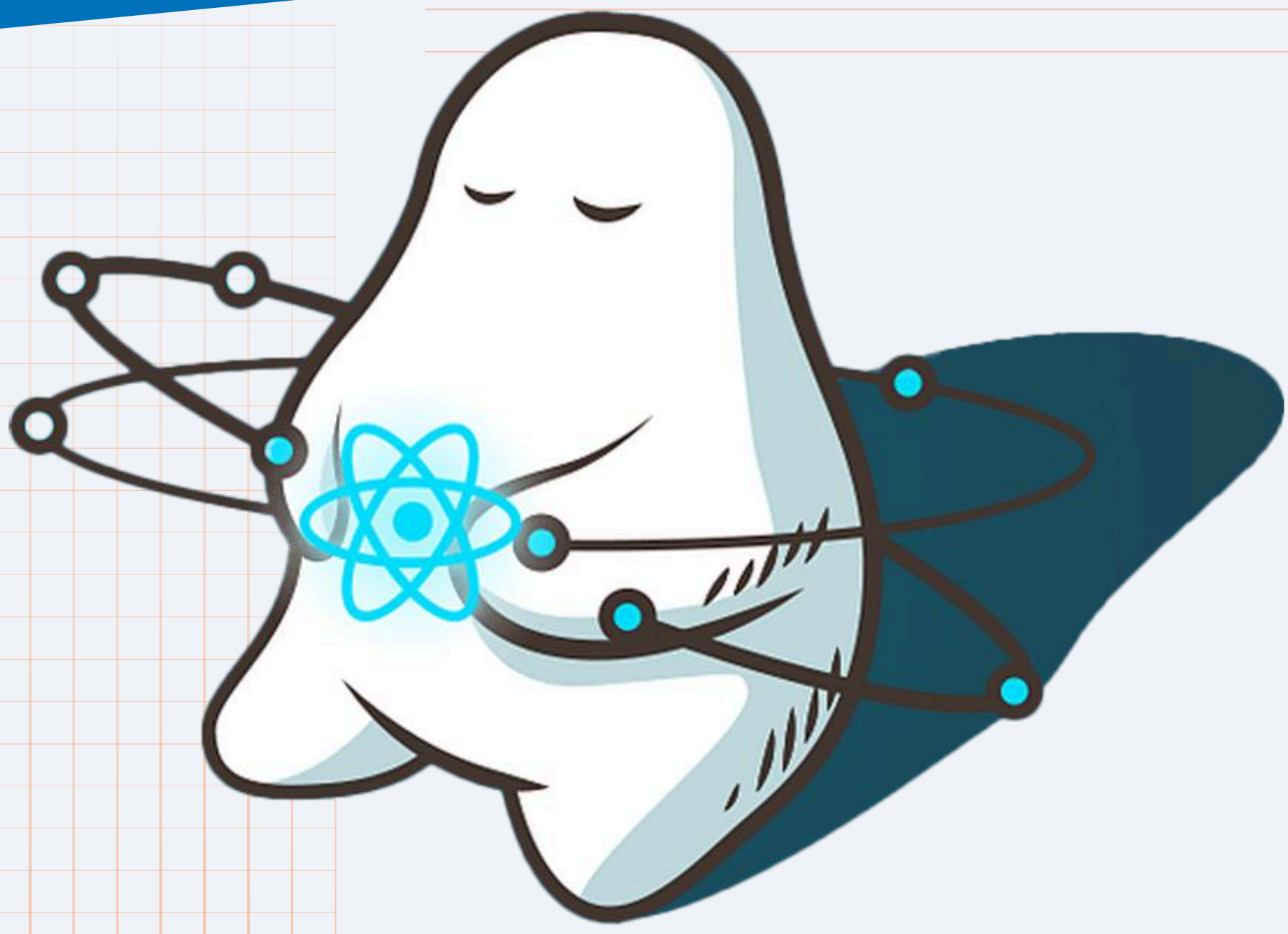
Learnbay

React Best Practices

Writing Clean and Efficient Code



www.learnbay.co

Introduction

React Best Practices

- Elevate your React applications by writing clean, efficient, and maintainable code.
- Master these practices to enhance performance and scalability.

