



Improve your ReactJS code



READ



SPEED



ERROR HANDLING



Use Types

Using static typing with tools like TypeScript or PropTypes can help catch errors early, improve code quality, and provide better documentation for your components' expected props and data.



```
import PropTypes from 'prop-types';

function MyComponent({ name, age }) {
  // Component logic
}

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



specify the expected
data types





Use Helpers / Utils

Helpers or utility functions can **encapsulate common, reusable logic** in your application, making your components cleaner and more focused on their core functionality.



```
// Helper.js
export function formatName(firstName,
lastName) {
  return `${firstName} ${lastName}`;
}

// Component.js
import { formatName } from './Helper';
const fullName = formatName("John", "Doe");
```

create a utility
function

imported and
used here





Learn How to Use Props

Props (short for properties) are a way to **pass data from parent to child components** in React. Properly understanding and utilizing props is essential for building **reusable and modular components**.

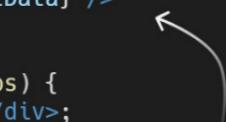


```
// ParentComponent.js
import ChildComponent from './ChildComponent';
const parentData = "Data from parent";
<ChildComponent data={parentData} />
```

```
// ChildComponent.js
function ChildComponent(props) {
  return <div>{props.data}</div>;
}
```



render



import a child component and pass data to it using props





Have One File for Each Component

Organizing each component in a separate file makes your codebase more **organized and manageable**. It also encourages reusability and **easier maintenance**.

Example: File structure where each component has its own file.

Implement a 404 Component and Route

Create a 404 component and route to **handle unknown URLs**, providing a user-friendly error page when a route doesn't exist.

Example: Using react-router for routing





Don't Use Inline Functions

Avoid defining inline functions within the render method as they can **create new function instances on each render**, potentially leading to **performance issues**.



```
// Inefficient  
render() {  
    return <button onClick={() => this.handleClick()}>  
        Click me  
    </button>;  
}  
  
// Efficient  
render() {  
    return <button onClick={this.handleClick}>  
        Click me  
    </button>;  
}
```

new function on
each render



directly references the
handleClick function





Fetch Data Progressively

Fetching data progressively means **loading data as needed**, rather than all at once. This can improve initial page load times and reduce data transfer.



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

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
}
```



Use the `lazy()` and `Suspense()` Functions





Use Unique Values for Key Attributes

When rendering a list of elements, each element should have a unique key attribute. This helps React **efficiently update the DOM** when elements are added, removed, or rearranged.



```
const items = [1, 2, 3];
const listItems = items.map(
  item => <li key={item}>{item}</li>
);
```



a unique key is provided
for each element





Use Constants

Constants are used to store static values or **configuration settings that are reused** throughout your application. Using constants makes your code more maintainable and reduces the chances of introducing bugs due to typos or inconsistencies.



```
// Constants.js
export const API_BASE_URL =
  "https://api.example.com";
```



```
// Component.js
import { API_BASE_URL } from './Constants';
const apiUrl = `${API_BASE_URL}/endpoint`;
```

defined

imported





codewithsloba.com

Get a weekly digest of my tips and
tutorials by subscribing now.