FROM CONCEPTS
TO CONFIDENCE

Build
Learn
Grow

THE COMPLETE

React Notes

Beginner to Advanced

LEARN
FUNDAMENTALS

BUILD
REAL APPS

SOLVE
REAL PROBLEMS

GROW
YOUR SKILLS

Modern React • Practical Examples • Real-World Projects

Dream
Code
Achieve
Repeat

"A Stronger You in Tech"

Components
Hooks
TypeScript
State Management
Testing
Deployment
Real-World Projects

Good
Code
Better
Tomorrow

SIMPLE CODE
BIGGER POSSIBILITIES

- Plan
- Practice
- Build
- Improve
- Repeat

CLEAN EXPLANATIONS

PRACTICAL EXAMPLES

REAL-WORLD PROJECTS

INTERVIEW PREPARATION

CAREER GROWTH

More
Than Notes
A Brighter Future

REACT TODAY
A BRIGHTER TOMORROW

LET'S
BUILD A BETTER
TOMORROW

Build
Learn
Grow ↗

React.js Complete Notes

"Components today, better UIs tomorrow."

A better web, one component at a time. ☺

Beginner to Advanced – Frontend Development Guide

LEARN • PRACTICE • BUILD • MASTER

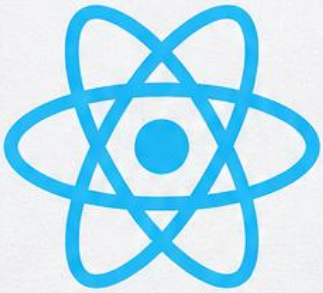
Think in Components

- [x] Component-Based
- [x] Declarative
- [x] Flexible
- [x] High Performance
- [x] Huge Community

1 Introduction to React

What is React?

React is a JavaScript library for building user interfaces, developed and maintained by Meta (Facebook). It allows you to build reusable UI components and efficiently update and render them when data changes.



React.js

Just Build It ♡

"React makes it painless to create interactive UIs." – Dan Abramov

A Simple React Component

App.jsx

```jsx
import React from 'react';

function App() {
  return (
    <div className="app">
      <h1>Hello, React! 👋</h1>
      <p>Let's build something amazing.</p>
    </div>
  );
}

export default App;
```

Benefits of React

- [x] **Component-based architecture**
  Build reusable and maintainable UIs
- [x] **Virtual DOM**
  Optimized rendering and better performance
- [x] **Large ecosystem & community**
  Plenty of resources and support
- [x] **Declarative UI**
  Easier to understand and debug
- [x] **Used by top companies**
  Facebook, Instagram, Netflix, Airbnb, and more

"Learn Once, Build Anywhere"

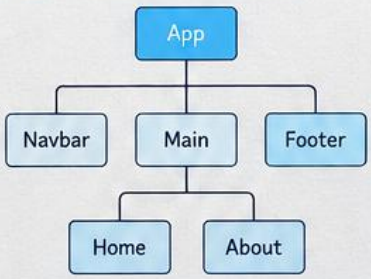
React Ecosystem

Next.js (SSR/SSG)	React Router (Routing)	Redux (State Management)
Vite (Build Tool)	ESLint (Linting)	React DevTools (Debugging)

Use Cases

- [x] Single Page Applications (SPAs)
- [x] E-commerce websites
- [x] Social media platforms
- [x] Dashboards and admin panels
- [x] Mobile apps (with React Native)
- [x] Progressive Web Apps (PWAs)

Component Tree



Real-world Example (JSX)

Welcome.jsx

```jsx
const Welcome = ({ name }) => {
  return (
    <div className="welcome">
      <h2>Welcome, {name}! 👋</h2>
    </div>
  );
};

export default Welcome;
```

"Small components make a big difference."

Keep Learning ♡

JavaScript First, React Next!

Same Language Bigger Possibilities →

JavaScript Prerequisites

Core JavaScript Concepts You Should Know Before Learning React

"A strong JavaScript foundation makes React easier and more enjoyable!"

1 Variables

Store and manage data.

```js
let name = 'React';
const version = 18;
var isAwesome = true;

console.log(name, version, isAwesome);
```

Use let and const (avoid var).

2 Functions

Reusable blocks of code.

```js
// Regular function
function add(a, b) {
  return a + b;
}

// Arrow function
const multiply = (a, b) => a * b;

console.log(add(2, 3)); // 5
console.log(multiply(2, 3)); // 6
```

Functions help you write modular and reusable code.

3 Objects

Store key-value pairs.

```js
const user = {
  name: 'Alice',
  age: 25,
  role: 'Developer'
};

console.log(user.name); // Alice
console.log(user['role']); // Developer
```

Objects model real-world entities.

4 Arrays

Store multiple values.

```js
const fruits = ['Apple', 'Banana', 'Mango'];

console.log(fruits[0]); // Apple
fruits.push('Orange');

fruits.forEach((fruit, index) => {
  console.log(index, fruit);
});
```

Arrays are ordered and zero-indexed.

5 Destructuring

Extract values easily.

```js
// Array destructuring
const [a, b] = [10, 20];

// Object destructuring
const { name, age } = user;

console.log(a, b); // 10 20
console.log(name, age); // Alice 25
```

Makes your code cleaner and more readable.

6 Spread Operator

Copy and merge data.

```js
// Copy an array
const nums1 = [1, 2, 3];
const nums2 = [...nums1, 4, 5];

// Merge objects
const user = { name: 'Alice' };
const updatedUser = { ...user, age: 25 };

console.log(nums2); // [1,2,3,4,5]
console.log(updatedUser);
// { name: 'Alice', age: 25 }
```

Spread helps create new copies without mutating the original.

7 Modules

Organize and reuse code.

math.js

```js
// Export
export const add = (a, b) => a + b;

export const PI = 3.14;
```

app.js

```js
// Import
import { add, PI } from './math.js';

console.log(add(2, 3)); // 5
console.log(PI); // 3.14
```

ES6 modules help you keep your code modular and maintainable.

8 Promises

Handle asynchronous operations.

```js
const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Data loaded!');
    }, 1000);
  });
};

fetchData()
  .then((data) => console.log(data))
  .catch((error) => console.error(error));
```

Promises help handle async code and avoid callback hell.

Promise States

Pending → Resolved (then) → Rejected (catch)

"Master JavaScript today, build amazing things tomorrow."

JS

Same Concepts Power React

"Good JavaScript skills = Smoother React Journey!"

Keep Going!

Good tools make great developers ! :)

Project Setup

Get Your Development Environment Ready

A solid setup leads to a smooth learning journey!

Setup Today
Build Tomorrow

React + Modern Tools = Awesome Projects ♡

1 Node.js

JavaScript runtime to run tools and scripts.

- [x] Run JavaScript outside the browser
- [x] Required for npm and build tools
- [x] Install from nodejs.org

```bash
# Check Node.js version
node -v
# Expected output (e.g.)
v20.11.0
```

Use LTS version for better stability.

2 npm

Node Package Manager to manage project dependencies.

- [x] Install and manage packages
- [x] Used in all modern JavaScript projects
- [x] Comes with Node.js (no separate install needed)

```bash
# Check npm version
npm -v
# Expected output (e.g.)
10.2.4
```

Use npm to install libraries like React, React Router, etc.

3 Vite

Modern build tool for fast development.

- [x] Super fast dev server
- [x] Better developer experience
- [x] Native ES modules support
- [x] Officially recommended for React

```bash
# Create a new React project with Vite
npm create vite@latest my-react-app
-- --template react
```

Vite is lightweight, fast, and great for modern React projects.

4 Project Structure

Understand the folder structure of a Vite + React project.

```
my-react-app/
├─ node_modules/      # Installed packages
├─ public/            # Static assets
│  └─ vite.svg
├─ src/               # Source code
│  ├─ assets/         # Images, styles, etc.
│  ├─ components/     # Reusable components
│  ├─ App.jsx         # Main component
│  ├─ main.jxx        # Entry point
│  └─ index.css       # Global styles
├─ .gitignore         # Git ignore rules
├─ index.html         # HTML template
├─ package.json       # Project config
├─ vite.config.js     # Vite configuration
└─ README.md          # Project documentation
```

"A clean project structure keeps your mind clean." :)

Organized Code Happy You :)

5 Development Server

Run your project locally and start building!

```bash
# Navigate to your project folder
cd my-react-app

# Install dependencies
npm install

# Start the development server
npm run dev
```

Once you run the dev server, you'll see an output like this:

```
vite v5.0.0  ready in 300ms

→ Local:   http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help
```

Open http://localhost:5173 in your browser.

Your React app is running!

"Setup well. Build more. Learn faster."

Small Steps Big Projects ♡

Write UI with JavaScript ✨

JSX Fundamentals

The Syntax Extension for React

"JSX makes it easy to write and visualize UI."

Same Concepts Bigger Possibilities ♡

1 What is JSX?

JSX (JavaScript XML) is a syntax extension for JavaScript that allows you to write HTML-like code inside JavaScript. It gets transformed into regular JavaScript by tools like Babel (in Create React App, Vite, etc.).

```jsx
const element = (
  <h1>Hello, React! 👋</h1>
);
```

JSX looks like HTML, but it's actually JavaScript!

2 JSX Expressions

You can embed JavaScript expressions inside JSX using curly braces { }.

```jsx
const name = "React";
const element = (
  <div>
    <h2>Hello, {name}! 👋</h2>
    <p>2 + 3 = {2 + 3}</p>
    <p>Is JS fun? {true ? 'Yes' : 'No'}</p>
  </div>
);
```

You can use variables, expressions, functions, and ternary operators inside { }.

3 JSX Rules

Follow these important rules while using JSX.

- ☑ Return a single parent element (or use fragments).
- ☑ Use camelCase for attributes (e.g., className instead of class).
- ☑ Close all tags (even self-closing tags).
- ☑ Use { } for JavaScript expressions.
- ☑ Components must start with a capital letter.
- ☑ Comments are written like this: {/* comment */}

JSX is stricter than HTML, but it helps you write cleaner and more predictable code.

4 JSX Attributes

JSX uses camelCase for most HTML attributes. Some attribute names are different in JSX.

```jsx
const element = (
  <div className="container">
    <img src="logo.png" alt="React Logo" />
    <input type="text" placeholder="Enter name" />
    <button onClick={handleClick} disabled={false}>
      Click Me
    </button>
  </div>
);
```

Common JSX Attributes

HTML	JSX
class	className
for	htmlFor
onclick	onClick
tabindex	tabIndex
readonly	readOnly

Use camelCase in JSX ✓

"Small Syntax Big Impact."

5 Fragments

Fragments let you group multiple elements without adding an extra DOM node.

```jsx
// Using React Fragment
import { Fragment } from 'react';

const element = (
  <Fragment>
    <h1>Title</h1>
    <p>Description</p>
  </Fragment>
);
```

```jsx
// Shorthand syntax
const element = (
  <>
    <h1>Title</h1>
    <p>Description</p>
  </>
);
```

Fragments keep your DOM clean and avoid unnecessary elements.

Rendered Output (in Browser)

http://localhost:5173

Title

Description

Cleaner UI Better Performance

"JSX bridges the gap between logic and design."

Keep Building ♡

Reusable Components = Scalable Apps ♡

Components

Build, Reuse, Compose

"Small pieces. Big possibilities."

Components Make React Powerful!

Write Once Use Anywhere ☺

1 Function Components

The modern way to create components in React.

```jsx
function Welcome() {
  return (
    <div>
      <h2>Hello, React!</h2>
      <p>This is a function component.</p>
    </div>
  );
}
```

Function components are simple, reusable and preferred in modern React (with Hooks).

2 Imports and Exports

Split your code into separate files and reuse components.

Button.jsx

```jsx
// Exporting a component
export default function Button() {
  return <button>Click Me</button>;
}
```

App.jsx

```jsx
// Importing the component
import Button from './components/Button';

function App() {
  return (
    <div>
      <h1>My App</h1>
      <Button />
    </div>
  );
}
```

Use export default for default exports and import to use them in other files.

3 Reusable UI Components

Create components that can be reused across your app.

Card.jsx

```jsx
function Card({ title, children }) {
  return (
    <div className="card">
      <h2>{title}</h2>
      <div className="card-body">{children}</div>
    </div>
  );
}

export default Card;
```

Benefits

- ☑ Keeps code clean
- ☑ Easier maintenance
- ☑ Consistent UI
- ☑ Saves development time
- ☑ Build a component library

4 Props – Passing Data

Props (short for properties) let you pass data to components.

```jsx
function Greeting({ name, age }) {
  return (
    <div>
      <h2>Hello, {name}!</h2>
      <p>You are {age} years old.</p>
    </div>
  );
}
// Usage
<Greeting name="Alice" age={25} />
```

Props make components dynamic and reusable.

