

# 髂 React Essential Training – Notes 😘



### 1. JSX – Writing HTML in JavaScript

#### Concept

JSX = JavaScript + XML. Lets you write HTML-like syntax inside JavaScript. React compiles this into React.createElement.

#### Example

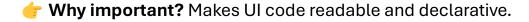
const element = <h1>Hello, React!</h1>;

#### **Behind the scenes**

const element = React.createElement("h1", null, "Hello, React!");

#### How it works

- JSX is just syntactic sugar.
- Must return one parent element (wrap children in <div> or <>).
- Values inside {} are pure JavaScript.



# 2. Props – Passing Data

### Concept

Props = inputs to a component. They make components reusable.

### Example

```
function Greeting({ name }) {
 return <h1>Hello {name}</h1>;
```

```
}
<Greeting name="Jinalee" />
```

#### **How it works**

- Greeting gets props.name = "Jinalee".
- {name} is replaced with value.
- Props are immutable the child cannot change them.
- **by Why important?** Lets you reuse the same component for different data.

### **3. State – Memory Inside Components**

#### Concept

State = a component's internal memory. Lets UI change without page reload.

### Example

```
import { useState } from "react";

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

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

#### How it works

- 1. useState(0) initializes count = 0.
- 2. setCount updates state.
- 3. Updating triggers a re-render with the new value.
- **the Why important?** Without state, your app is static.

### 4. Toggling State

#### Concept

Common pattern: switch true/false.

#### Example

```
const [isOn, setIsOn] = useState(false);

<button onClick={() => setIsOn(!isOn)}>
    {isOn ? "ON" : "OFF"}

</button>
```

#### **How it works**

- State starts as false.
- Clicking flips value.
- React re-renders, showing "ON" or "OFF".
- **the Why important?** Builds interactivity (menus, toggles, dark mode).

### 5. useReducer - Complex State

#### Concept

Alternative to useState. Better for complex updates.

#### Example

```
function reducer(state, action) {
  switch (action.type) {
    case "increment": return { count: state.count + 1 };
    case "decrement": return { count: state.count - 1 };
    default: return state;
  }
}
const [state, dispatch] = useReducer(reducer, { count: 0 });
```

#### **How it works**

- dispatch({type: "increment"}) triggers reducer.
- Reducer calculates new state.
- · React re-renders with updated data.

**Why important?** Keeps logic organized when multiple actions affect state.

### 6. useEffect – Handling Side Effects

### Concept

Side effects = things outside React's rendering (fetching data, timers, subscriptions).

### Example

```
useEffect(() => {
  console.log("Component mounted");
```

**}**, []);

#### **How it works**

- Runs after render.
- [] = run once on mount.
- [dependency] = runs again when dependency changes.

**Why important?** React only handles rendering; useEffect connects your app to the outside world.

## 7. Fetching Data

#### Concept

Combine useState and useEffect for data fetching.

#### Example

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

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

return {users.map(u => {u.name}}/ul>;
}
```

#### **How it works**

- 1. Initial render → empty list.
- 2. useEffect runs → fetch data.
- 3. Data stored in state.
- 4. React re-renders with updated UI.
- **Why important?** Modern apps almost always fetch from APIs.

### 8. Next.js Routing

#### Concept

File-based routing = every file becomes a page.

### Example

pages/about.js → /about

#### **How it works**

- · Next scans pages/ directory.
- Each file = route.
- Nested folders = nested routes.
- **Why important?** No need to configure routers manually.

### 9. Server vs Client Components

#### Concept

- **Server Components**: Rendered on server, no JS sent to client (fast, SEO-friendly).
- Client Components: Run in browser, handle events, state, interactivity.

#### How it works

- Use server components for heavy data fetching.
- Use client components for buttons, forms, state.
- **Why important?** Balances speed and interactivity.

### **■ 10. Tailwind CSS – Styling Made Easy**

#### Concept

Utility-first CSS. Apply styles directly as classes.

#### Example

<div className="bg-pink-200 text-lg p-4 rounded shadow">
Styled with Tailwind

</div>

#### How it works

- Classes are pre-made utilities (bg-pink-200 = background color).
- Combine small classes to design UI.
- **Why important?** No writing CSS files, speeds up UI design.

### 11. Advanced Next.js

#### **Concepts**

- Caching: Next caches results for faster loads.
- SSR (Server-Side Rendering): Generate at request → fresh data.
- SSG (Static Site Generation): Generate at build → fast for static pages.
- Server Actions: Run form submissions/data logic securely on server.

**Why important?** Professional apps rely on these patterns for speed and scalability.

### 🔭 Quick Map to Remember

- **Props** → External data passed to a component.
- State → Internal memory of a component.
- **useEffect** → Side effects (fetching, timers, APIs).
- **Reducer** → Complex state management.
- **Next.js** → Routing + server optimizations.
- **Tailwind** → Fast, utility-based styling.

#### So now you have:

- ✓ Deep explanations (like a textbook)
- Examples (code snippets)
- Step-by-step "how it works" (like flashcards)