **Component-Based Architecture** → Code reusability and modular development.
- **Virtual DOM (VDOM)** → Optimized UI rendering with minimal performance impact.
- **Unidirectional Data Flow** → Predictable and maintainable state management.
- **Hooks System** → Simplifies state and side-effects handling.
- **Large Ecosystem** → Strong community support and numerous third-party libraries.
- **Cross-Platform Development** → Can be used with React Native for mobile applications.

1. useEffect Hook

What is useEffect?

useEffect is a built-in **React Hook** used to handle **side effects** in functional components. It replaces lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount in class components.

When to Use useEffect?

- **Fetching Data from APIs**
- **Listening to Events (e.g., scroll, resize)**
- **Updating Document Title Dynamically**
- **Handling Subscriptions (WebSockets, Firebase, etc.)**
- **Running Timers (setTimeout, setInterval)**

Basic Syntax:

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

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

Example: Fetching Data with useEffect

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

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

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/posts")
      .then(response => response.json())
      .then(data => setData(data));

    return () => console.log("Cleanup executed!");
  }, []); // Empty dependency array → Runs only on mount

  return (
    <div>
      {data.map(post => (
        <p key={post.id}>{post.title}</p>
      ))}
    </div>
  );
}
```

Types of useEffect Usage:

1. **Runs on every render** → useEffect(() => {...})
2. **Runs only on mount** → useEffect(() => {...}, [])
3. **Runs when dependencies change** → useEffect(() => {...}, [dependency])

Common Mistakes & Best Practices:

- **Don't use useEffect for simple calculations** (use useMemo instead).
- **Always clean up subscriptions or event listeners** in the return function.
- **Minimize unnecessary re-renders** by correctly specifying dependencies.

---

2. React Hooks

What are Hooks?

Hooks allow **functional components** to use state and lifecycle methods **without writing class components**.

Why Use Hooks?

- **Simpler Code** → No need for complex class components.
- **Better Reusability** → Custom hooks allow logic sharing.
- **Improved Readability** → Cleaner syntax and better function composition.

Common Hooks and Their Usage:

Hook	**Purpose**
useState	Manages component state
useEffect	Handles side effects
useContext	Manages global state
useRef	Maintains references across renders
useMemo	Optimizes performance by memoizing calculations
useCallback	Memoizes functions to prevent re-renders

Example: useState Hook

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


function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

---

3. Redux (State Management)

What is Redux?

Redux is a **state management library** that helps manage global state efficiently across an application.

Key Concepts:

1. **Store** → Centralized place to store application state.
2. **Actions** → Events that trigger state changes.
3. **Reducers** → Functions that update state based on actions.
4. **Dispatch** → Sends an action to update the state.
5. **Selectors** → Retrieve specific data from the store.

Redux Flow:

1. Component **dispatches an action**.
2. Action is sent to the **reducer**.
3. Reducer updates the **store**.
4. Updated store sends new data to the component.

Example Implementation:

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

// Reducer function
const counterReducer = (state = { count: 0 }, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    default:
      return state;
  }
};

// Create Redux store
const store = createStore(counterReducer);

// Dispatch an action
store.dispatch({ type: "INCREMENT" });

console.log(store.getState()); // Output: { count: 1 }
```

---

4. React Virtualization

What is React Virtualization?

React Virtualization optimizes performance when displaying **large lists or tables** by **only rendering visible elements**.

Example Using react-window:

```
import { FixedSizeList } from "react-window";

const Row = ({ index, style }) => (
  <div style={style}>Row {index}</div>
);

function VirtualizedList() {
  return (
    <FixedSizeList
      height={400}
      width={300}
      itemSize={35}
      itemCount={1000}
    >
      {Row}
    </FixedSizeList>
  );
}
```

---

5. Reconciliation (How React Updates the DOM)

What is Reconciliation?

Reconciliation is the process React uses to update the **DOM efficiently**. React **compares the new Virtual DOM with the previous one** and updates only the changed parts instead of re-rendering everything.

Example: React Key Usage

```
function ItemList({ items }) {
  return (
    <ul>
      {items.map(item => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
}
```

---

6. Higher-Order Components (HOC)

What is a Higher-Order Component (HOC)?

A **Higher-Order Component (HOC)** is a **function** that takes a component and returns a **new enhanced component**. It is a pattern in React for **reusing component logic**.

Why Use HOC?

- **Code Reusability** → Extracts common logic into a reusable function.
- **Separation of Concerns** → Enhances components without modifying them.
- **Makes Components More Modular** → Easily add features like authentication, logging, etc.