5 Default Props

Set default values for props.

```jsx
function Button({ text = 'Click' }) {
  return <button>{text}</button>;
}

// Usage
<Button />                  // Click
<Button text="Submit" />    // Submit
```

Default props ensure your component works even if no value is provided.

6 Children Prop

Use children to pass JSX content.

```jsx
function Box({ children }) {
  return <div className="box">{children}</div>;
}

// Usage
<Box>
  <h3>Title</h3>
  <p>This is inside the Box component.</p>
</Box>
```

The children prop lets you nest elements inside a component.

7 Read-only Props

Props are read-only and cannot be modified inside the component.

```jsx
function User({ name }) {
  // name = 'Bob'; ❌ Cannot reassign
  return <h2>{name}</h2>;
}

// Correct: Use a new variable if needed
function User({ name }) {
  const displayName = name || 'Guest';
  return <h2>{displayName}</h2>;
}
```

Props are immutable. Use state if you need to change values.

"Reusable components today, better applications tomorrow."

Build Beautiful Things ♡

Control What Renders Build Better UIs

Conditional Rendering

Right Content at the Right Time

Render Different UI Based on Conditions

"Same component. Different possibilities."

1 If Statement

Use a regular if statement (outside JSX).

```jsx
function Profile({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h2>Welcome back! 👋</h2>;
  }
  return <h2>Please log in.</h2>;
}
```

Use if for more complex logic before the return statement.

2 Ternary Operator

Use the ternary operator for simple conditions.

```jsx
function Status({ isOnline }) {
  return (
    <div>
      {isOnline ? (
        <span className="online">🟢 Online</span>
      ) : (
        <span className="offline">🔴 Offline</span>
      )}
    </div>
  );
}
```

Ternary Syntax

```
condition
? <TrueComponent />
: <FalseComponent />
```

Great for inline rendering inside JSX.

3 Logical AND (&&)

Use && for rendering something conditionally.

```jsx
function Notifications({ count }) {
  return (
    <div>
      {count > 0 && (
        <p>You have {count} new notifications.</p>
      )}
    </div>
  );
}
```

How it works?

If the left side is true, the right side is rendered.

Useful for conditional UI elements.

4 Returning Null

Return `null` when you don't want to render anything.

```jsx
function Modal({ isOpen }) {
  if (!isOpen) {
    return null;
  }
  return (
    <div className="modal">
      <h2>Modal Content</h2>
    </div>
  );
}
```

When to use?

- ☑ Hide components without removing them completely.
- ☑ Useful in conditional logic.
- ☑ Keeps the code cleaner.

Loop Filter Render Repeat Build Amazing UIs

Lists and Keys

Good Keys Happy React Better Performance

Render Collections of Data Efficiently

"Maps data to real UI."

1 Using map()

Use .map() to render lists from arrays.

```jsx
const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' }
];

function UserList() {
  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

2 Filtering Lists

Use .filter() to show only specific items.

```jsx
const users = [
  { id: 1, name: 'Alice', active: true },
  { id: 2, name: 'Bob', active: false }
];

const activeUsers = users.filter(
  user => user.active
);

return (
  <ul>
    {activeUsers.map(user => (
      <li key={user.id}>{user.name}</li>
    ))}
  </ul>
);
```

3 Stable Keys

Always use a unique and stable key (preferably from data).

- ✅ Use unique IDs (e.g., user.id)
- ✅ Keys should be stable across renders
- ✅ Avoid using array index (if list can change)

❌ *Avoid this (if list changes)*

```jsx
{items.map((item, index) => (
  <li key={index}>{item.name}</li>
))}
```

✅ *Prefer this*

```jsx
{items.map(item => (
  <li key={item.id}>{item.name}</li>
))}
```

4 List Rendering Example

A complete example with mapping and filtering.

```jsx
function ProductList({ products }) {
  const availableProducts =
  products.filter(p => p.inStock);

  return (
    <div>
      <h2>Available Products</h2>
      <ul>
        {availableProducts.map(product => (
          <li key={product.id}>
            {product.name} - ${product.price}
          </li>
        ))}
      </ul>
    </div>
  );
}
```

"Render the right things. Keep your lists fast and clean."

Learn Build Grow

Interactive UIs Make Better Experiences

Event Handling

Respond to User Interactions

"Events make your app interactive and alive."

Handle Events Manage State Build Dynamic Apps

1 Click Events

Handle user clicks and other events in React.

```jsx
function Button() {
  const handleClick = () => {
    console.log('Button clicked!');
  };

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

Use camelCase event names like onClick, onChange, etc.

2 Event Objects

Access event details using the event object.

```jsx
function handleClick(e) {
  console.log('Event type:',
    e.type);
  console.log('Target:',
    e.target);
  console.log('Value:',
    e.target.value);
}
```

The event object contains useful info like type, target, value, etc.

3 Event Propagation

Events bubble up the component tree (unless stopped).

```jsx
function Parent() {
  const handleParent = () => {
    console.log('Parent clicked');

  const handleChild = (e) => {
    e.stopPropagation(); // Stop
    console.log('Child clicked');
  };

  return (
    <div onClick={handleParent}>
      <button onClick={handleChild}>
        Click Me
      </button>
    </div>
  );
}
```

Use e.stopPropagation() to prevent event bubbling.

4 Passing Handlers

You can pass event handlers as props to child components.

```jsx
// Parent Component
function App() {
  const handleClick = () => {
    console.log('Hello from
    parent!');
  };

  return <Button onClick={
    handleClick} />;
}

// Child Component
function Button({ onClick }) {
  return (
    <button onClick={onClick}>
      Click Me
    </button>
  );
}
```

Pass functions as props just like any other value.

State = Dynamic UI

State with useState

Manage and Update Component State

"State lets your components remember and react to changes."

Small Changes Big Differences

1 Initialization

Declare state using the useState hook.

```jsx
import { useState } from 'react';

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

  return (
    <div>
      <p>Count: {count}</p>
    </div>
  );
}
```

useState returns an array with the current state value and a function to update it.

2 Updating State

Use the setter function to update state.

```jsx
function Counter() {
  const [count, setCount] =
    useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() =>
        setCount(count + 1)}>
        Increment
      </button>
    </div>
  );
}
```

State updates trigger a re-render of the component.

3 Functional Updaters

Use a function when the new state depends on the previous one.

```jsx
function Counter() {
  const [count, setCount] =
    useState(0);

  const increment = () => {
    setCount(prevCount =>
      prevCount + 1);
  };

  return (
    <button onClick={increment}>
      Count: {count}
    </button>
  );
}
```

Functional updaters are useful when the next state depends on the previous state.

4 Multiple State Variables

You can use multiple useState hooks in a single component.

```jsx
function UserProfile() {
  const [name, setName] =
    useState('Alice');
  const [age, setAge] =
    useState(25);

  return (
    <div>
      <p>Name: {name}</p>
      <p>Age: {age}</p>
      <button onClick={() =>
        setAge(age + 1)}>
        Birthday
      </button>
    </div>
  );
}
```

Each piece of state is independent and can be updated separately.

"Events handle actions. State remembers them."

Keep Coding!

Understand State Build Better Apps

State in Depth

How State Really Works in React

"State is a snapshot, not a variable."

Same State Different Possibilities

Immutable Updates = Predictable UI

1 State Snapshots

State behaves like a snapshot. Updating state does not change the current value, but schedules a re-render.

```jsx
import { useState } from 'react';

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

  console.log(count); // 0 (snapshot)

  const handleClick = () => {
    setCount(count + 1);
    console.log(count); // still 0
  };

  return <button onClick={handleClick}>
    Count: {count}
  </button>;
}
```

State is a snapshot of the value at the time of render.

2 Batching Updates

React batches multiple state updates for better performance. All updates inside the same event are grouped.

```jsx
function handleClick() {
  setCount(count + 1);
  setCount(count + 1);
  setCount(count + 1);

// React batches these updates
// Result: count increases by 1
}

// To update based on previous state
setCount(c => c + 1);
setCount(c => c + 1);
setCount(c => c + 1);
// Result: count increases by 3
```

Use functional updates when the new state depends on the previous state.

3 Updating Objects and Arrays

State should be treated as immutable. Always create a new object or array instead of mutating the existing one.

Updating Objects

```jsx
const [user, setUser] = useState({
  name: 'Alice',
  age: 25
});
// ✅ Correct (immutable update)
setUser({ ...user, age: 26 });

// ❌ Wrong (mutates state)
user.age = 26;
setUser(user);
```

Updating Arrays

```jsx
const [items, setItems] = useState([1, 2, 3]);

// ✅ Add item
setItems([...items, 4]);

// ✅ Remove item
setItems(items.filter(item => item !== 2));

// ❌ Wrong (mutates state)
items.push(4);
setItems(items);
```

Render Reconcile Update Repeat

Rendering in React

From State to Screen

"React turns your state into UI."

Same Concepts Bigger Understanding

4 Render and Commit

React has a two-step process:
1. Render – compute the next UI
2. Commit – update the DOM

Trigger
(state/props change)
↓
Render
(Compute next UI)
↓
Commit
(Update the DOM)

Rendering is pure and does not mutate the DOM. The commit phase applies changes.

5 Reconciliation

React compares the new UI with the previous one using a smart algorithm (reconciliation) to update only what has changed.

Previous Render

```
<App>
  <Header />
  <Main />
  <Footer />
</App>
```

→

New Render

```
<App>
  <Header />
  <Main />
  <Footer />
</App>
```

- ✅ Only changed components are updated.
- ✅ Makes React fast and efficient.
- ✅ You don't need to manually optimize in most cases.

6 Component Identity

React uses the component type and its position in the tree to determine identity.

```jsx
function App() {
  return (
    <div>
      <Counter /> {/* Same position */}
      <Counter /> {/* Different position */}
    </div>
  );
}
```

If a component stays in the same position, its state is preserved.

Changing the position or type treats it as a new component (with fresh state).

7 Resetting State

You can reset a component's state by changing its key or removing it from the tree.

```jsx
function App() {
  const [show, setShow] = useState(true);

  return (
    <div>
      <button onClick={() =>
        setShow(!show)}>
        Toggle
      </button>
      {show && <Counter key={"1"} />}
    </div>
  );
}
```

Use a different key to reset state:

```jsx
<Counter key={someId} />
```

"Understand how React renders, and you'll write better React code."

Deeper Knowledge Stronger Apps

Good Forms Better Experiences

Forms and Inputs

Collect and Handle User Input

Input Validate Submit Build Amazing Apps

"Forms turn user interactions into real data."

1 Controlled Inputs

Controlled inputs keep the form data in React state.

```jsx
import { useState } from 'react';

function ControlledInput() {
  const [name, setName] = useState('');

  return (
    <div>
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Enter your name"
      />
      <p>You typed: {name}</p>
    </div>
  );
}
```

The input value is controlled by React state using value and onChange.

2 Uncontrolled Inputs

Uncontrolled inputs use a ref to access the value.

```jsx
import { useRef } from 'react';

function UncontrolledInput() {
  const inputRef = useRef();

  const handleSubmit = () => {
    alert('Value: ' + inputRef.current.value);
  };

  return (
    <div>
      <input ref={inputRef}
        type="text"
        placeholder="Enter something" />
      <button onClick={handleSubmit}>
        Get Value
      </button>
    </div>
  );
}
```

Use refs when you don't need to control the value on every change.

3 Checkbox Input

Handle checkbox (boolean) values in React.

```jsx
import { useState } from 'react';

function Checkbox() {
  const [isChecked, setIsChecked] =
  useState(false);

  return (
    <label>
      <input
        type="checkbox"
        checked={isChecked}
        onChange={(e) =>
          setIsChecked(e.target.checked)}
      />
      {''} I agree to the terms
    </label>
  );
}
```

Checkbox uses checked and onChange with a boolean value.

4 Select and Textarea

Use select dropdowns and textarea for larger inputs.

```jsx
function SelectInput() {
  const [country, setCountry] =
    useState('');
  return (
    <select value={country}
      onChange={(e) =>
        setCountry(e.target.value)}>
      <option value="">Choose a country</option>
      <option value="IN">India</option>
      <option value="US">USA</option>
      <option value="UK">UK</option>
    </select>
  );
}
```

```jsx
function TextArea() {
  const [message, setMessage] =
    useState('');
  return (
    <textarea
      value={message}
      onChange={(e) => setMessage(e.target.value)}
      rows={4}
      placeholder="Write your message..."
    />
  );
}
```

Use select for options and textarea for multi-line input.

