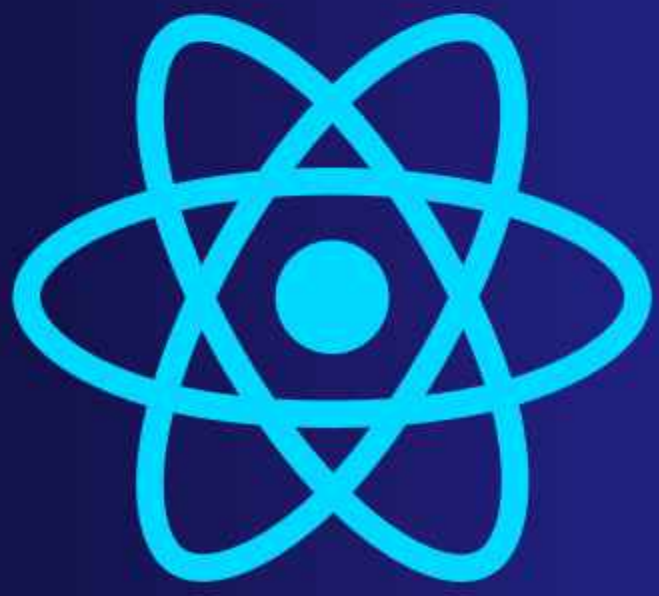
Save for later

DESIGN PATTERNS IN REACT



@subhajit-adhikary

001

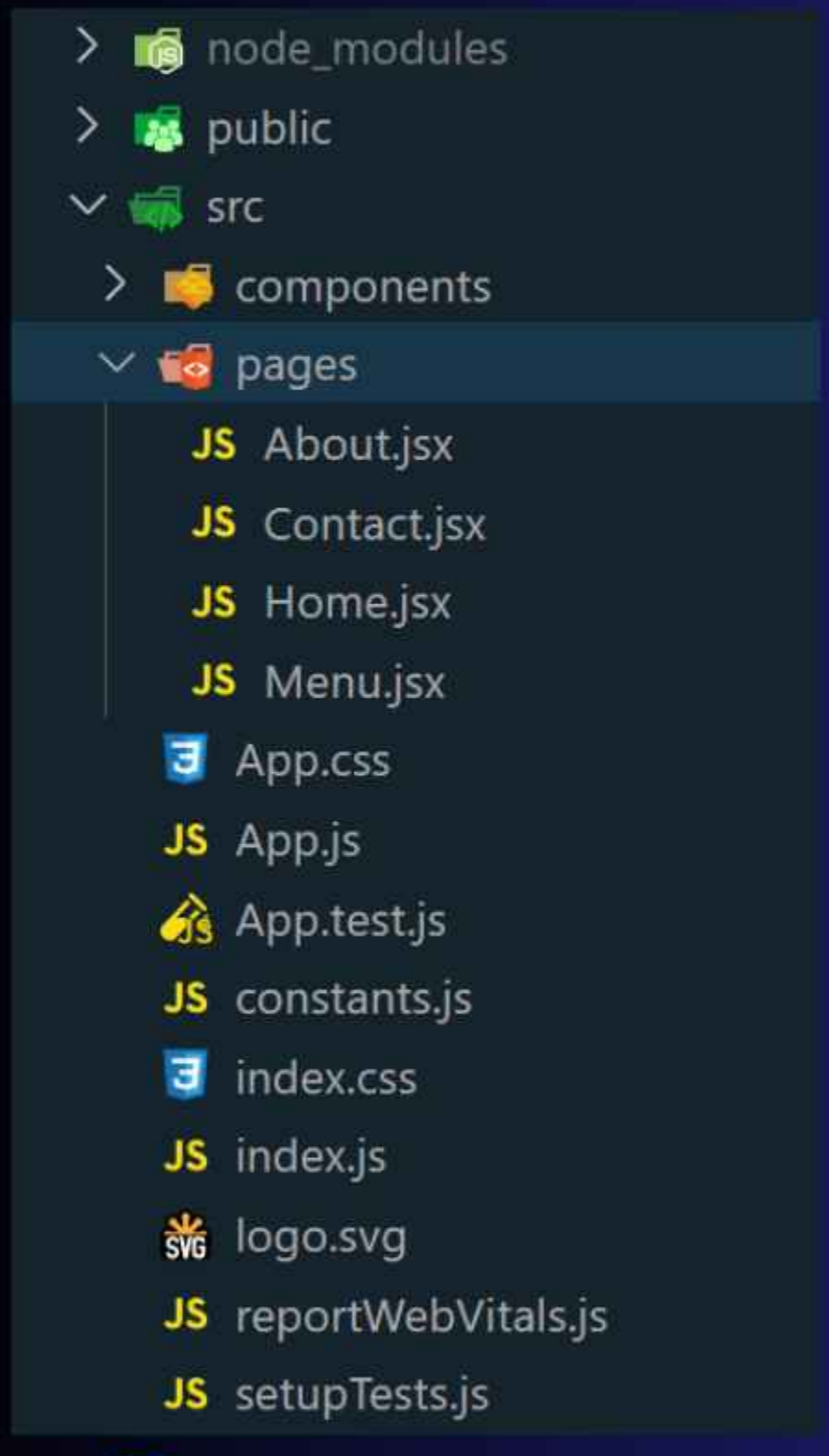
STATE MANAGEMENT

Keep your state in check! It's the backbone of React apps. Share state between components wisely by moving it up to the nearest common ancestor. For complex scenarios, consider using state management tools like Redux.

```jsx
// Good: State lifted up
import React, { useState } from 'react';

// ChildComponent receives count and setCount directly as props
function ChildComponent({ count, setCount }) {
  return <button onClick={() => setCount(count + 1)}>Increment</button>;
}

// ParentComponent manages the count state and passes it down to
ChildComponent
function ParentComponent() {
  const [count, setCount] = useState(0);

  return <ChildComponent count={count} setCount={setCount} />;
}

// App component renders the ParentComponent
function App() {
  return <ParentComponent />;
}

export default App;
```