Example of HOC

```
import React from "react";


// Higher-Order Component
const withLogging = (WrappedComponent) => {
  return (props) => {
    console.log("Component Rendered:", WrappedComponent.name);
    return <WrappedComponent {...props} />;
  };
};


// Normal Component
const Hello = ({ name }) => <h1>Hello, {name}!</h1>;


// Enhanced Component
const EnhancedHello = withLogging(Hello);


function App() {
  return <EnhancedHello name="React" />;
}


export default App;
```

Common Use Cases of HOC

- **Authentication Handling**
- **Permission Control**
- **Data Fetching**

- **Logging and Analytics**

Best Practices for HOC

- **Pass all props** using {...props} to avoid losing data.
- **Use meaningful names** for better readability.
- **Avoid nesting multiple HOCs**, as it can make debugging difficult.

---

7. Test Cases in React

Why is Testing Important?

Testing ensures your **React application works correctly** and prevents **bugs**.

Types of Testing in React

Test Type	Purpose
Unit Testing	Tests small components or functions individually.
Integration Testing	Tests interactions between components.
End-to-End Testing (E2E)	Tests the entire application flow.

Testing Libraries in React

1. **Jest** → Unit and integration testing.
2. **React Testing Library** → Simulates user interactions.
3. **Cypress** → End-to-end testing for full UI flows.

Example: Writing a Test with Jest & React Testing Library

```
import { render, screen } from "@testing-library/react";
import "@testing-library/jest-dom";
import Button from "./Button";


test("renders button with correct text", () => {
  render(<Button label="Click Me" />);
  const buttonElement = screen.getByText(/Click Me/i);
  expect(buttonElement).toBeInTheDocument();
});
```

Best Practices for Testing React Apps

- Test **critical functionalities** (e.g., forms, buttons).
- Use **mocking** to simulate API calls.
- Write tests **before adding new features** to ensure reliability.

---

8. React Strict Mode

What is Strict Mode?

Strict Mode is a tool in React that **highlights potential problems** in the application **during development**. It does not affect production builds.

Why Use Strict Mode?

- **Detects Unsafe Lifecycle Methods**
- **Identifies Side Effect Bugs**
- **Warns About Deprecated APIs**
- **Ensures Best Practices in Concurrent Mode**

Enabling Strict Mode in React

```
import React from "react";

import ReactDOM from "react-dom";

import App from "./App";


ReactDOM.createRoot(document.getElementById("root")).render(

  <React.StrictMode>

    <App />

  </React.StrictMode>

);
```

What Does Strict Mode Detect?

- **Unsafe use of useEffect** → Ensures side effects are handled correctly.
- **Legacy String Refs** → Encourages using useRef.
- **Finds Accidental State Mutations** → Helps maintain a predictable state.

Common Issues When Using Strict Mode

- **Components might render twice in development mode** (not in production).

- **Warnings for old APIs might appear even if you aren't using them** (usually due to third-party libraries).

Best Practices

- **Always use Strict Mode in new projects**.
- **Fix all warnings** shown in Strict Mode before going to production.
- **Use React Developer Tools** to debug warnings efficiently.

---

9. React Lifecycle Methods

What is a Component Lifecycle?

A React component goes through **various phases** from creation to removal.

Lifecycle Phases in React

Phase	Description
Mounting	Component is created and added to the DOM.
Updating	Component updates due to changes in state or props.
Unmounting	Component is removed from the DOM.

Lifecycle Methods in Class Components

Method	Purpose
constructor()	Initializes state.
componentDidMount()	Runs once when component mounts (fetch data, add event listeners).
componentDidUpdate()	Runs after re-render (update UI based on new props/state).
componentWillUnmount()	Cleanup before component is removed (remove event listeners, cancel API calls).

Example of Lifecycle Methods in Class Components

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


class LifecycleDemo extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  componentDidMount() {
    console.log("Component Mounted!");
  }

  componentDidUpdate() {
    console.log("Component Updated!");
  }

  componentWillUnmount() {
    console.log("Component Unmounted!");
  }

  render() {
    return (
      <div>
        <h2>Lifecycle Demo</h2>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Increment
        </button>
      </div>
    );
  }
}

export default LifecycleDemo;
```

Lifecycle Methods in Functional Components (Using Hooks)

React Hooks provide an alternative to lifecycle methods. The useEffect hook can be used to replace class-based lifecycle methods.

Class Component Method	**Functional Component Equivalent (Hook)**
componentDidMount	useEffect(() => {...}, [])
componentDidUpdate	useEffect(() => {...}, [dependencies])
componentWillUnmount	useEffect(() => { return cleanup }, [])

Example Using Hooks Instead of Lifecycle Methods

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


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


  useEffect(() => {
    console.log("Component Mounted!");


    return () => {
      console.log("Component Unmounted!");
    };
  }, []);


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

