React Props, Hooks, Refs, & Side Effects

Props & Arguments passing

To give arguments to the component, we use props.

- There are three different ways:

```
// 1. Access props through the parameter
const SlowText = (props) => {
  props.speed // access props.speed
  props.text  // access props.text

// 2. Destructure props directly
const SlowText = (props) => {
  const { speed, text } = props;

// 3. Destructure props directly 2
const SlowText = ({ speed, text }) => {
   ...
```

Using Components and Props

- Anything inside `{...}` is treated as a JavaScript expression (like variables, numbers, or function calls).
- If you write a string literal, you can use quotes directly ("..."), but when you're passing a variable or any computed value, you use {...} instead.
- If you wrote text="content", it would literally pass the string "content", not the value stored in the variable content.

```
<SlowText speed={20} text={content} />
<SlowText speed={20} text="content" />
```

JSON in {{...}}

The Outer { ... }

In JSX, anything inside { ... } means “insert a JavaScript expression here.”
So this:

```
<div style={{ background: "white" }} />
```

uses the outer braces to switch from JSX → JavaScript mode.

The Inner { ... }

Inside style={...}, React expects a JavaScript object, not a CSS string.

- In JavaScript, objects use { key: value } — like JSON but without quotes on keys.

So this is the inner object literal that defines CSS properties.

```
{ background: aiMessage ? "rgb(247, 247, 248)" : "white" }
```

Putting It Together

```
style={{ background: aiMessage ? "rgb(247, 247, 248)" : "white" }}
```

means:

1. The outer {} — JSX expression boundary.
2. The inner {} — JavaScript object for inline styles.

Equivalent Example

```
const bg = aiMessage ? "rgb(247, 247, 248)" : "white";
<div style={{ background: bg }} />
```

This works the same way — React expects style to receive an object, not a string.

React Hook & useState

The useState is a React Hook that lets a component remember values between renders — it stores "state."

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

- value -> the current state variable
- setValue -> the function to change that state
- initialValue -> the starting value when the component first renders

```
const [placeholder, setPlaceholder] = useState(text[0]);
```

```
function tick() {
  index.current++;                    // update the index stored in useRef
  console.log(placeholder);           // read the current displayed text
  setPlaceholder("He");               // update state → React re-renders the DOM
}
```

- placeholder is a state variable that will hold the currently displayed text (like “H” at first).
- setPlaceholder is the function you call later to update it (setPlaceholder("He"), etc.).
- useState(text[0]) initializes the state with the first character of text.

Functional Updater

```
setPlaceholder((prev) => prev + text[index.current]);
```

setPlaceholder can take:

1. a new value directly: `setPlaceholder("He");`
2. or a function that receives the previous state value:
   `setPlaceholder((prev) => prev + "l");`

That second form is called the functional updater form.

Why use the function form?

Because React state updates are asynchronous — React might delay or batch them.

- If you simply did this, then placeholder might still hold an old value when React processes it, especially if multiple updates happen close together.

```
setPlaceholder(placeholder + text[index.current]);
```

- So React gives us the previous (latest) value safely via the prev parameter.

```
setPlaceholder((prev) => prev + text[index.current]);
```

useRef & Persistent variable

The useRef is like a “persistent variable” that doesn’t reset or cause re-renders; It is perfect for things like indexes, timers, and DOM references.

```
const index = useRef(0); // create index ref
index.current++; // use it with .current property
```

- useRef() creates a persistent object with a .current property.
- It keeps its value even when the component re-renders, but changing it does not cause re-rendering.

So index.current behaves like a hidden box that React does not watch for updates.

Why not let index = 0;?

Because every time the component re-renders,

let index = 0; runs again — resetting it to 0.

You'd lose your progress (the typing index).

```
let index = 0; // resets every render
```

Why not useState?

We could use useState(index), but:

- Every update with setIndex() would cause a re-render.
- In this typing effect, the index is used only internally (not for rendering).
- Re-rendering that often would be unnecessary and slower.

useEffect & Side Effect

useEffect() runs side effects in React — things that happen outside rendering, like timers, network requests, or event listeners.

- Component renders with name = "Sam"

```
const [name, setName] = useState("Sam");
onClick={() => setName("Cho")}
```

- useEffect runs prints “Hello, Sam”.

```
useEffect(() => {
  console.log("Hello, " + name);
}, [name]); // Run when 'name' changes
```

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

function Hello() {
  const [name, setName] = useState("Sam");

  useEffect(() => {
    console.log("Hello, " + name);
  }, [name]); // Run when 'name' changes

  return (
    <button onClick={() => setName("Cho")}>
      Change Name
    </button>
  );
}

export default Hello;
```

When you click the button:

- setName("Cho") updates the state.
- React re-renders.
- useEffect runs again → prints “Hello, Cho”.

Example: Use useEffect for Animation

```
useEffect(() => {
  function tick() {
    // 1 Move to next letter
    index.current++;
    // 2 Add it
    setPlaceholder((prev) => prev + text[index.current]);
  }

  if (index.current < text.length - 1) {
    // 3 Repeat every few ms
    let addChar = setInterval(tick, speed);
    // 4 Stop old timers
    return () => clearInterval(addChar);
  }
}, [placeholder, speed, text]);
```

Step-by-Step Animation Flow

1. Initial render:

```
const [placeholder, setPlaceholder] = useState(text[0]);
```

2. useEffect runs → starts a timer (setInterval).

