React Clean Code Cheat Sheet

React encourages a component-based architecture, which naturally fosters clean code practices. However, adhering to best practices, staying consistent, and writing maintainable code can sometimes be tricky.

Here's a comprehensive cheat sheet to guide you in writing clean React code:

---

1. Component Design

Keep Components Small and Focused

- **Single Responsibility Principle (SRP)**: Each component should only do one thing.
- **Max 50-100 lines**: If a component exceeds this, it's a sign that it should be split into smaller sub-components.

Functional Components over Class Components

- **Prefer Functional Components**: They are simpler, support hooks, and are generally easier to reason about.

```
// Good: Functional component
const MyComponent = () => {
  return <div>Hello, World!</div>;
};
```

Avoid Side Effects in Components

- **No side effects inside the render function**: Keep your components declarative and avoid triggering side effects (like data fetching or modifying state) during the render cycle.
- **Use `useEffect` or event handlers for side effects**.

Destructure Props and State

- Destructuring props and state makes code cleaner and improves readability.

```
// Good: Destructuring props and state
const MyComponent = ({ name, age }) => {
  const [counter, setCounter] = useState(0);

  return (
    <div>
      <p>{name} is {age} years old.</p>
      <p>Counter: {counter}</p>
    </div>
  );
};
```

Avoid Inline Functions Inside JSX

- **No Inline Functions**: Inline functions on every render can hurt performance and make debugging harder.

```
// Bad: Inline function in JSX
<button onClick={() => handleClick()}>Click Me</button>

// Good: Use pre-defined functions
const handleClick = () => { ... };
<button onClick={handleClick}>Click Me</button>
```

---

2. Hooks Best Practices

Use `useEffect` with Care

- Avoid running unnecessary side effects in `useEffect`. Specify dependencies carefully.

```
// Good: Specify dependencies
useEffect(() => {
  fetchData();
}, [userId]); // Only re-run when userId changes
```

Avoid Multiple Effects in One Hook

- Each `useEffect` should focus on a single concern.

```
// Good: Separate concerns
useEffect(() => { fetchData(); }, [dataId]);
useEffect(() => { document.title = `New data: ${data}`; }, [data]);
```

Use Custom Hooks for Reusability

- If you have repeated logic, extract it into custom hooks.

```
// Good: Custom Hook
function useFetchData(url) {
  const [data, setData] = useState(null);
  useEffect(() => {
    fetch(url).then(response => setData(response.data));
  }, [url]);
```

```
  return data;
}
```

---

3. State Management

Lift State Up

- If multiple components need to share state, lift the state up to their closest common ancestor.
- **Avoid global state unless necessary**: Use React Context or external libraries like Redux only when needed.

Use `useState` Wisely

- **Avoid deeply nested state**: Keep state as flat as possible to avoid complex updates.

```
// Good: Flat state structure
const [user, setUser] = useState({ name: '', email: '' });

// Bad: Nested state structure
const [user, setUser] = useState({ info: { name: '', email: '' } });
```

---

4. Code Structure

Organize Your Files

- **Group related components**: Group components, hooks, and utilities in the same directory.
- **Use clear and consistent naming**: Name files based on the component they export. E.g., `MyComponent.js` for a component named `MyComponent`.

```
src/
  components/
    Button.js
    Header.js
  hooks/
    useForm.js
  utils/
    formatDate.js
```

Avoid Logic in JSX

- **Keep JSX clean**: Place business logic outside JSX to improve readability and testing.

```
// Good: Logic outside JSX
const isValid = value.length > 0;
return (
  <div>
    {isValid ? 'Valid input' : 'Invalid input'}
  </div>
);

// Bad: Logic inside JSX
return (
  <div>
    {value.length > 0 ? 'Valid input' : 'Invalid input'}
  </div>
);
```

---

5. Styling

CSS-in-JS vs Traditional CSS

- Choose a styling approach that fits your needs (e.g., CSS Modules, styled-components, or traditional CSS).
- **Use CSS variables for theme management** and **media queries for responsiveness**.

```
// Styled-components example
const Button = styled.button`
  background-color: ${props => props.primary ? 'blue' : 'gray'};
  color: white;
`;
```

Avoid Inline Styles in JSX

- **Use CSS classes or CSS-in-JS libraries** instead of inline styles.

```
// Bad: Inline styles
<div style={{ backgroundColor: 'blue', color: 'white' }}>Hello</div>

// Good: CSS Class
<div className="my-class">Hello</div>
```

---

6. Performance Optimization

Memoize Expensive Computations

- Use `useMemo` and `useCallback` to memoize values or functions that depend on specific dependencies.

```
// Good: UseMemo for expensive calculation
const expensiveValue = useMemo(() => computeExpensiveValue(data), [data]);

// Good: UseCallback for stable functions
const handleClick = useCallback(() => { ... }, [dependency]);
```

Lazy Loading Components

- Use `React.lazy` and `Suspense` for code splitting and lazy loading components.

```
const LazyComponent = React.lazy(() => import('./LazyComponent'));

<Suspense fallback={<div>Loading...</div>}>
  <LazyComponent />
</Suspense>
```

---

7. Error Handling

Use Error Boundaries

- Implement **Error Boundaries** to catch JavaScript errors anywhere in your component tree.

```
// Error Boundary Example
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

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

  componentDidCatch(error, info) {
    logErrorToMyService(error, info);
  }

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

---

8. Testing

Write Unit Tests for Components

- Use **React Testing Library** and **Jest** to test your components and hooks.
- Focus on testing **user behavior** and **rendering**, not implementation details.

```
// Example: React Testing Library
import { render, screen } from '@testing-library/react';
import MyComponent from './MyComponent';

test('renders Hello World text', () => {
  render(<MyComponent />);
  expect(screen.getByText(/Hello, World!/)).toBeInTheDocument();
});
```

---

9. Miscellaneous

Keep Dependencies Up-to-Date

- Regularly update dependencies to keep your codebase secure and maintainable.

Document Components

- Document important components, hooks, and utility functions to make the codebase easier to understand for new developers.

```
/**
 * A simple button component
 * @param {object} props - The component props
 * @param {string} props.label - The button label
 * @param {function} props.onClick - The click event handler
 */
const Button = ({ label, onClick }) => (
  <button onClick={onClick}>{label}</button>
);
```

Conclusion

By following these best practices, you'll be well on your way to writing clean, maintainable, and performant React code. Remember, consistency is key! Always strive to make your code easy to read, reusable, and simple to debug.