```
export default FunctionalLifecycle;
```

---

10. **React Advanced Concepts**

Tree Shaking

What is Tree Shaking?

Tree shaking is a **dead code elimination** technique that removes unused JavaScript code from the final bundle, reducing the file size and improving performance.

How Tree Shaking Works?

- **ES6 Module System (import/export)** → Only imports the necessary parts of a module.
- **Static Analysis** → Determines which code is actually used.
- **Minification Tools (Terser, UglifyJS)** → Remove unreachable code.
- **Bundlers (Webpack, Rollup, Parcel)** → Optimize the bundle by removing unused exports.

Example of Tree Shaking

```
// utils.js
export function add(a, b) {
  return a + b;
}

export function subtract(a, b) {
  return a - b;
}

// main.js
import { add } from "./utils";
console.log(add(2, 3));
```

In this case, the subtract function is **never used** and will be removed during the build process.

How to Enable Tree Shaking?

- Use **ES6 modules (import/export)** instead of CommonJS (require/module.exports).
- In Webpack, set mode: "production" and enable optimization.usedExports: true.
- In package.json, set "sideEffects": false to remove unused code.

---

11. Pure and Impure Components

Pure Components

A **Pure Component** in React is one that **renders the same output for the same state and props**. It prevents unnecessary re-renders by implementing **shouldComponentUpdate()** internally.

Characteristics of Pure Components

- **No Side Effects** → Does not modify external state.
- **Same Input → Same Output** → Always produces the same output for the same props/state.
- **Uses Shallow Comparison** → Checks if state or props have changed before re-rendering.

Example of a Pure Component

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


class PureComp extends PureComponent {
  render() {
    console.log("Rendered");
    return <h1>{this.props.message}</h1>;
  }
}


export default PureComp;
```

Here, PureComp will **only re-render** if the message prop changes.

Impure Components

An **Impure Component** does not guarantee the same output for the same input and may re-render unnecessarily.

Characteristics of Impure Components

- **Modifies State Directly**.
- **Uses External Variables**.
- **No Optimized Rendering**.

Example of an Impure Component

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

class ImpureComp extends Component {
  render() {
    console.log("Rendered");
    return <h1>{Math.random()}</h1>; // Different output each time
  }
}

export default ImpureComp;
```

This component re-renders **even if props/state do not change**.

---

12. Flux Concept

Flux is an **architecture pattern** for managing application state and data flow in React applications. It was introduced by Facebook to **ensure unidirectional data flow**.

Flux Architecture

Flux consists of **four** major components:

1. **Action** → Describes what happens (e.g., USER_LOGGED_IN).
2. **Dispatcher** → Central hub that sends actions to stores.
3. **Store** → Holds and updates state based on actions received.
4. **View** → UI components that display the updated state.

Flux Data Flow

1. **User Interacts with the UI** → Triggers an Action.
2. **Action is Dispatched** → Sent to the Dispatcher.
3. **Dispatcher Updates Store** → Store modifies state accordingly.

4. **View Updates** → React components update based on the new state.

Flux vs Redux

Feature	**Flux**	**Redux**
Store	Multiple Stores	Single Store
Dispatcher	Explicit Dispatcher	No Dispatcher (Uses Reducers)
State Updates	Event-based	Function-based
Ease of Use	More Complex	More Structured

Flux is **less commonly used** today, as Redux provides a more structured way to handle state.

---

13. Presentation Segment - Component with Only HTML

A **Presentation Component** (also called a Stateless or Dumb Component) is a component that **only renders UI** without managing state or logic.

Characteristics of Presentation Components

- **Only Receives Props** → Does not manage state.
- **No Business Logic** → Only focuses on rendering UI.
- **Reusable & Easy to Test** → Can be used in different places.

Example of a Presentation Component

```
import React from "react";


const Button = ({ label }) => {
  return <button className="btn">{label}</button>;
};


export default Button;
```

This button component **only displays** the label and does not handle any logic.

---

14. Synthetic Events in React

What is a Synthetic Event?

A **Synthetic Event** is a React wrapper around the browser's **native event system**. It ensures that events work consistently across all browsers.

Why Use Synthetic Events?

- **Cross-browser Compatibility**.
- **Event Pooling (Performance Optimization)**.
- **Unified API (Same event system for all elements).**

Example of a Synthetic Event

```
import React from "react";


function App() {
  const handleClick = (event) => {
    console.log("Button Clicked!", event);
  };


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


export default App;
```

Here, onClick uses React's **Synthetic Event** system, which wraps the browser's native click event.

Synthetic vs Native Events

Feature	**Synthetic Event**	**Native Event**
Cross-browser Support	Yes	No
Performance Optimized	Yes (Event Pooling)	No
Unified API	Yes	No (Different APIs for different browsers)

Note: In newer React versions (React 17+), event pooling is removed, making synthetic events behave more like native events.
