React

Hooks · Components · Projects, explained in simple words

React JS Notes · easy to understand · free for students

1. Start with React

- **React** is a JavaScript library to build websites. You make small pieces called **components** and join them to make a full page.
- Think of LEGO blocks. Navbar, Card and Button are blocks. You build a block once and use it many times.
- When your data changes, React updates only that part of the screen. You do not touch the DOM yourself.
- Setup: `npm create vite@latest my-app -- --template react`, then `cd my-app`, `npm install`, `npm run dev`. Open `localhost:5173` in the browser.

What is inside the project folder

File or folder	Its job
index.html	One empty page. React fills it with your UI
src/main.jsx	Starting point. Puts the App component on the page
src/App.jsx	Your main component. Start editing here
src/components	Folder for your own components
package.json	List of libraries and run commands

IDEA A React website is a **tree** of components. App is at the top. Navbar, Home and Footer sit inside App. Card sits inside Home.

2. Components

- A **component** is a JavaScript function that returns UI. Its name must start with a capital letter.
- Use it like an HTML tag: `<Card />`. Write it once, use it as many times as you want.

EXAMPLE Your first component.

```
function Card() {
  return <div className="card"><h2>Hello React</h2></div>;
}
export default Card;
```

- Keep each component in its own file. Share it with `export default` and bring it in with `import Card from './Card'`.
- Components can sit inside other components. This is called **nesting**.

EXAMPLE Reusing one component two times.

```
function App() {
  return (
    <div>
      <Navbar />
      <Card />
      <Card />   // same block, used twice
    </div>
  );
}
```

- Always use **function components**. Class components are old and not needed for new projects.

3. JSX

- **JSX** looks like HTML written inside JavaScript. The browser cannot read it, so Vite converts it into normal JavaScript for you.

HTML	JSX
class	className
for	htmlFor
onclick	onClick (camelCase)
 	 (close every tag)
style="color:red"	style={{ color: 'red' }}

- Use curly braces `{ }` to put JavaScript inside JSX: `<h1>Hello {name}</h1>` or `{2 + 2}`.
- Return only **one parent** element. Wrap things in a `<div>` or an empty fragment `<> </>`.

➤ Show things on a condition

EXAMPLE Two ways to show or hide UI.

```
{isLoggedIn ? <p>Welcome back</p> : <p>Please login</p>}
{items.length > 0 && <p>You have items</p>}
```

Show a list

EXAMPLE Turn an array into list items with map.

```
{fruits.map(f => <li key={f}>{f}</li>)}
```

REMEMBER Every list item needs a **unique key**. It helps React know which item changed. Use an id, not the index, if the list can change order.

4. Props

- **Props** are the information a parent gives to a child. They work like arguments of a function.

IDEA A Product card is a template. Props fill in the name and price, so one component can show many different products.

EXAMPLE Same component, different props.

```
function Product({ name, price }) {
  return <p>{name} costs Rs {price}</p>;
}
// <Product name="Pen" price={10} />
// <Product name="Book" price={250} />
```

- Props are read-only. A child can use them but cannot change them.
- The special prop `children` is the content between the tags: `<Box>Hi</Box>`.
- Want the child to talk back to the parent? The parent sends a **function** as a prop and the child calls it.

5. State and useState

- **State** is the memory of a component. When state changes, React redraws that component.
- A normal variable will not update the screen. A state variable will.

SYNTAX useState gives you the value and a function to change it.

```
const [value, setValue] = useState(initialValue);
```

EXAMPLE Counter.

```
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <button onClick={() => setCount(count + 1)}>
      Clicked {count} times
    </button>
  );
}
```

	Props	State
Who gives it	Parent	The component itself
Can change?	No	Yes, using the setter
Example	Product name	Counter value

- Never change state directly. Wrong: `count = 5`. Right: `setCount(5)`.
- For arrays and objects, make a new copy: `setTodos([...todos, newTodo])`.
- If the new value depends on the old one, use `setCount(c => c + 1)`.

REMEMBER After `setCount(5)`, the variable `count` does not change in that same moment. The new value shows up in the next render.

6. Events and Forms

- Common events: `onClick`, `onChange`, `onSubmit`. Give the function name, do not call it: `onClick={handle}`.
- **Controlled input**: the input value comes from state, and `onChange` updates the state. React always knows what the user typed.

EXAMPLE Show what the user types.

```
const [name, setName] = useState("");
<input value={name} onChange={e => setName(e.target.value)} />
<p>You typed: {name}</p>
```

- On a form submit, call `e.preventDefault()` to stop the page from reloading.