Why It's Great: Streamlined codebases lead to faster development and easier maintenance.

Next →

Use Functional Components

Do: Prefer functional components with hooks.

Example:

```
// ✅ Functional Component with Hooks
const Counter = () => {
  const [count, setCount] = useState(0);
  return <button onClick={() => setCount(count + 1)}>Count: {count}</button>;
};
```

- Simpler Syntax: Easier to read and write.
- Hooks Advantage: Leverage powerful React hooks for state and side effects.
- Improved Testing: Functional components are easier to test

Next →

Learnbay

Don't: Rely on class-based components unnecessarily.

Example:

```
class Counter extends React.Component {
  state = { count: 0 };
  increment = () => this.setState({ count: this.state.count + 1 });
  render() {
    return <button onClick={this.increment}>Count: {this.state.count}</button>;
  }
}
```

- Verbose Syntax: More boilerplate code.
- Less Intuitive: Harder to grasp compared to functional components.
- Lifecycle Complexity: Managing lifecycle methods can be cumbersome.

Pro Tip : Functional components are simpler and easier to test.

Next →

Learnbay

Keep Components Small and Reusable

Do: Break down UI into small, reusable components.

Example:

```
// ✅ Reusable Button Component
const Button = ({ onClick, children }) => (
  <button onClick={onClick}>{children}</button>
);
```

- Enhanced Readability: Easier to understand each component's purpose.
- Reusability: Use the same component across different parts of your app.
- Maintainability: Simplifies updates and bug fixes

Next →

Learnbay

Don't: Create large, monolithic components.

Example:

```
// ❌ Large Component (Avoid)
const Dashboard = () => (
  <div>
    <header>Header</header>
    <main>
      <section>Section 1</section>
      <section>Section 2</section>
    </main>
    <footer>Footer</footer>
  </div>
);
```

- Hard to Manage: Difficult to navigate and maintain.
- Low Reusability: Components are tightly coupled and specific.
- Increased Complexity: Higher chance of bugs and performance issues.

Pro Tip : Small components enhance readability and reusability

Next →

Learnbay

Proper State Management

Do: Lift state up or use state management libraries when necessary.

Example:

```
// ✅ Lifting State Up
const Parent = () => {
  const [data, setData] = useState(null);
  return <Child data={data} updateData={setData} />;
};
```

- Single Source of Truth: Avoids state duplication.
- Easier Data Flow: Simplifies understanding how data moves.
- Better Synchronization: Ensures consistent state across components

Next →

Learnbay

Don't: Overuse local state for shared data.

Example:

```
// ❌ Overusing Local State (Avoid)
const ChildA = () => {
  const [data, setData] = useState(null);
  // ...
};
const ChildB = () => {
  const [data, setData] = useState(null);
  // ...
};
```

- State Duplication: Leads to inconsistencies and bugs.
- Complex Data Flow: Harder to manage and synchronize.
- Increased Memory Usage: Redundant states consume more resources

Pro Tip : Use Context API or Redux for complex state scenarios

Next →

Learnbay

Optimize Performance

Do: Use React.memo and useCallback to prevent unnecessary re-renders.

Example:

```
// ✅ Using React.memo
const Display = React.memo(({ value }) => <div>{value}</div>);
    <button onClick={onClick}>{children}</button>
 );
```

- Performance Boost: Reduces unnecessary renders.
- Efficient Rendering: Only updates when props change.
- Resource Saving: Optimizes CPU and memory usage.

Next →

Learnbay

Don't: Pass new references to props on every render

Example:

```
// ❌ Avoid Inline Functions
const Parent = () => {
  const handleClick = () => { /* handle click */ };
  return <Button onClick={() => handleClick()} />;
};
```

- Frequent Re-renders: Causes child components to re-render unnecessarily.
- Performance Degradation: Impacts app responsiveness.
- Memory Leaks: Can lead to unexpected behaviors.