Validate Guide Help Build Trust

Form Validation

Ensure Correct and Meaningful Input

Better Forms Happier Users

"Good validation prevents errors and improves user experience."

1 Field Validation

Validate input fields before submission.

```jsx
function validateEmail(email) {
  return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email);
}
function EmailInput() {
  const [email, setEmail] = useState('');
  const [error, setError] = useState('');

  const handleBlur = () => {
    if (!validateEmail(email)) {
      setError('Please enter a valid email');
    } else {
      setError('');
    }
  };

  return (
    <div>
      <input type="email" value={email}
      onChange={(e) => setEmail(e.target.value)}
      onBlur={handleBlur}
      placeholder="Enter your email" />
    {error && <p className="error">{error}</p>}
    </div>
  );
}
```

Validate on change, blur, or submit depending on your use case.

2 Error Messages

Show helpful error messages to users.

Email

john@example

Please enter a valid email address.

Password

••••••••

Password must be at least 6 characters.

Good error messages are:

- Clear and specific
- User-friendly
- Shown at the right time
- Accessible (screen reader friendly)

Avoid:

- Vague messages (e.g. "Invalid")
- Showing all errors at once
- Blocking without explanation

3 Form Submission

Handle form submission in React (using controlled inputs).

```jsx
function ContactForm() {
  const [formData, setFormData] =
    useState({
      name: '',
      email: '',
      message: ''
    });
  const handleSubmit = (e) => {
    e.preventDefault(); // prevent page reload
    console.log('Form submitted:', formData);
    // send data to API
  };
  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        placeholder="Name"
        value={formData.name}
        onChange={(e) => setFormData({
          ...formData,
          name: e.target.value
        })}
      <button type="submit">Send</button>
    </form>
  );
}
```

Use e.preventDefault() to handle form submission in React.

4 Accessible Feedback

Make forms accessible for all users (including screen readers).

```jsx
return (
  <div>
    <label htmlFor="email">Email</label>
    <input
      id="email"
      type="email"
      aria-invalid={!!error}
      aria-describedby="email-error"
      value={email}
      onChange={(e) => setEmail(e.target.value)}
    />
    {error && (
      <p id="email-error" role="alert"
        className="error">
        {error}
    )</p>
  </div>
);
```

Accessibility Tips:

- Use proper labels (htmlFor)
- Use aria-invalid for errors
- Use role="alert" for error messages
- Ensure keyboard navigation works

"Valid forms create confident users."

Clean Code Better Products

Write
Clean Code
Follow Rules
Build Better

Rules of React and Hooks

Same
Rules
Happier
Components

Follow the Rules, Build Better Apps

"These rules keep your components predictable and your app stable."

1 Purity

Components should be pure functions.

```jsx
let count = 0;

function Counter() {
  // ❌ Avoid side effects
  // in the render
  count = count + 1;
  return <h2>{count}</h2>;
}

// ✅ Pure component
function Counter() {
  return <h2>Hello React!</h2>;
}
```

A pure component always returns the same output for the same inputs and has no side effects.

2 Immutability

Never mutate state directly — always create a new value.

```jsx
function TodoList() {
  const [items, setItems] =
    useState(['Learn React']);

  // ❌ Mutating the array
  function addItemBad(item) {
    items.push(item);
    setItems(items);
  }

  // ✅ Create a new array
  function addItem(item) {
    setItems<[...items, item]);
  }
}
```

Immutability helps React detect changes and keeps your UI consistent.

3 Hook Rules

Only call hooks at the top level and from React functions.

```jsx
// ✅ Correct
function App() {
  const [count, setCount] =
  useState(0);
  useEffect(() => {
    // side effect
  }, []);
  return <div>Count: {count}</div>;
}
// ❌ Incorrect
function App() {
  if (condition) {
    const [count, setCount] =
    useState(0); // Don't do this
  }
  return null;
}
```

Always call hooks at the top level, never inside loops, conditions, or nested functions.

4 Strict Mode

Helps find problems early during development.

```jsx
import { StrictMode } from
 'react';
import { createRoot } from
 'react-dom/client';
import App from './App';

const root = createRoot(
 document.getElementById('root')
);

root.render(
  <StrictMode>
    <App />
  </StrictMode>
);
```

- ✅ *Highlights potential problems*
- ✅ *Runs components twice (in development)*
- ✅ *Helps identify unsafe side effects*

Sync
Update
Clean Up
Repeat

useEffect Basics

Side Effects
Made Simple
Keep Things
in Sync

Synchronize Your Component with the Outside World

"useEffect lets you perform side effects in functional components."

1 Synchronization

Use useEffect to sync with external systems like APIs, event listeners, or timers.

```jsx
import { useEffect, useState } from 'react';

function UserProfile() {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch('https://jsonplaccholder.typicode.com/users/1')
      .then(res => res.json())
      .then(data => setUser(data));
  }, []); // runs once on mount

  return (
    <div>
      <h2>{user ? user.name : 'Loading...'}</h2>
    </div>
  );
}
```

useEffect runs after the component renders and lets you synchronize with external systems.

2 Dependencies

Control when the effect runs using the dependency array.

```jsx
function Search({ query }) {
  const [results, setResults] = useState([]);

  useEffect(() => {
    if (!query) return;

    fetch('/api/search?q=${query}')
      .then(res => res.json())
      .then(data => setResults(data));
  }, [query]); // runs when 'query' changes

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

Include all values from your component that are used inside the effect.

3 Cleanup

Return a cleanup function to avoid memory leaks.

```jsx
function Clock() {
  const [time,setTime] = useState(new Date());

  useEffect(() => {
    const interval = setInterval(() => {
      setTime(new Date());
    }, 1000);

    // Cleanup
    return () => {
      clearInterval(interval);
    };
  }, []);

  return <h2>{time.toLocaleTimeString()}</h2>;
}
```

Cleanup runs before the component unmounts or before the effect runs again.

"Use effects wisely, and keep your app in sync with the real world."

Learn
Build
Improve
Repeat

Real Examples Real Understanding

Effects in Practice

Common Real-World Use Cases

"Effects let you connect your component to the outside world."

Observe Sync Clean Up Avoid Pitfalls Build Better Apps

1 Subscriptions

Use effects to subscribe to external data sources (e.g. WebSocket, event listeners).

```jsx
import { useEffect, useState }
from 'react';

function ChatRoom() {
  const [messages, setMessages] =
    useState([]);

  useEffect(() => {
    const socket = new
      WebSocket('wss://example.com');

    socket.onmessage = (e) => {
      setMessages((prev) => [
        ...prev,
        e.data,
      ]);
    };

    return () => socket.close();
  }, []);

  return <div>{messages.map((m, i) =>
    <p key={i}>{m}</p>
  ))}</div>;
}
```

Always clean up subscriptions in the return function.

2 Timers

Use effects for timers like setInterval and clear them on cleanup.

```jsx
import { useEffect, useState }
from 'react';

function Timer() {
  const [seconds, setSeconds] =
    useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setSeconds((s) => s + 1);
    }, 1000);

    return () => clearInterval(id);
  }, []);

  return <h2>Seconds: {seconds}</h2>;
};
```

Always clear timers to prevent memory leaks.

3 Stale Closures

Be careful with stale state inside effects. Use dependencies or functional updates.

```jsx
import { useEffect, useState }
from 'react';

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

  // ❌ Stale closure (count is 0)
  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1);
    }, 1000);
    return () => clearInterval(id);
  }, []);

  // ✅ Correct: use functional updater
  useEffect(() => {
    const id = setInterval(() => {
      setCount((c) => c + 1);
    }, 1000);
    return () => clearInterval(id);
  }, []);
}
```

Use functional updates or include all dependencies to avoid stale values.

4 Unnecessary Effects

Don't use effects for things that can be done during render (e.g. derived state).

```jsx
import { useState, useEffect }
from 'react';

function UserList({ users }) {
  const [filtered, setFiltered] =
    useState(users),

  // ❌ Unnecessary effect
  useEffect(() => {
    setFiltered(
      users.filter((u) => u.active)
    );
  }, [users]);

  // ✅ Do it directly during render
  const activeUsers = users.filter(
    (u) => u.active

  return (
    <ul>
      {activeUsers.map((u) => (
        <li key={u.id}>{u.name}</li>
      ))}
    </ul>
  );
```

Only use effects for side effects (like fetching, subscriptions, or DOM APIs).

Access Control Manipulate The DOM

useRef and DOM Access

Keep Values, Access Elements, Do More

"useRef lets you persist values and interact with DOM elements directly."

Small Powerful Useful In Many Cases

1 Persistent Values

useRef can hold values that persist across renders without causing re-renders.

```jsx
import { useRef } from 'react';

function RenderCount() {
  const countRef = useRef(0);

  countRef.current += 1;

  return <p>Rendered {countRef.current}
  times</p>;
```

Great for storing mutable values like previous state, IDs, or flags.

2 Focus Elements

Use refs to focus input elements programmatically.

```jsx
import { useRef } from 'react';

function FocusInput() {
  const inputRef = useRef(null);

  const handleClick = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef}
        type="text"
      placeholder="Click the button" />
      <button onClick={handleClick}>
        Focus Input
      </button>
    </div>
  );
};
```

Use refs to improve UX (e.g. focus, select, scroll).

3 Scrolling

Use refs to scroll to an element.

```jsx
import { useRef } from 'react';

function ScrollToSection() {
  const sectionRef = useRef(null);

  const scrollToSection = () => {
    sectionRef.current?.scrollIntoView({
      behavior: 'smooth',
    };

  return (
    <div>
      <button onClick={scrollToSection}>
        Go to Section
      </button>
      <div style={{ height: '500px' }} />
      <h2 ref={sectionRef}>Target Section</h2>
    </div>
  );
};
```

Useful for smooth scrolling and navigation.

4 Callback Refs

Use callback refs for more control (e.g. when you need to run code on element mount/unmount).

```jsx
import { useState } from 'react';

function CallbackRef() {
  const [node, setNode] = useState(null);

  const setRef = (element) => {
    setNode(element);
    if (element) {
      console.log('Element mounted:',
        element);
    } else {
      console.log('Element unmounted');
    };
  };

  return <div ref={setRef}>I'm a box</div>;
}
```

Callback refs are useful when you need to perform actions on mount or unmount.

"Effects and refs help you build interactive, real-world React applications."

Practice Build Improve Repeat

Manage Complex State With Ease
Think in Actions Not Mutations

useReducer

A Powerful Alternative to useState

"useReducer is great for complex state logic."

Predictable State
Scalable Apps
Better Structure

1 Reducers

A reducer is a function that returns the next state based on the current state and an action.

```jsx
function counterReducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {...state, count: state.count + 1};
    case 'decrement':
      return {...state, count: state.count - 1};
    case 'reset';
      return { count: 0 };
    default:
      return state;
  }
}
```

Reducers must be pure functions (no side effects).

2 Actions

Actions are plain objects that describe what happened.

```jsx
const initialState = { count: 0 };

function Counter() {
  const [state, dispatch] =
  useReducer(counterReducer, initialState);

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({
        type: 'increment'
      })}>+</button>
      <button onClick={() => dispatch({
        type: 'decrement'
      })}>-</button>
      <button onClick={() => dispatch({
        type: 'reset'
      })}>Reset</button>
    </div>
  );
}
```

Dispatch actions to trigger state transitions.

3 Complex State Transitions

useReducer is useful when state logic is complex or involves multiple sub-values.

- Handles complex state logic
- Keeps state updates predictable
- Great for forms, wizards, or nested state
- Easier to test than multiple useState calls

When to use useReducer?

- You have complex state logic
- State depends on multiple actions
- You want a more predictable structure
- You're managing state that grows over time

When not to use it?

- Simple state (use useState instead)
- Only a few independent values
- No complex transitions

Share State Across Components
No More Prop Drilling

Context API

Share Data Across Your Component Tree

"Context lets you pass data without passing props manually at every level."

Global Data
Cleaner Code
More Flexible Applications

1 createContext

Create a context to hold the data you want to share.

```jsx
import { createContext } from 'react';

// Create a context with a default value
export const ThemeContext =
  createContext('light');
```

The default value is used when no provider is found.

2 Providers

Wrap your components with a provider to supply the data.