@subhajit-adhikary

PROJECT STRUCTURE



Start strong with a solid organization. Arrange your project by features or modules, not just file types. This way, finding and updating code related to specific functions becomes a breeze.

@subhajit-adhikary

003

DESTRUCTURING PROPS

Keep things tidy with destructuring and PropTypes. Destructuring props in functional components boosts readability. PropTypes help document and validate props, saving you from runtime headaches.

```jsx
// Destructuring props for improved readability
function SayHello({ name, age}) {
    return <div>{`Hello, ${name}! I am ${age} years old.`}</div>;
}
```

```jsx
// PropTypes for documenting and validating props
import PropTypes from 'prop-types';

function SayHello({ name, age }) {
  return <div>{`Hello, ${name}! I am ${age} years old.`}</div>;
}

SayHello.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number.isRequired,
};
```

@subhajit-adhikary

COMPONENT LIFECYCLE

Say hello to functional components and Hooks! They're your new best friends for managing state, side effects, and context. They simplify your code and make lifecycles a lot less complicated.

```jsx
// Functional component with useState Hook
function DemoComponent() {
    const [count, setCount] = useState(0);

    return <button onClick={() => setCount(count+1)}>{count}</button>;
  }
```

@subhajit-adhikary

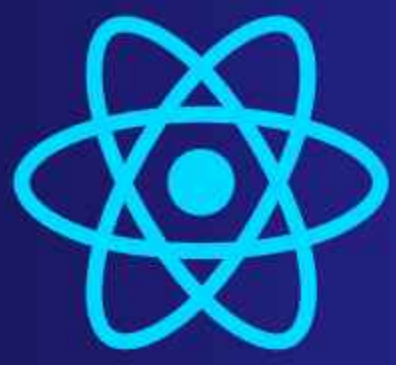
005

ERROR BOUNDARIES

Handle hiccups gracefully! Wrap your components with error boundaries to prevent full-on crashes. It's all about giving users a smoother experience and making debugging less of a headache.

```jsx
// defining error boundary
function ErrorBoundary({ children }) {
  const [isError, setIsError] = useState(false);

  const componentDidCatch = (error, info) => {
    setIsError(true);
    console.error(error, info);
  };

  if (isError) {
    return <div>Some error occurred!</div>;
  }

  return children;
}
```

@subhajit-adhikary



REUSABLE COMPONENTS

Why reinvent the wheel? Create a component library for common UI patterns. It speeds up development and keeps your app looking consistent.

```jsx
// Reusable Input component
function Input({ onChange , value}) {
    return <input onChange={onChange} value={value} />;
  }
```

@subhajit-adhikary

CODE SPLITTING

Less is more! Break down your app into smaller chunks and load them when needed. This speeds up load times and keeps your app feeling responsive.

```jsx
// Using React.lazy for code splitting
const LazyComponent = React.lazy(() => import('./LazyComponent'));

// Suspense will show the fallback div till LazyCompnent not ready
const App = () => (
  <React.Suspense fallback={<div>On the way ... </div>}>
    <LazyComponent />
  </React.Suspense>
);
```

@subhajit-adhikary

PERFORMANCE OPTIMIZATION

Keep things snappy with memoization. Memoize components with React.memo . This way, you avoid unnecessary renders and keep your app running smoothly.

```jsx
// defining memoizedcomponent using React memo
import { memo } from 'react';
const MemoizedComponent = (props) => {
  /* render using props */
  console.log('rendering component');
  return (
    <div>
      <p>{props.name}</p>
    </div>
  );
};

export default memo(MemoizedComponent);
```

@subhajit-adhikary

ACCESSIBILITY FEATURES

Everyone's invited! Make sure your app is accessible to all users. Stick to web accessibility standards, use semantic HTML, and test with screen readers to ensure inclusivity.

```
// Using semantic HTML for better accessibility
<button aria-label="Close" onClick={onClose}>
  <span aria-hidden="true">&times;</span>
</button>
```

@subhajit-adhikary

TESTING PROPERLY

Don't skip this step! Test, test, and test some more. Use tools like Jest and React Testing Library to cover all your bases: unit tests, integration tests, and end-to-end tests.

```
import { render, screen } from '@testing-library/react';
import App from './App';

test('renders learn react link', () => {
  render(<App />);
  const linkElement = screen.getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

@subhajit-adhikary

Save for later

KEEP LEARNING

PS:- Remember, these tips are just the start of your journey with React.js. There's always more to learn and explore, so keep coding and keep growing!

Save this post for future use

Was this helpful ??

@subhajit-adhikary