7. useEffect Hook

- **Hooks** are special functions whose names start with `use`. Two rules: call them at the top level (not inside if or loops) and only inside components.
- **useEffect** runs your code after the screen is shown. Use it to fetch data, start timers or change the page title.

SYNTAX The list at the end decides when it runs.

```
useEffect(() => {
  // do the work here
  return () => { /* cleanup */ };
}, [dependencies]);
```

Dependency list	When the effect runs
[ ]	One time, when the component appears
[x]	First time, and again whenever x changes
not written	After every render (usually avoid this)

EXAMPLE Get data from an API one time.

```
const [users, setUsers] = useState([]);
useEffect(() => {
  fetch("https://jsonplaceholder.typicode.com/users")
    .then(res => res.json())
    .then(data => setUsers(data));
}, []);   // runs once
```

- The **cleanup** function tidies up (stop a timer, remove a listener) when the component goes away.

REMEMBER Forgetting `[ ]` in a fetch effect makes an infinite loop: fetch, set state, render, fetch again.

8. More Useful Hooks

Hook	Simple meaning	Use it for
useRef	A box that keeps a value without redrawing	Focus an input, keep a timer id
useContext	Data shared with the whole app	Theme, logged-in user
useReducer	useState for bigger logic	Many related updates (cart)
useMemo	Remember a calculated result	Heavy calculations
useCallback	Remember a function	Passing it to memoized child

➤ useRef example

EXAMPLE Click a button to focus the input.

```
const inputRef = useRef(null);
<input ref={inputRef} />
<button onClick={() => inputRef.current.focus()}>Focus</button>
```

➤ useContext in 3 steps

- **Step 1** Create: `const ThemeContext = createContext('light')`.
- **Step 2** Wrap your app: `<ThemeContext value="dark">` (older code: `ThemeContext.Provider`).

- **Step 3** Read anywhere inside: `const theme = useContext(ThemeContext)`.

Custom hook

- Your own hook that you can reuse in many components. The name must start with `use`.

EXAMPLE A reusable counter hook.

```
function useCounter(start = 0) {
  const [count, setCount] = useState(start);
  const increase = () => setCount(c => c + 1);
  return { count, increase };
}
// const { count, increase } = useCounter(5);
```

9. Projects to Build

Project	What you practice
1. Counter App	useState, events
2. Todo List	useState, lists, keys, forms
3. Weather App	useEffect, API fetch, loading and error state
4. Quiz App	State, props, conditional rendering
5. Shopping Cart	useContext or useReducer, components, props
6. Notes App	useEffect with localStorage, custom hook

Mini project, Todo List

- **Step 1** Make two states: `text` for the input and `todos` for the array.
- **Step 2** On Add, make a new array with the new todo and clear the input.
- **Step 3** Show the todos with `map` and a `key`.
- **Step 4** To delete, use `filter` to keep every todo except that one.

EXAMPLE Complete Todo app in one component.

```
function App() {
  const [text, setText] = useState("");
  const [todos, setTodos] = useState([]);
  const add = () => {
    if (!text.trim()) return;
    setTodos([...todos, { id: Date.now(), text }]);
    setText("");
  };
  const remove = id => setTodos(todos.filter(t => t.id !== id));
  return (
    <div>
      <input value={text} onChange={e => setText(e.target.value)} />
      <button onClick={add}>Add</button>
      {todos.map(t => (
        <p key={t.id}>{t.text} <button onClick={() => remove(t.id)}>X</button></p>
      ))}
    </div>
  );
}
```

TIP Start small, type the code yourself, then add features. Put the project on GitHub, deploy it on Vercel or Netlify and add a README with a screenshot. Recruiters love this.

10. Common Mistakes and Practice

Mistake	Fix
`todos.push(x)` on state	Use the setter with a new array
No key in a list	Add a unique key like id
`onClick={handle()}`	Write `onClick={handle}`
No `[ ]` in a fetch effect	Add the dependency list
Hook inside if or loop	Move it to the top level
Component name in small letters	Start with a capital letter

➤ Practice questions

1. What is a component? Why do we reuse components?
2. What is the difference between props and state?
3. What is JSX? Name three differences from HTML.
4. What does useState return? Why not change state directly?
5. Explain useEffect and its dependency list with examples.
6. When do you use useRef and useContext?
7. What is a custom hook? Write one.
8. Build a Todo app with add and delete.

- Revise one section daily and build one small project every week.