```jsx
import { ThemeContext } from './ThemeContext';

function App() {
  const [theme, setTheme] = useState('light');

  return (
    <ThemeContext.Provider
      value={{ theme, setTheme }}
    >
      <Header />
      <Main />
    </ThemeContext.Provider>
  );
}
```

The provider makes the data available to all child components.

3 useContext

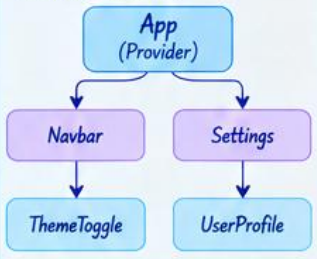
Use the context in any child component.

```jsx
import { useContext } from 'react';
import { ThemeContext } from
  './ThemeContext';
function Header() {
  const { theme, setTheme } =
    useContext(ThemeContext);

  return (
    <button
      onClick={() =>
        setTheme(theme === 'light'
        ? 'dark'
        : 'light')
      }
      Current theme: {theme}
    </button>
});
```

useContext gives you access to the nearest provider's value.

4 Avoiding Prop Drilling

Context helps you share data without passing props through many components.



- Cleaner component tree
- Easier to maintain
- Great for global data (theme, auth, etc.)
- Use wisely (avoid overusing context)

"useReducer for complex logic. Context for a more connected app."

Better Components
Happier Developers

Reuse
Logic
Write Less
Do More

Custom Hooks

Composable
Maintainable
Flexible
Better Apps

Extract Logic, Reuse Everywhere

"Custom hooks let you share stateful logic between components."

1 Reusable Logic

Extract and reuse stateful logic into a custom hook.

```jsx
import { useState, useEffect }
from 'react';

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

  const increment = () =>
    setCount(c => c + 1);
  const decrement = () =>
    setCount(c => c - 1);

  return { count, increment,
    decrement };
}

// Using the hook
function Counter() {
  const { count, increment } =
    useCounter(0);
  return <button onClick={increment}>
    Count: {count}
  </button>;
}
```

Custom hooks help you keep components clean and focused.

2 Naming

Custom hooks should start with "use" and describe what they do.

```jsx
// Good ✅
function useLocalStorage(key,
  intialValue) {
  const [value, setValue] =
    useState(() => {
      const saved =
      localStorage.getItem(key);
      return saved ? JSON.parse(saved)
        : initialValue;
    });

  useEffect(() => {
    localStorage.setItem(key,
      JSON.stringfy(value));
  }, [key, value]);

  return [value, setValue];
}

// Bad ❌
function useData() {
  // too generic, not descriptive
}
```

Use a clear, descriptive name that starts with "use".

3 Composition

Combine hooks to build more powerful hooks.

```jsx
import { useState, useEffect }
from 'react';
import { useLocalStorage } from
  './useLocalStorage';

function useOnlineStatus() {
  const [isOnline, setIsOnline] =
    useState(navigator.onLine);

  useEffect(() => {
    const handleOnline = () =>
      setIsOnline(true);
    const handleOffline = () =>
      setIsOnline(false);

    window.addEventListener('online',
      handleOnline);
    window.addEventListener('offline',
      handleOffline);
    return () => {
      window.removeEventListener('online',
        handleOnline);
      window.removeEventListener('offline',
        handleOffline);
    }
  }, []);
  return isOnline;
}
```

You can compose multiple hooks to solve complex problems.

4 useDebugValue

Helps you display a label for your custom hook in React DevTools.

```jsx
import { useState, useDebugValue }
from 'react';

function useUser(userId) {
  const [user, setUser] = useState(null);

  useDebugValue(user
    ? 'User: ${user.name}'
    : 'No user');

  useEffect(() => {
    // fetch user...
  }, [userId]);

  return user;
}

function Profile({ id }) {
  const user = useUser(id);
  return <div>{user ? user.name :
    'Loading...'}</div>;
}
```

useDebugValue is useful for debugging custom hooks in React DevTools.

Get Data
Handle States
Build Real Apps

Fetching API Data

Async
Data
Better UX
Real World
Ready

Load Data, Handle Errors, Keep It Real

"Fetching data is a core part of modern React applications."

1 Fetch API

Use the Fetch API to get data from an external source.

```jsx
const [data, setData] = useState([]);

useEffect(() => {
  async function fetchPosts() {
    try {
      const res = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );
      const json = await res.json();
      setData(json);
    } catch (error) {
      console.error('Error fetching
        posts:', error);
    }
  }
  fetchPosts();
}, []);
```

Use async/await for cleaner and more readable code.

2 Loading State

Show a loading indicator while the data is being fetched.

```jsx
const [data, setData] = useState(null);
const [loading, setLoading] =
  useState(true);

useEffect(() => {
  async function fetchData() {
    try {
      const res = await fetch('/api/users');
      const json = await res.json();
      setData(json);
    } finally {
      setLoading(false);
    }
  }
  fetchData();
}, []);

if (loading) return <p>Loading...</p>;
return <UserList users={data} />;
```

Loading states improve user experience.

3 Error Handling

Handle errors gracefully and show a helpful message.

```jsx
const [error, setError] = useState(null);

useEffect(() => {
  async function fetchData() {
    try {
      const res = await fetch('/api/data');
      if (!res.ok) throw new Error(
        'Failed to fetch data')
      const json = await res.json();
      setData(json);
    } catch (err) {
      setError(err.message);
    }
  }
  fetchData();
}, []);
if (error) return <p className="error">
  {error}</p>;
```

Always handle errors to avoid silent failures.

4 Cancellation & Race Conditions

Cancel ongoing requests and avoid race conditions.

```jsx
useEffect(() => {
  const controller = new AbortController();

  async function fetchData() {
    try {
      const res = await fetch('/api/search', {
        signal: controller.signal,
      });
      const json = await res.json();
      setResults(json);
    } catch (error) {
      if (error.name !== 'AbortError') {
        console.error('Fetch error:', error);
      }
    }
  }
  fetchData();
  return () => controller.abort(); // cancel
}, [query]);
```

Use AbortController to cancel requests and prevent stale data.

"Custom hooks and data fetching turn your React knowledge into real-world power."

Learn
Build
Improve
Repeat

Fresh Data
Better UX
Less Boilerplate
More Possibilities

Server State

Manage, Cache, and Sync Data from the Server

"Server state libraries make data fetching simple, powerful, and reliable."

Cache
Refetch
Mutate
Paginate
Build Scalable Apps

1 Caching

Cache server data to avoid unnecessary network requests.

```jsx
import { useQuery } from
'@tanstack/react-query';

function Users() {
  const { data, isLoading } =
    useQuery({
      queryKey: ['users'],
      queryFn: fetchUsers,
      staleTime: 5 * 60 * 1000,
    });

  if (isLoading) return <p>
    Loading...</p>;

  return (
    <ul>
      {data?.map(user => (
      <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

Cached data improves performance and provides a faster user experience.

2 Refetching

Refetch data when needed (e.g., on window focus, interval, or manually).

```jsx
import { useQuery } from
'@tanstack/react-query';

function Posts() {
  const { data, refetch,
    isFetching } = useQuery({
      queryKey: ['posts'],
      queryFn: fetchPosts,
      refetchOnWindowFocus: true,
      refetchInterval: 30000,
    });

  return (
    <div>
      <button onClick={refetch}>
        Refetch
      </button>
      {isFetching && <p>Updating...
      </p>}
    </div>
  );
}
```

Keep data fresh with automatic or manual refetching.

3 Mutations

Update, create, or delete data on the server.

```jsx
import { useMutation,
useQueryClient } from
'@tanstack/react-query';

function AddTodo() {
const queryClient = useQueryClient();

const mutation = useMutation({
  mutationFn: (newTodo) =>
    fetch('/api/todos', {
      method: 'POST',
      body: JSON.stringify(newTodo),
    }).then(res => res.json()),
  onSuccess: () => {
    queryClient.invalidateQueries({
      queryKey: ['todos'],
    });
  }
});

const handleAdd = (text) => {
  mutation.mutate({ text });
  };
}
```

After a mutation, invalidate or update the cache to keep UI in sync.

4 Pagination

Load data in pages for better performance.

```jsx
import { useQuery } from
'@tanstack/react-query';

function Users({ page = 1}) {
 const { data, isLoading } =
 useQuery({
   queryKey: ['users', page],
   queryFn: () =>
    fetch(`/api/users?page=${page}`)
     .then(res => res.json()),
   keepPreviousData: true,
 });
 return (
   <div>
    {/* Render users */}
    <button onClick={() =>
      setPage(p => p - 1)}
      disabled={page === 1}>
      Peutton>
    <button onClick={() =>
      setPage(p => p + 1)}>
      Next
   </button>
 );
```

Pagination helps handle large datasets efficiently.

5 Query Library Overview

Popular library: TanStack Query (React Query).

TanStack Query

- ✓ Simple and powerful API
- ✓ Caching and background refetching
- ✓ Mutations and optimistic updates
- ✓ Pagination and infinite queries
- ✓ DevTools for easier debugging
- ✓ Works great with React

Other options:

- SWR (by Vercel)
- RTK Query (Redux Toolkit)
- Apollo Client (GraphQL)

Use a query library to handle server state instead of building it from scratch.

Navigate
Build Flows
Create Layouts
Better Experiences

Routing

Build Multi-Page Experiences in Your React App

"React Router lets you create smooth, dynamic, and scalable navigation."

Routes
Links
Layouts
Navigation
Real Apps

1 Core Concepts

React Router enables client-side routing in single-page applications.

```jsx
import {
  BrowserRouter,
  Routes,
  Route
} from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={
        {<Home />} />
        <Route path="/about"
          element={<About />} />
    </BrowserRouter>
  );
```

Routes map URLs to components.

2 Routes

Define routes for different pages or components.

```jsx
import { Routes, Route }
from 'react-router-dom';

function App() {
 return (
   <Routes>
    <Route path= "/"
      element={<Home />} />
    <Route path="/products"
      element={<Products />} />
    <Route path="/products/:id"
      element={<Product />} />
    <Route path="*"
      element={<NotFound />} />
  </Routes>
);
```

Use route parameters (e.g., :id) for dynamic pages.

3 Links

Use the Link component for client-side navigation.

```jsx
import { Link } from
'react-router-dom';

function Navbar() {
 return (
   <nav>
    <Link to="/">Home</Link>
    <Link to="/about">About</Link>
    <Link to="/products">Products</Link>
  )</nav>
});
```

Home About Products

Use <Link> instead of <a> to prevent full page reloads.

4 Layouts

Create shared layouts using nested routes.

```jsx
import { Outlet, Link } from
'react-router-dom';

function Layout() {
 return (
   <div>
    <header>My App</header>
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
    </nav>
    <main>
      <Outlet />
    </main>
   </div>
 );
}
```

Use <Outlet /> to render child routes.

5 Navigation

Navigate programmatically (e.g., after form submission).

```jsx
import { useNavigate } from
'react-router-dom';

function Login() {
  const navigate = useNavigate();

  const handleLogin = async () => {
    // login logic
    navigate('/dashboard');
  };

  return (
    <button onClick={handleLogin}>
      Login
    </button>
  );
}
```

Use **useNavigate()** to redirect programmatically.

"Good routing turns your app into a real product."

Build
Navigate
Grow
Repeat

Advanced Routing

Better Routing Better Experiences

Dynamic Flexible Scalable Applications

Take Your Routes to the Next Level

"Advanced routing helps you build dynamic, scalable, and user-friendly applications."

1 Nested Routes

Create nested routes to render child components inside a parent layout.

```jsx
import {
  createBrowserRouter,
  Outlet
} from 'react-router-dom';

const router =
 createBrowserRouter([
  {
    path: '/',
    element: <Layout />,
    children: [
      { index: true,
        element: <Home /> },
      { path: 'about',
        element:<About />},
    ]
  }
]);

function Layout() {
  return (
    <div>
      <Navbar />
      <Outlet />
    </div>
  );
}
```

Use <Outlet /> to render child routes.

2 Route Parameters

Access dynamic values from the URL using useParams.

```jsx
import { useParams }
 from 'react-router-dom';

function User() {
  const { id } = useParams();

  return (
    <div>
      <h2>User Profile</h2>
      <p>User ID: {id}</p>
    </div>
  );
}
```

Example URL:

`/users/123`

Use route parameters for dynamic content (e.g., /users/:id).

3 Search Parameters

Read and update query parameters using useSearchParams.

```jsx
import { useSearchParams }
 from 'react-router-dom';

function Products() {
  const [searchParams,
  setSearchParams] =
  useSearchParams();

  const page = searchParams
  .get('page') || '1';

 return (
  <div>
    <h2>Products</h2>
    <p>Page: {page}</p>
    <button
      onClick={() =>
       setSearchParams({
        page:String(Number(page)
        + 1)
       })
    }
      Next Page
    </button>
  </div>
 );
}
```

Use search params for filters, pagination, and shareable links.