Pro Tip : Profiling tools can help identify performance bottlenecks

Next →

Learnbay

Use PropTypes or TypeScript

Do: Define clear prop types for components

Example:

```
// ✅ Using PropTypes
Button.propTypes = {
  onClick: PropTypes.func.isRequired,
  children: PropTypes.node.isRequired,
};
```

- Type Safety: Prevents bugs by enforcing prop types.
- Better Documentation: Clarifies component usage.
- Enhanced IDE Support: Improves autocomplete and error checking.

Next →

Learnbay

Don't: Skip type checking.

Example:

```
// ❌ Without PropTypes (Avoid)
const Button = ({ onClick, children }) => (
  <button onClick={onClick}>{children}</button>
);
```

- Increased Bugs: Harder to track type-related issues.
- Poor Documentation: Unclear component requirements.
- Maintenance Challenges: Difficult to manage large codebases.

Pro Tip : TypeScript offers robust type safety for larger projects

Next →

Manage Side Effects with useEffect

Do: **Properly handle side effects and cleanup in** useEffect.

Example:

```
// ✅ useEffect with Cleanup
useEffect(() => {
  const subscription = subscribe();
  return () => subscription.unsubscribe();
}, []);
```

- Resource Management: Prevents memory leaks.
- Controlled Execution: Runs effects at appropriate times.
- Reliable Cleanup: Ensures subscriptions and listeners are removed.

Next →

Learnbay

Don't: Forget to include dependencies or cleanup.

Example:

```
// ❌ Missing Dependencies (Avoid)
useEffect(() => {
  fetchData();
}, []); // Might miss dependencies
```

- Stale Data: Effects may use outdated values.
- Memory Leaks: Subscriptions or listeners remain active.
- Unexpected Behaviors: Effects run inconsistently.

Pro Tip : Always specify dependencies to avoid unexpected behaviors

Next →

Code Organization and File Structure

Do: Organize files by feature or component.

Example:

```
/src
  /components
    /Button
      Button.jsx
      Button.css
      Button.test.jsx
  /hooks
  /utils
```

- Scalability: Easily manage large codebases.
- Team Collaboration: Simplifies navigation for multiple developers.
- Maintainability: Locates files quickly based on functionality.

Next →

Learnbay

Don't: Use a flat structure with all files in one directory.

Example:

```
/src
  Button.jsx
  Header.jsx
  Footer.jsx
  ...
```

- Cluttered Directories: Hard to find specific files.
- Poor Organization: No logical grouping of related files.
- Maintenance Nightmare: Difficult to manage as the project grows.

Pro Tip : Consistent structure improves team collaboration and scalability

Next →

Testing Your Components

Do: Write unit and integration tests for your components.

Example:

```
// ✅ Example with Jest
test('calls onClick when clicked', () => {
  const handleClick = jest.fn();
  render(<Button onClick={handleClick}>Click Me</Butt
  fireEvent.click(screen.getByText(/click me/i));
  expect(handleClick).toHaveBeenCalled();
});
```

- Bug Prevention: Catches issues early in development.
- Code Reliability: Ensures components behave as expected.
- Documentation: Tests serve as usage examples.

Next →

Learnbay

Don't: Neglect testing or rely solely on manual testing.

Example:

```
// ❌ No Tests (Avoid)
const Button = ({ onClick, children }) => (
  <button onClick={onClick}>{children}</button>
);
```

- Undetected Bugs: Issues go unnoticed until production.
- Low Confidence: Hard to ensure component reliability.
- Increased Maintenance: Manual testing is time-consuming and error-prone.

Pro Tip : Automated tests catch bugs early and ensure code reliability

Next →

Upskill with

Learnbay

India's most trusted

Program For **Working Professional**

Data Structure Algorithms & System Design

With **Gen-AI** For Software Developers

Program electives:

- GenAI
- Product management
- DevOps
- Data engineering

Get Certification from :

IBM | IIT Guwahati | WOOLF UNIVERSITY | Microsoft