4 Lazy Routes

Use code splitting with React.lazy for better performance.

```jsx
import { lazy, Suspense }
 from 'react';
import { Routes, Route }
 from 'react-router-dom';

const Dashboard =
lazy(() => import('../pages/
Dashboard'));

function App() {
 return (
   <Suspense fallback={
     <div>Loading...</div>
   }>
     <Routes>
       <Route path="/"
       element={<Home />} />
       <Route path="/dashboard"
       element={<Dashboard />} />
     </Routes>
   </Suspense>
 );
}
```

Lazy load routes to reduce bundle size.

5 404 Pages

Handle unknown routes with a catch-all route.

```jsx
import { Routes, Route }
from 'react-router-dom';

function NotFound() {
 return (
  <div className="not-found">
    <h2>404</h2>
    <p>Page not found</p>
    <Link to="/">
      Go Home
    </Link>
  </div>
 );

function App() {
 return (
   <Routes>
     <Route path="/"
     element={<Home />} />
     <Route path="*"
     element={<NotFound/>} />
   </Routes>
 );
```

Use path="" to catch all unknown routes.*

Shared State Libraries

Share State Across Components Build Bigger Apps

Simple Predictable Scalable Maintainable

Manage Global State with Ease

"Use the right state management tool for the right problem."

1 Redux Toolkit Overview

Modern, simplified way to use Redux.

```jsx
import { configureStore, createSlice } from '@reduxjs/toolkit';
import { Provider, useSelector, useDispatch } from 'react-redux';

const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => { state.value += 1; },
    decrement: (state) => { state.value -= 1; },
  },
});

export const { increment, decrement } = counterSlice.actions;

const store = configureStore({
  reducer: { counter: counterSlice.reducer },
});
```

Redux Toolkit reduces boilerplate and makes Redux easier to use.

2 Zustand Overview

A lightweight and simple state management library.

```jsx
import { create } from 'zustand';

const useStore = create((set) => ({
  count: 0,
  increment: () => set((state) => ({
    count: state.count + 1),
  })),
  decrement: () => set((state) => ({
    count: state.count - 1),
  })),
}));

function Counter() {
  const { count, increment, decrement } = useStore();
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>+</button>
      <button onClick={decrement}>-</button>
    </div>
  );
}
```

Zustand is minimal, flexible, and great for smaller to medium apps.

3 Choosing State Location

Decide where to keep your state.

- ✅ *UI State (local) – use useState (for component-specific state)*
- ✅ *Shared UI State – use Context or Zustand (for simple global state)*
- ✅ *Complex Global State – use Redux Toolkit (for larger, complex apps)*
- ✅ *Server State – use React Query / TanStack Query (for fetching, caching, syncing)*
- ✅ *URL State – use React Router (search params) (for filters, pagination, etc.)*

"Choose the simplest tool that solves your problem today, and can grow with you tomorrow."

Performance Basics

Faster
Smoother
Better UX
Real Impact

Measure
Optimize
Eliminate Waste
Build Better

Build Faster and More Efficient React Apps

"Good performance makes your app feel great."

1 Measuring Renders

Use React DevTools to understand when and why components re-render.

```jsx
import { Profiler } from 'react';

function App() {
  const onRender = (
    id, phase, actualDuration
  ) => {
    console.log(
      `${id} (${phase}):
      ${actualDuration}ms`
    );
  };

  return (
    <Profiler id="App"
      onRender={onRender}>
      <MyComponent />
    </Profiler>
  );
}
```

Use the Profiler tab in React DevTools to find unnecessary re-renders.

2 State Placement

Keep state as local as possible and only lift it when needed.

```jsx
function Parent() {
  const [count, setCount] =
  useState(0);

  return (
    <div>
      <Child />
      <button onClick={() =>
       setCount(c => c + 1)}>
        Count: {count}
      </button>
    </div>
  );
}
// Better: keep state local
function Child() {
  const [value, setValue] =
  useState("");
  return <input value={value}
    onChange={(e) => setValue(e
    .target.value)} />;
```

Keep state close to where it's used to avoid re-renders across the tree.

3 React.memo

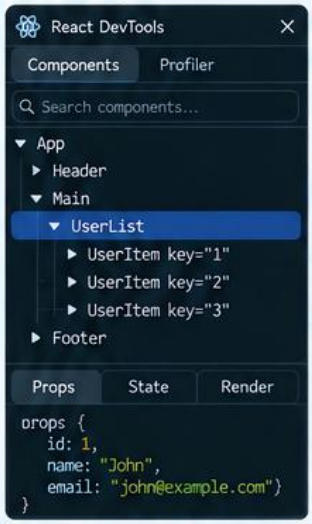
Use React.memo to prevent unnecessary re-renders of components.

```jsx
import { memo } from 'react';

const UserCard = memo(
  function UserCard({ user }) {
    console.log('UserCard rendered');
    return (
      <div className="card">
        <h3>{user.name}</h3>
        <p>{user.email}</p>
      </div>
    );
  );

function App() {
  const [user, setUser] =
    useState({ name: 'John',
    email: 'john@example.com' });

  return <UserCard user={user} />;
}
```

React.memo skips re-rendering if props haven't changed (shallow comparison).

4 React DevTools

Use React DevTools to inspect components, props, state, and render performance.



Use React DevTools to inspect component trees and analyze render performance.

Memoization and React Compiler

Optimize
Reduce Re-renders
Write Less Code
Do More

Smarter
Faster
Automatic
The Future

Optimize Today, Automate Tomorrow

"Memoization keeps your app fast. The React Compiler makes it easier."

1 useMemo

Use useMemo to memoize expensive calculations and avoid recalculating on every render.

```jsx
import { useMemo, useState } from 'react';

function ExpensiveList({ items }) {
  const filtered = useMemo(() => {
    console.log('Filtering...');
    return items.filter(item =>
      item.active);
  }, [items]);

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

useMemo is useful for expensive calculations or derived data.

2 useCallback

Use useCallback to memoize functions and prevent unnecessary re-renders in child components.

```jsx
import { useCallback, useState } from 'react';
import { memo } from 'react';

const Button = memo(({ onClick }) => {
  console.log('Button rendered');
  return <button onClick={onClick}>Click</button>;
});
function App() {
  const [count, setCount] = useState(0);
  const handleClick = useCallback(() => {
    setCount(c => c + 1);
  }, []); // stable reference

  return <Button onClick={handleClick} />;
}
```

useCallback returns a memoized function reference that only changes when dependencies change.

3 React Compiler

The React Compiler automatically optimizes your components, reducing the need for manual memoization.

React Compiler
Automatic Optimization

- Automatically memoizes components and values
- Reduces the need for useMemo and useCallback
- Improves performance with zero manual work
- Works at build time
- Lets you focus on writing great code

The React Compiler analyzes your code and applies optimizations automatically.

"Write efficient components today. Let the compiler make them even better tomorrow."

Better Performance
Happier Users

Responsive Updates

Keep Your App Fast and Responsive

Smooth Interactions
Better UX
Non-Blocking UI

"Concurrent features help you keep the UI responsive, even during heavy updates."

Responsive
Fast
Smooth
Modern React
Better Users

1 useTransition

useTransition lets you mark state updates as non-urgent, keeping the UI responsive.

```jsx
import { useState, useTransition }
from 'react';

function Search() {
  const [query, setQuery] = useState('');
  const [results, setResults] = useState([]);
  const [isPending, startTransition] =
   useTransition();

  function handleChange(e) {
    const value = e.target.value;
    setQuery(value);

    startTransition(() => {
      // Non-urgent update
      setResults(searchItems(value));
    });
  }
  return (
    <div>
      <input
        value={query}
        onChange={handleChange}
        placeholder="Search..."
      />
      {isPending && <p>Searching...</p>}
      <Results items={results} />
    </div>
  );
}
```

Use startTransition for non-urgent updates like filtering, searching, or large list rendering.

2 startTransition

startTransition is used to wrap non-urgent state updates.

```jsx
import { startTransition,
  useState } from 'react';

function handleFilter(value) {
  startTransition(() => {
    // Non-urgent state update
    setFilteredData(filterData(value));
  });
}

function App() {
  const [filter, setFilter] =
   useState('');

  return (
    <input
      value={filter}
      onChange={(e) => {
        setFilter(e.target.value);
        handleFilter(e.target.value);
      }}
      placeholder="Type to filter..."
    />
);
```

Wrap expensive updates with startTransition to keep the UI interactive.

When to use it?

- ✅ Filtering large lists
- ✅ Updating charts or complex UI
- ✅ Anything that can wait a little

3 useDeferredValue

useDeferredValue lets you defer a value and keep showing the previous value until the new one is ready.

```jsx
import { useState, useDeferredValue }
from 'react';

function Search() {
  const [query, setQuery] = useState('');
  const deferredQuery = useDeferredValue(query);
  const results = searchItems(deferredQuery);

  return (
    <div>
      <input
        value={query}
        onChange={(e) => setQuery(e.target.value)}
        placeholder="Search..."
      />
      <p>Showing results for: {deferredQuery}</p>
      <Results items={results} />
    </div>
  );
}
```

useDeferredValue is great for deferring fast-changing values like search input.

Key Benefits

- ❌ Keeps UI responsive
- ❌ Prevents lag during expensive renders
- ❌ Improves user experience

Code Splitting and Suspense

Load Less, Do More

Smaller Bundles
Faster Loads
Better Performance
Happier Users

"Split your code and load components on demand for faster, more efficient apps."

Dynamic
Lazy Loading
Loading States
Real World Apps

1 Dynamic Imports

Use dynamic imports to load modules only when needed.

```jsx
// Load a module dynamically
async function loadDashboard() {
  const module = await import('./pages/Dashboard');
  return module.Dashboard;
}

// Or with inline import
button.onclick = async () => {
  const { Dashboard } = await import('./Dashboard');
  // use Dashboard component
};
```

Dynamic imports help reduce your initial bundle size.

2 React.lazy

Use React.lazy to load components lazily with Suspense.

```jsx
import { lazy, Suspense } from 'react';

const Dashboard = lazy(() =>
  import('./pages/Dashboard')
);

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

React.lazy works great with Suspense to show a loading state.

3 Suspense and Loading Boundaries

Use Suspense to display a fallback UI while the component is loading.

```jsx
import { Suspense, lazy } from 'react';

const Settings = lazy(() =>
  import('./pages/Settings')
);

function App() {
  return (
    <div>
      <h1>My App</h1>
      <Suspense fallback={<div>Loading settings...</div>}>
        <Settings />
      </Suspense>
    </div>
  );
}
```

Use meaningful loading UI to improve the user experience.

"A faster app is a happier app. Load what you need, when you need it."

Build
Optimize
Ship
Repeat

Handling

Build Resilient Apps
Handle Errors
Improve UX
Keep Users Happy

Graceful Fallbacks
Better UX
More Reliable Apps

Catch Errors, Show Fallbacks, Keep Your App Running

"Errors happen. A great user experience handles them gracefully."

1 Error Boundaries

Catch JavaScript errors in child components and show a fallback UI.

```jsx
import { Component } from 'react';

class ErrorBoundary extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

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

  render() {
    if (this.state.hasError) {
      return <h2>Something went wrong.</h2>;
    }
    return this.props.children;
  }
}
```

Error boundaries catch errors in rendering, lifecycle methods, and constructors (not event handlers).

2 Fallback UI

Show a user-friendly UI when something goes wrong.

```jsx
function ErrorFallback({
  error,
  resetError
}) {
  return (
    <div className="error-box">
      <h2>Oops!</h2>
      <p>{error.message}</p>
      <button onClick={resetError}>
        Try Again
      </button>
    </div>
  );
}

// Usage (with a library like
// react-error-boundary)

<ErrorBoundary
  FallbackComponent={ErrorFallback}
  onReset={() => window.location.
    reload()}
>
  <App />
</ErrorBoundary>
```

Provide a helpful message and a way to retry or go back.

3 Retry Patterns

Let users retry the action after a failure.

```jsx
import { useState } from
'react';

function DataFetcher() {
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);
  const [loading, setLoading] = useState(false);
  const fetchData = async () => {
    try {
      setLoading(true);
      setError(null);
      const res = await fetch('/api/data');
      if (!res.ok) throw new Error('Failed');
      const json = await res.json();
      setData(json);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  };
  return (
    <div>
      {error && (
        <div>
          <p>{error}</p>
          <button onClick={fetchData}>
            Retry
          </button>
        </div>
      )}
```

Give users a clear way to retry failed actions (e.g., refetch data).

4 Limitations

Understand what error boundaries can and cannot catch.

Cannot catch:

- Errors in event handlers (e.g., onClick)
- Errors in async code (e.g., setTimeout, promises)
- Errors during server-side rendering (SSR)
- Errors inside the error boundary itself

Can catch:

- Errors in rendering
- Errors in lifecycle methods
- Errors in constructors (of child components)

Use try/catch for async code and event handlers. Combine with error boundaries for full coverage.

Advanced UI and Hooks

More Powerful UI
Better Control
Advanced Hooks
Build Complex Apps

Flexible UI
Unique IDs
Direct DOM Control
Reusable Components

Unlock More Possibilities with Advanced React APIs

"Advanced hooks and UI techniques give you the control to build exceptional experiences."

1 Portals

Render components outside the parent DOM hierarchy.

```jsx
import { createPortal } from
'react-dom';

function Modal({ children }) {
  return createPortal(
    <div className="modal">
      {children}
    </div>,
    document.getElementById('modal-root')
  );
}

// Usage
<Modal>
  <h2>Modal Content</h2>
</Modal>
```

Use portals for modals, tooltips, and other overlay UI elements.

2 useId

Generate unique IDs for accessibility (especially with forms).

```jsx
import { useId } from 'react';

function InputField() {
  const id = useId();
  return (
    <div>
      <label htmlFor={id}>Name:</label>
      <input id={id} type="text" />
    </div>
  );
}
```

useId ensures consistent, unique IDs across server and client renders.

3 useLayoutEffect

Run side effects synchronously after DOM updates (before paint).

```jsx
import { useRef, useLayoutEffect }
from 'react';

function Tooltip() {
  const ref = useRef(null);

  useLayoutEffect(() => {
    // Measure and position the tooltip
    const rect = ref.current.getBoundingClientRect();
    console.log('Tooltip position:', rect);
  }, []);

  return <div ref={ref}>Tooltip</div>;
}
```

Use useLayoutEffect for DOM measurements and visual updates to avoid flicker.

4 useImperativeHandle

Customize the value exposed by a ref to parent components.

```jsx
import { forwardRef,
  useImperativeHandle,
  useRef } from 'react';

const TextInput = forwardRef(
  (props, ref) => {
    const inputRef = useRef(null);
    useImperativeHandle(ref, () => ({
      focus: () => inputRef.current.focus(),
      clear: () => { inputRef.current.value = ''}
    }));
    return <input ref={inputRef} {...props} />;
  });
// Usage
const ref = useRef();
<TextInput ref={ref} />
<button onClick={() => ref.current.focus()}>
  Focus Input
</button>
```

useImperativeHandle lets you expose specific methods to parent components.

"Handle errors, leverage advanced hooks, and build UIs that stand out."

Learn
Build
Improve
Repeat

Advanced Hooks
Real Use Cases
Better Apps

Specialized Hooks

Use the Right Hook for the Right Problem

"These specialized hooks solve specific challenges and unlock advanced use cases."

Understand
Use Wisely
Solve Real Problems
Build Better Experiences

1 useSyncExternalStore

Subscribe to external data sources in a way that's compatible with concurrent rendering.

```jsx
import { useSyncExternalStore }
from 'react';
// A simple external store example
let listeners = new Set();
let value = 0;
const subscribe = (listener) => {
  listeners.add(listener);
  return () => listeners.delete(listener);
};
const getSnapshot = () => value;
const setValue = (newValue) => {
  value = newValue;
  listeners.forEach((l) => l());
};
function Counter() {
  const count = useSyncExternalStore(
    subscribe,
    getSnapshot
  );
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setValue(count + 1)}>
        Increment
      </button>
    </div>
  );
}
```

Use useSyncExternalStore for external stores (e.g., Zustand, Redux, browser APIs) to ensure consistent data with concurrent rendering.

2 useInsertionEffect

Run synchronous code before any DOM mutations. Mainly used by CSS-in-JS libraries.

```jsx
import { useInsertionEffect, useState }
from 'react';

function useInjectedStyles(cssText) {
  useInsertionEffect(() => {
    const style = document.createElement('style');
    style.textContent = cssText;
    document.head.appendChild(style);
    return () => {
      document.head.removeChild(style);
    };
  }, [cssText]);
}

function App() {
  useInjectedStyles(`
    .box { color: #6366f1; font-weight: bold; }
  `);

  return <div className="box">Styled with
    useInsertionEffect</div>;
}
```

useInsertionEffect is for libraries that inject styles (e.g., styled-components, Emotion). It runs before layout effects.

3 Appropriate Use Cases

These hooks are not for everyday use. Use them when you have a specific need.

When to use:

- **useSyncExternalStore**
  - *External state management (Redux, Zustand)*
  - *Browser APIs (localStorage, media queries)*
  - *Real-time data (WebSocket, etc.)*
- **useInsertionEffect**
  - *CSS-in-JS libraries*
  - *Injecting critical styles before DOM mutations*

When NOT to use:

- *Don't use for regular component state.*
- *Don't replace useEffect or useLayoutEffect unless you need the specific behavior.*
- *Avoid overusing — these are advanced tools.*
- *Only use when necessary for performance or integration reasons.*

Choose the right hook for your specific use case. Keep your code simple and only use advanced hooks when necessary.

Better Forms
Smoother UX
Modern React
Real World Ready

Modern Forms and Actions

Build Powerful, User-Friendly Forms with Modern React APIs

"Modern form APIs make handling form submission, pending states, and optimistic UI simple."

Less Boilerplate
Better UX
Optimistic UI
Modern Apps

1 Form Actions

Use form actions to handle form submissions (especially with Server Actions in React 19+).

```jsx
// Server Action (can be in a
// separate file with 'use server')
async function createUser(formData) {
  'use server';
  const name = formData.get('name');

  // Save to database...
  return { success: true };
}
export function UserForm() {
  return (
    <form action={createUser}>
      <input name="name"
      placeholder="Name" />
      <button type="submit">Create</button>
    </form>
  );
}
```

Form actions simplify form handling and work great with server-side logic.

2 useActionState

Track the state of a form action (e.g., success, error, pending).

```jsx
import { useActionState } from
'react';
async function submitForm(prevState,
  formData) {
  const name = formData.get('name');
  if (!name) {
    return { error: 'Name is required' };
  }
  // Save to database...
  return { success: true, message: 'User
  created' };
}
function App() {
  const [state, formAction] =
    useActionState(submitForm, {
      error: null,
      success: false,
      message: ''
    });
  return (
    <form action={formAction}>
    <input name="name" placeholder="Name" />
    <button type="submit">Submit</button>
    {state.error && <p className="error">{state.error}</p>}
    {state.success && <p className="success">{state.message}</p>}
  </form>
```

useActionState gives you form state like errors, success, and pending.

3 useFormStatus

Access the current form status (e.g., pending) inside form components.

```jsx
import { useFormStatus } from
'react-dom';

function SubmitButton() {
  const { pending } = useFormStatus();
  return (
    <button type="submit"
    disabled={pending}
    className="btn">
    {pending ? 'Submitting...' : 'Submit'}
    </button>
  );
}

function App() {
  return (
    <form action={createUser}>
      <input name="name" />
      <SubmitButton />
    </form>
  );
}
```

useFormStatus lets you show pending states and disable UI during submission.

4 useOptimistic

Show optimistic UI updates while the action is in progress.

```jsx
import { useOptimistic } from
'react';
function Comments() {
  const [comments, setComments] =
  useOptimistic(
    [],
    (state, newComment) => {
      ...state,
      { id: Date.now(), text: newComment,
      optimistic: true }
    }
  );
async function addComment(formData) {
  const text = formData.get('text');
  setComments(text); // optimistic update
  // await save to server...
}
return (
  <form action={addComment}>
    <input name="text" placeholder="Add a comment" />
    <button type="submit">Post</button>
    <ul>
      {comments.map((c => (
        <li key={c.id} style={{opacity: c.optimistic ? 0.6 : 1}}>
          {c.text}
        </li>
      ))}
    </ul>
  </form>
```

useOptimistic makes your app feel faster by updating the UI immediately.

"Better forms. Happier users. Modern React makes it possible."

Learn
Build
Ship
Repeat

Modern React APIs

Modern APIs Cleaner Code Brighter Future

Less Boilerplate More Flexibility Modern React Better DX

Simpler Patterns. More Powerful Apps.

"Modern React APIs reduce boilerplate and make your code more intuitive."

1 use

The use API lets you read resources like promises and contexts directly in your components (especially useful with Server Components).

```jsx
import { use } from 'react';

async function UserProfile({ userId }) {
  const user = use(fetchUser(userId));

  return (
    <div>
      <h2>{user.name}</h2>
      <p>{user.email}</p>
    </div>
  );
}
```

Use to read promises, contexts, or other resources directly during render (mainly with Server Components).

2 ref as a prop

In React 19, you can pass ref as a regular prop — no need for forwardRef in most cases.

```jsx
function MyInput({ ref, ...props }) {
  return (
    <input
      ref={ref}
      className="input"
      {...props}
    />
  );
}

// Usage
function App() {
  const inputRef = useRef(null);

  return (
    <div>
      <MyInput ref={inputRef}
        placeholder="Type here..." />
    </div>
  );
```

Pass ref directly as a prop. Simpler and more consistent with standard props.

3 Context Provider Shorthand

React 19 introduces a shorthand for context providers, making them cleaner and easier to use.

```jsx
import { createContext, useContext }
from 'react';

const ThemeContext = createContext('light');

function App() {
  return (
    <ThemeContext value="dark">
      <Toolbar />
    </ThemeContext>
  );
}

function Toolbar() {
  const theme = useContext(ThemeContext);
  return <button className={theme}>
    {theme === 'dark' ? 🌙 Dark' : ☀️Light'}
  </button>;
}
```

Use the context directly as a provider. It's cleaner and reduces boilerplate (React 19+).

React 19.2 Features

New Possibilities Better Performance Smoother UX The Future is Here

More Control More Insights More Power React 19.2

More Control. Better Performance. Smoother Experiences.

"React 19.2 brings powerful new features to build faster and more interactive apps."

1 Activity

The Activity component lets you toggle between UI states while preserving state and improving transitions.

```jsx
import { Activity } from 'react';

function App() {
  const [tab, setTab] = useState('feed');

  return (
    <div>
      <nav>
        <button onClick={() => setTab('feed')}>
        </button>
        <button onClick={() => setTab('profile')}>
        </button>
      </nav>
      <Activity mode={tab}>
        <Feed key="feed" />
        <Profile key="profile" />
      </Activity>
    </div>
  );
}
```

Use Activity to switch between UI states without losing state and for smoother transitions.

2 useEffectEvent

useEffectEvent lets you create stable event functions inside effects, avoiding unnecessary re-runs.

```jsx
import { useEffect, useEffectEvent }
from 'react';

function Page() {
  const onVisit = useEffectEvent(
    (url) => {
      console.log('Visited:', url);
    },);

  useEffect(() => {
    onVisit(window.location.pathname);
  }, [onVisit]);

  return <div>Page</div>;
}
```

Use useEffectEvent for stable event callbacks inside effects without extra re-renders.

3 cacheSignal

cacheSignal provides a way to integrate with caching mechanisms and cancel work when needed (e.g., data fetching).

```jsx
import { cacheSignal } from 'react';

async function getData(id, signal) {
  const res = await fetch(`/api/data/${id}`,
    { signal }
  );
  return res.json();
}

function useData(id) {
  const signal = cacheSignal();

  return use(fetchData(id, signal));
}
```

Use cacheSignal to cancel stale requests and integrate with caching systems.

4 Performance Tracks

Performance Tracks give you better insights into your app's performance with built-in tracking and DevTools integration.

```jsx
import { startPerformanceTrack }
from 'react';

function App() {
  function handleRender() {
    startPerformanceTrack('render', {
      detail: { screen: 'Home' }
    });
  }
  useEffect(() => {
    handleRender();
  }, []);

  return <div>Welcome!</div>;
}
```

Use Performance Tracks to measure and analyze performance in React DevTools.

"New APIs. Greater possibilities. Build the next generation with React."

Explore Experiment Build Repeat

Different Rendering Strategies
Better Performance
More Possibilities
Same Great React

Server Rendering Concepts

Same React. Different Ways to Render.

"Choose the right rendering strategy for your app's needs."

Faster Apps
Better SEO
Improved UX
More Control
Real Results

1 CSR (Client-Side Rendering)

The entire app is rendered in the browser using JavaScript.

Blank HTML + JS → JavaScript loads and renders the app in browser

```jsx
// main.jsx (CSR)
import { createRoot }
from 'react-dom/client';
import App from './App';

createRoot(
  document.getElementById(
    'root')
  ).render(<App />);
}
```

Simple to set up, great for highly interactive apps, but slower initial load and not SEO-friendly.

2 SSR (Server-Side Rendering)

HTML is rendered on the server for each request, then sent to the browser.

Server renders HTML → Send HTML to browser

```jsx
// e.g. Next.js (SSR)
export async function
getServerSideProps() {
  const data = await
    fetch('https://api...');
  return { props: { data } };
}

export default function
Page({ data }) {
  return <div>{data.title}</div>;
}
```

Good for dynamic content and SEO. Higher server cost and slightly slower response times.

3 Static Generation (SSG)

HTML is generated at build time and served as static files.

Build time generates HTML → Serve static files (CDN)

```jsx
// e.g. Next.js (SSG)
export async function
getStaticProps() {
  const data = await
    fetch('https://api...');
  return { props: { data } };
}

export default function
Page({ data }) {
  return <div>{data.title}</div>;
}
```

Super fast, scalable and SEO-friendly. Best for content that doesn't change often.

4 Streaming

HTML is sent in chunks as it's ready, improving perceived performance.

Stream HTML in chunks → Render progressively

```jsx
// e.g. React 18+ Streaming
import { renderToPipeableStream }
from 'react-dom/server';

const { pipe } = renderToPipeableStream(
  <App />,
  {
    onShellReady() {
      pipe(response);
    }
  }
);
```

Lets users see content faster, especially for larger pages.

5 Hydration

Attaches event listeners to server-rendered HTML to make it interactive.

Server HTML → Hydrate with JavaScript

```jsx
// Hydration (client side)
import { hydrateRoot }
from 'react-dom/client';
import App from './App';

hydrateRoot(
  document.getElementById(
    'root')
  <App />
);
```

Makes server-rendered HTML interactive by attaching event listeners on the client.

Less Client JS
More Performance
Cleaner Architecture
Modern React

Server Components

Do More on the Server. Send Less JavaScript.

"Server Components let you build faster, lighter, and more efficient React apps."

Powerful by Default
Flexible and Scalable
Works with Modern Frameworks
The Future is Here

1 Server vs Client Components

Server Components run on the server (by default). Client Components run in the browser (when interactivity is needed).

Server Component	Client Component
Runs on the server	Runs in the browser
No client-side JS	Handles user interaction
Can fetch data directly	Uses state, effects, events
Smaller bundle size	Included in client bundle

Use Server Components for data fetching and static content. Use Client Components only when you need interactivity.

2 "use client"

Add 'use client' at the top of a file to make it a Client Component (with access to hooks, state, etc).

```jsx
'use client';

import { useState } from 'react';

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

Use 'use client' when you need interactivity, state, or browser APIs.

3 "use server"

Use 'use server' in functions to run them on the server (e.g., form actions, mutations).

```jsx
'use server';

import { db } from '@/lib/db';

export async function createPost
(formData) {
  const title = formData.get('title');
  await db.post.create({
    data: { title }
  });
  return { success: true };
}
```

Use 'use server' for server-only logic like database operations or form actions.

4 Framework Integration

Server Components are supported in modern frameworks like Next.js (App Router), making it easy to build full-stack React apps.

- **Next.js** — Full support for Server Components (App Router, RSC, Actions)
- **Remix** — Supports Server Components and streaming.
- **Astro** — Islands architecture with server rendering
- **Other Frameworks** — The ecosystem is rapidly adopting Server Components

Use a framework like Next.js to get the best developer experience with Server Components.

"Render on the server. Interact on the client. Build for what's next."

Learn
Build
Experiment
Repeat

Testing

Test Early
Write Confident Code
Fewer Bugs
Better UX
Ship Faster

Build Reliable Apps with Confidence

"Good tests make refactoring easier and your app more maintainable."

Test
Refactor
Improve
Repeat
Be Confident

1 Component Tests

Test individual components to ensure they render and behave correctly.

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

test('renders button with text',
 () => {
  render(<Button>Click me</Button>);
  expect(
    screen.getByRole('button',
      { name: /click me/i })
  ).toBeInTheDocument();
});
```

Test what users see and do, not implementation details.

2 React Testing Library

A simple and effective testing library focused on user interactions.

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

test('calls onSearch when typing',
  () => {
  const onSearch = jest.fn();
  render(<SearchBox onSearch={onSearch}
/>);
  fireEvent.change(
    screen.getByPlaceholderText(
      'Search...'
    ),
    { target: { value: 'react' } }
  );
  expect(onSearch).toHaveBeenCalledWith(
    'react'
  );
});
```

Use queries like getByRole, getByLabelText, getByText, etc.

3 Mocking

Mock external modules, APIs or dependencies to isolate your tests.

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

jest.mock('axios');
const mockedAxios = axios as
  jest.Mocked<typeof axios>;

mockedAxios.get.mockResolvedValue({
  data: [{ id: 1, name: 'John' }]
});

test('displays users from API',
 async () => {
  render(<UserList />);
  expect(await screen.findByText
  ('John')).toBeInTheDocument();
});
```

Mock network requests, third-party libraries, and complex dependencies.

4 End-to-End Testing

Test complete user flows in the browser to ensure everything works together.

```ts
// Using Playwright
import { test, expect } from
 '@playwright/test';

test('user can login', async
 ({ page }) => {
  await page.goto('/login');
  await page.fill(
    'input[name="email"]',
    'user@example.com'
  );
  await page.fill(
    'input[name="password"]',
    'password123'
  );
 await page.click('button[type="submit"]');
 await expect(page).toHaveURL('/dashboard');
 await expect(page.getByText('Welcome'))
   .toBeVisible();
});
```

Use tools like Playwright or Cypress to test real user journeys across pages.

Debugging and Code Quality

Find Issues Early
Write Better Code
Use the Right Tools
Keep Improving
Be Proud

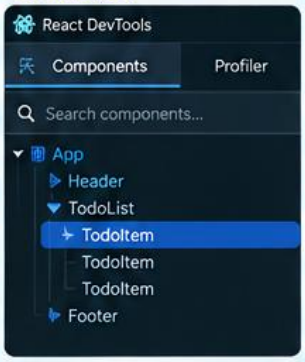
Write Cleaner Code. Find Issues Faster.

"Clean code and good tools make you a more productive developer."

Clean Code
Happy Developers
Better Projects
Fewer Headaches
More Time to Build

1 DevTools

Use React DevTools to inspect components, state, and render performance.



Inspect component hierarchy, props, state, and find unnecessary re-renders.

2 Linting

Use ESLint to catch errors, enforce best practices, and keep your code consistent.

```json
{
  "extends": [
    "eslint:recommended",
    "plugin:react/recommended",
    "plugin:react-hooks/recommended"
  ],
  "rules": {
    "react-hooks/exhaustive-deps":
      "warn",
    "no-unused-vars": "error"
  }
}
```

Use ESLint with React plugins to maintain code quality and avoid common mistakes.

3 Hook Dependencies

Always include the correct dependencies in useEffect, useMemo, and useCallback.

```tsx
import { useEffect } from 'react';

function UserProfile({ userId } :
  { userId: string }) {

  useEffect(() => {
    fetchUser(userId);
  }, [userId]); // ✅ Include deps

  return <div>...</div>;
}
```

Include all necessary dependencies to avoid stale data and unexpected bugs.

4 Common Mistakes

Be aware of common issues and how to fix them.

Common mistakes:

- Missing hook dependencies
- Mutating state directly
- Unnecessary re-renders
- Not cleaning up effects
- Using array index as key
- Overusing useEffect
- Forgetting error boundaries

Best practices:

- Follow the React docs
- Use proper linting rules
- Keep components small
- Write meaningful tests
- Use DevTools regularly

"Debug. Learn. Improve. Build better every day."

Better Code
Brighter Future

Secure Apps
Better Users
Protect Data
Build with Trust
Modern React

Authentication and Security Basics

Build Secure and Trustworthy React Apps

"Security isn't a feature, it's a foundation."

Secure Users
Protect Routes
Validate Data
Prevent Attacks
Build Safer Apps

1 Session-Aware UI

Show different UI based on authentication state (e.g., logged in or guest).

```jsx
import { useSession } from 'next-auth/react';

function Navbar() {
  const { data: session } = useSession();

  return (
    <nav>
      {session ? (
        <div>
          <span>Hi, {session.user.name}</span>
          <button>Sign Out</button>
        </div>
      ) : (
        <a href="/login">Sign In</a>
      )}
    </nav>
  );
}
```

Use session state to provide a personalized and secure user experience.

2 Protected Routes

Restrict access to certain pages or components based on authentication.

```jsx
import { useRouter } from 'next/navigation';
import { useSession } from 'next-auth/react';

function ProtectedPage({ children }: { children: React.ReactNode }) {
  const { data: session, status } = useSession();
  const router = useRouter();

  if (status === 'loading') return <p>Loading...</p>;

  if (!session) {
    router.push('/login');
    return null;
  }

  return <>{children}</>;
}
```

Protect routes on the client (and also on the server) to prevent unauthorized access.

3 Server-Side Authorization

Verify user permissions on the server before returning data or allowing actions.

```jsx
// e.g. Next.js API route
import { getServerSession } from 'next-auth';
import { authOptions } from '@/lib/auth';

export async function GET(req: Request) {
  const session = await getServerSession(authOptions);

  if (!session || session.user.role !== 'admin') {
    return new Response('Forbidden', { status: 403 });
  }

  // Return protected data
  return Response.json({ message: 'Secret data' });
}
```

Always enforce authorization on the server. Never rely only on client-side checks.

4 Safe Rendering

Prevent common security issues like XSS by safely rendering user content.

```jsx
import DOMPurify from 'dompurify';
import { useMemo } from 'react';

function UserContent({ content }: { content: string }) {
  const safeContent = useMemo(() =>
    DOMPurify.sanitize(content), [content]
  );

  return (
    <div dangerouslySetInnerHTML={{ __html: safeContent }} />
  );
}
```

Avoid rendering untrusted HTML directly. Use a sanitizer like DOMPurify to prevent XSS attacks.

Ship with Confidence
Optimize for Production
Scalable Apps
Real World Ready
Modern Tooling

Production and Deployment

From Code to the Real World

"A great app isn't complete until it's in the hands of users."

Build
Configure
Deploy
Monitor
Keep Improving

1 Builds

Create optimized production builds for your React app (e.g., with Vite or Next.js).

```bash
# Vite
npm run build

# Next.js
npm run build

# Output (Next.js)
✓ Compiled successfully
✓ Generating static pages
✓ Finalizing optimization
```

Production builds minify your code, remove dead code, and improve performance.

2 Environment Variables

Use environment variables to keep sensitive information out of your code.

```jsx
// .env.local
NEXT_PUBLIC_API_URL=https://api.example.com
DATABASE_URL=your-secret-key

// Access in code (Next.js)
const apiUrl = process.env.NEXT_PUBLIC_API_URL;

// Use in API route (server only)
const dbUrl = process.env.DATABASE_URL;
```

Use NEXT_PUBLIC_ only for variables safe to expose to the browser. Keep secrets on the server.

3 Asset Handling

Handle static assets like images, fonts, and other files efficiently.

```jsx
// Next.js example
import Image from 'next/image';

function Logo() {
  return (
    <Image
      src="/logo.png"
      alt="App logo"
      width={120}
      height={40}
      priority
    />
  );
}
```

Use optimized image components and place static assets in the public/ directory (or framework-specific folders).

4 Routing Configuration

Configure routes for your production app (e.g., with Next.js).

```
app/
  layout.tsx
  page.tsx
  about/
    page.tsx
  dashboard/
    page.tsx
  api/
    route.ts
```

```jsx
// Example route file
export default function About() {
  return <h1>About</h1>;
}
```

Organize routes clearly and follow your framework's conventions for scalable apps.

"Build. Deploy. Learn. Repeat."

Better Apps
Happier Users

Understand the Roots
Learn from the Past
Stronger Fundamentals
Better Developers
Modern Mindset

Legacy React

Still Relevant. Great for Understanding.

"Legacy React concepts help you understand how far React has come."

Same Concepts
New Perspective
Easier Migration
Stronger Skills
More Opportunities

1 Class Components

Class components are the older way to write React components using ES6 classes.

```jsx
import React, { Component }
from 'react';

class Welcome extends
 Component {
  render() {
    return (
     <div>
     <h1>Hello, {this.props.name}!
     </h1>
     </div>
    );
  }
}
```

Use class components to understand React's history and for maintainning older code.

2 Lifecycle Methods

Lifecycle methods let you run code at specific stages in a component's life.

```jsx
class Timer extends
Component {
  componentDidMount() {
    console.log('Mounted');
  }
  componentDidUpdate() {
    console.log('Updated');
  }
  componentWillUnmount() {
    console.log('Unmounting');
  }
  render() {
    return <h1>Timer</h1>;
  }}
```

Common lifecycle methods: componentDidMount, componentDidUpdate, componentWillUnmount.

3 Higher-Order Components (HOCs)

HOCs are functions that take a component and return a new component with extra props or behavior.

```jsx
const withLogger =
(WrappedComponent) => {
  return (props) => {
    console.log('Props:', props);
    return <WrappedComponent
      {...props} />;
  };
};

export default withLogger;
```

Use HOCs to reuse logic across multiple components (such as authentication, logging, etc.).

4 Render Props

Render props is a pattern where a component's child is a function that returns a React element.

```jsx
class MouseTracker
 extends Component {
 state = { x: 0, y: 0 };

 render() {
  return this.props.render(
    this.state
  );
 }

<MouseTracker
  render={({ x, y }) => (
   <p>Mouse position: {x}, {y}</p>
  )} />
```

Use render props to share stateful logic between components.

5 Migration Concepts

When working with older codebases, you may need to migrate class components to modern functional components with hooks.

- [x] Convert lifecycle methods to useEffect
- [x] Replace this.state with useState
- [x] Replace this.props with props
- [x] Migrate HOCs to custom hooks (when possible)
- [x] Test thoroughly after migration

Migrate gradually. Keep functionality the same and improve step by step.

Build Something Real
Apply What You Learn
Strengthen Your Skills
Track Your Progress
You Got This!

Mini Project: Task Manager

Plan. Build. Practice. Grow.

"A simple project that brings together the core concepts of React."

Small Project
Big Learning
Real Experience
Better Confidence
Keep Building

1 CRUD Operations

Create, read, update and delete tasks.

```jsx
const [tasks, setTasks] =
 useState([]);

// Add task
const addTask = (task) => {
  setTasks([...tasks, task]);
};

// Delete task
const deleteTask = (id) => {
  setTasks(tasks.filter(t =>
    t.id !== id));
};
```

Implement all CRUD operations to manage tasks effectively.

2 Filters

Add filters to view all, active or completed tasks.

```jsx
const [filter, setFilter] =
 useState('all');

const filteredTasks = tasks
  .filter(task => {
   if (filter === 'active')
    return !task.completed;
   if (filter === 'completed')
    return task.completed;
   return true;
  });
```

Use filters to make the app more useful and user-friendly.

3 Forms

Use forms to add and update tasks.

```jsx
const [input, setInput] =
 useState('');

const handleSubmit = (e) => {
  e.preventDefault();
  if (input.trim()) {
    addTask({
      id: Date.now(),
      text: input,
      completed: false
    });
    setInput('');
  }
};
```

Use controlled components to handle form inputs.

4 Local Storage

Save tasks in local storage so they persist after refresh.

```jsx
useEffect(() => {
  const saved = localStorage
   .getItem('tasks');
  if (saved) setTasks(JSON.parse
   (saved));
}, []);

useEffect(() => {
  localStorage.setItem(
   'tasks', JSON.stringify(tasks)
  );
}, [tasks]);
```

Use localStorage to persist tasks between sessions.

"Build projects. Turn knowledge into skills."

Learn
Build
Improve
Repeat

Build Real Apps
Use APIs
Solve Real Problems
Apply Your Knowledge
You've Got This!

Capstone: API Dashboard

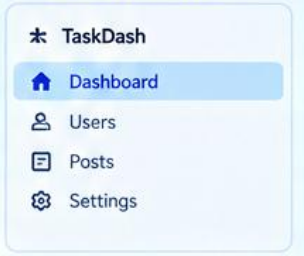
Bring It All Together

"A complete project to apply everything you've learned."

Plan
Build
Debug
Deploy
Be Proud

1 Routing

Set up multiple pages using React Router (or Next.js) for a real app structure.

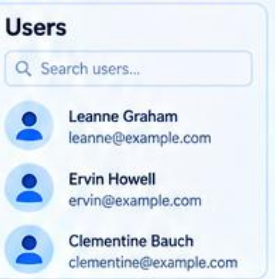



```jsx
import { BrowserRouter,
  Routes, Route, Link }
 from 'react-router-dom';

function App() {
 return (
  <BrowserRouter>
   <nav>
    <Link to="/">Dashboard</Link>
    <Link to="/users">Users</Link>
   </nav>
   <Routes>
    <Route path="/"
     element={<Dashboard />} />
    <Route path="/users"
      element={<Users />} />
   </Routes>
  </BrowserRouter>
 );
}
```

Organize your app with meaningful routes and a clean navigation structure.

2 Data Fetching

Fetch data from a real API (e.g.. JSONPlaceholder) and display it in your dashboard.

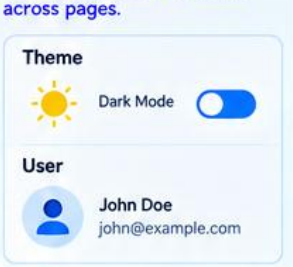



```jsx
import { useEffect,
 useState } from 'react';

function Users() {
 const [users, setUsers] =
   useState([]);

useEffect(() => {
 fetch('https://jsonplaceholder
  .typicode.com/users')
  .then(res => res.json())
  .then(setUsers)
  <catch(err =>
    console.error(err));
}, []);

return (
  <div>
    {users.map(user => (
     <UserCard key={user.id}
     user={user} />)
    ))}
  </div>
);
```

Use useEffect to fetch data, handle errors, and show real data in your UI.

3 Shared State

Use Context API (or a state library) to share data like user preferences or theme across pages.




```jsx
import { createContext,
 useContext, useState }
 from 'react';

const AppContext =
 createContext();

export function AppProvider
 ({ children }) {
 const [theme, setTheme] =
 useState('light');

 return (
  <AppContext.Provider
   value={{ theme,setTheme }}>
    {children}
  </AppContext.Provider>
 );
}

export const useApp = () =>
useContext(AppContext);
```

Share global state like theme, user info, or filters using Context API.

4 Loading and Error States

Show helpful loading indicators and error messages for a better user experience.


```jsx
function Posts() {
  const [posts, setPosts] =
  useState([]);
  const [loading, setLoading] =
  useState(troe);
  const [error, setError] =
  useState(null);

useEffect(() => {
  fetch('https://jsonplaceholder
   .typicode.com/posts')
   .then(res => res.json())
   .then(data => setPosts(data))
   .catch(err => setError(err))
   .finally(()=> setLoading(false));
}, []);

 if (loading) return <Loading />;
 if (error) return <Error message={
   "Failed to load posts."} />;

 return <PostsList posts={posts} />;
}
```

Handle loading and errors gracefully to keep your app reliable and user-friendly.

Review
Reinforce
Remember
Prepare
Keep Growing

Revision Cheat Sheet and Roadmap

Look Back. Practice More. Plan Ahead.

"A quick reference to strengthen your knowledge and guide your next steps."

Same Concepts
Bigger Dreams
You're Ready

1 Hooks Summary

Quick reference of commonly used React hooks.

- useState – manage state
- useEffect – side effects
- useContext – global state
- useReducer – complex state
- useRef – DOM references
- useMemo – memoize values
- useCallback – memoize functions
- useImperativeHandle – customize refs
- useLayoutEffect – sync with DOM

Keep this handy for quick recall during projects and interviews.

2 Common Patterns

Reusable patterns to write cleaner and scalable code.

- Container / Presentational
- Custom Hooks
- Higher-Order Components (HOCs)
- Render Props
- Compound Components
- Context for Global State
- Controlled vs Uncontrolled
- Component Composition
- Error Boundaries

Use these patterns to build maintainable and flexible apps.

3 Interview Questions

Be ready for technical interviews with these key topics.

- What is the Virtual DOM?
- Difference between CSR and SSR?
- How does useEffect work?
- What are keys in lists?
- What are HOCs and Render Props?
- How do you optimize re-renders?
- What is the Context API?
- How does React handle events?
- Explain reconciliation.
- How would you structure a large app?

Practice and explain concepts in your own words.

4 Next Steps

Continue your learning journey beyond the basics.

- [x] Build more real-world projects
- [x] Explore Next.js
- [x] Learn TypeScript deeply
- [x] Try state management (Zustand/Redux)
- [x] Learn testing (Jest, RTL)
- [x] Understand performance optimization
- [x] Contribute to open source
- [x] Build and deploy your portfolio
- [x] Keep learning and stay consistent!

The best way to learn is to keep building. You've got this!

"From concepts to confidence — your React journey continues!"

Learn
Build
Grow
Repeat

Great UI Makes Great Experiences ♡

Styling React Apps

Make Your Apps Look Amazing

"Style with purpose, build with confidence."

Clean Components Beautiful Interfaces Happier Users ☺

1 Regular CSS

Use regular CSS files to style your components.

```css
/* App.css */
container {
  padding: 20px;
  color: #333;
  background: #f5f5f5;
}
```

```jsx
/* App.jsx */
import './App.css';

function App() {
  return (
    <div className="container">
      <h1>Hello React</h1>
    </div>
  );
}
```

Import a CSS file and use className to apply styles.

2 Inline Styles

Use inline styles for quick, component-specific styling.

```jsx
function Button() {
  const styles = {
    backgroundColor: '#2563eb',
    color: 'white',
    padding: '10px 16px',
    border: 'none',
    borderRadius: '6px',
  };

  return (
    <button style={styles}>
      Click Me
    </button>
  );
}
```

Inline styles are written as a JavaScript object (camelCase).

3 CSS Modules

Use CSS Modules to scope styles to a component.

```css
/* Button.module.css */
.button {
  background: #10b981;
  color: white;
  padding: 8px 16px;
  border-radius: 6px;
}
```

```jsx
// Button.jsx
import styles from
  './Button.module.css';

export default function Button() {
  return (
    <button className={styles.button}>
      Click Me
    </button>
  );
}
```

CSS Modules keep styles local and avoid conflicts.

4 Conditional Classes

Apply classes conditionally based on state or props.

```jsx
function Button({ isActive }) {
  return (
    <button
      className={
        `btn ${isActive ? 'active' :
        'inactive'}`
      }
    >
      {isActive ? 'Active' : 'Inactive'}
    </button>
  );
}
```

```css
.btn { padding: 8px 16px; }
.active { background: #10b981; }
.inactive { background: #6b7280; }
```

Use template literals or libraries like clsx for complex conditions.

Better Components Better Products ♡

Component Design

Build Maintainable and Scalable UIs

"Good component design makes your app grow with ease."

Think Reusability Think Bigger ♡

1 Composition

Use composition to build flexible and reusable components.

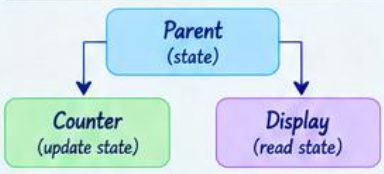
```jsx
function Card({ children, title }) {
  return (
    <div className="card">
      {title && <h3>{title}</h3>}
      <div className="card-content">
        {children}
      </div>
    </div>
  );
}

function App() {
  return (
    <Card title="Welcome">
      <p>This is a reusable card!</p>
    </Card>
  );
}
```

Composition lets you pass elements as children for more flexible UIs.

2 Lifting State Up

Move state to a common parent when multiple components need to share it.

```jsx
function Parent() {
  const [count, setCount] =
    useState(0);

  return (
    <div>
      <Counter count={count}
               setCount={setCount} />
      <Display count={count} />
    </div>
  );
}
```



Lift state up to keep components in sync.

3 Single Source of Truth

Keep one source of truth for each piece of state in your app.

```jsx
function TodoApp() {
  const [todos, setTodos] = useState([]);

  const addTodo = (text) => {
    setTodos([...todos, { id: Date.now(),
    text, completed: false }]);
  };

  return (
    <div>
      <h2>Todo List</h2>
      <TodoForm onAdd={addTodo} />
      <TodoList todos={todos} />
    </div>
  );
}
```

Each piece of state should have a single, clear source of truth to avoid bugs.

"Well-designed components turn ideas into powerful applications."

Build Learn Create Repeat ♡