```
if (index.current < text.length - 1) {
  // 3 Repeat every few ms
  // addChar contains the old timer
  let addChar = setInterval(tick, speed);
  // 4 Stop old timers
  return () => clearInterval(addChar);
}
```

```
let addChar = setInterval(tick, speed);
```

setInterval(tick, speed) Tells the browser:
“Run the tick() function every speed milliseconds.”

- It keeps calling tick() repeatedly until stopped.>

```
// addChar is an old timer
return () => clearInterval(addChar)
```

It ensures no duplicate timers keep running in the background.

- This is the cleanup function for useEffect.
- React calls it before: the effect runs again (next render), or the component is removed (unmounted).

```
useEffect(() => {
  function tick() {...}
  if (index.current < text.length - 1) {...}
}, [placeholder, speed, text]);
```

Every time something important changes (like placeholder or text), React stops the old timer and starts a fresh one — keeping the animation clean, accurate, and in sync.

1. For example, the tick() updates the `placeholder` to run useEffect again repeatedly.
2. User can change `text`.

The useEffect() makes it easy and effective.

3. Every speed ms:

```
function tick() {
  index.current++;  // 1 Move to next letter
  setPlaceholder((prev) => prev + text[index.current]); // 2 Add it
}
```

- index.current++ → move to next character
- setPlaceholder() → add next letter to text
- React re-renders UI → screen updates

```
return () => clearInterval(addChar);
```

4. Each update triggers the effect again,
   clears the old timer, starts a new one.

5. When all letters are shown,
   if condition fails → timer stops.

className and .css Files

className in React

- In HTML, we normally use class="...".
- In React JSX, we use className instead — because class is a reserved JavaScript keyword.

So, we need to compile this JS into HTML:

```
<p className="new_chat_button_text">New Chat</p>
```

becomes

```
<p className="new_chat_button_text">New Chat</p>
```

`className="new_chat_button_text"` connects this `<p>` tag to a CSS selector defined in your stylesheet (e.g., dashboard.css).

```
.new_chat_button_icon {
    margin-top: 4px;
    margin-left: 10px;
}
```

1. The CSS file (e.g. import "./dashboard.css";) is imported once in your component or root file.
2. When the app runs, React (via the bundler) injects those styles into the page's `<style>` section.
3. Any element with a matching className automatically gets those styles applied.

Nested className

We can nest `<div>` elements (or any HTML/JSX elements) and assign each one its own className freely in React; this is is standard practice in structuring React components.

```
const Chat = () => {
  return (
    <div className="chat_container">
      <div className="chat_selected_container">
        <Messages />
      </div>
    </div>
  );
};
const Messages = () => {
  return <div></div>;
};
```

The rendered HTML becomes as follows:

```
<div class="chat_container">
  <div class="chat_selected_container">
    <div></div> <!-- rendered by Messages -->
  </div>
</div>

So, this CSS will be applied.

```css
.chat_container .chat_selected_container {
  border: 1px solid white;
}
```
```

In this DOM structure

```
chat_container
└── chat_selected_container
    └── messages_container
```

This means:

- .messages_container is inside .chat_selected_container, and .> chat_selected_container is inside .chat_container.

So, the DOM structure is true in both cases:

```
.chat_container .chat_selected_container .messages_container { ... }
.chat_selected_container .messages_container { ... }
```

Ai Icon

AiOutlinePlus is a React component from the react-icons/ai library.

- The prefix Ai = Ant Design Icons set.
- OutlinePlus = a plus symbol outline.
- It renders an SVG icon, just like an `<img>` or inline SVG.
- The prop color="white" sets the icon's fill color.

```
      <div className="new_chat_button_icon">
        <AiOutlinePlus color="white" />
      </div>
```

It renders a small white "+" symbol.

Event Handler

When we need to make a button with an icon & character such as this one.

We can have this React component:

```
import React from "react";
import { AiOutlinePlus } from "react-icons/ai";

const NewChatButton = () => {
  return (
    <div className="new_chat_button" onClick={() => {}}>
      <div className="new_chat_button_icon">
        <AiOutlinePlus color="white" />
      </div>
      <p className="new_chat_button_text">New Chat</p>
    </div>
  );
};
```

```
<div className="new_chat_button" onClick={() => {}}>
```

- The onClick is a React event handler for mouse clicks: It works the same way as the browser's onclick event, but in React it/s written in camelCase (onClick, not onclick).
- {} braces mean: You're embedding a JavaScript expression inside JSX.
- The value inside must be a function reference or function expression: () => {} is an arrow function that currently does nothing.

Example of Event Handler

You can replace the empty function with something meaningful:

```
<div
  className="new_chat_button"
  onClick={() => {console.log("New chat started!");}}
>
```

This version receives a function from props, allowing parent components to decide what happens on click (for example, opening a new chat window).

```
const NewChatButton = ({ onNewChat }) => (
  <div className="new_chat_button" onClick={onNewChat}>
    ...
  </div>
);
```

React Components - Layout Container

The React components can have other components.

- The App.js has Dashboard component in the ./Dashboard directory.

```
import Dashboard from "./Dashboard/Dashboard";

function App() {
  return (
    <div className="App">
      <Dashboard />
    </div>
  );
}
```

- The Dashboard component acts as the layout container.
    - The dashboard component has Sidebar and Chat componets.

```
import React from "react";
import Sidebar from "./Sidebar/Sidebar";
import Chat from "./Chat/Chat";

import "./dashboard.css";

const Dashboard = () => {
  return (
    <div className="dashboard_container">
      <Sidebar />
      <Chat />
    </div>
  );
};
```

Advantages of Structured React Components

We can manage React components in the structured directory.

Advantage	Description
Modular	Easier to maintain and reuse components
Clear structure	Directory hierarchy matches UI hierarchy
Scalable	Works well as the app grows — new features fit naturally

```
src
├── App.js
├── Dashboard
│   ├── Chat
│   │   ├── Chat.js
│   │   ├── Message.js
│   │   ├── Messages.js
│   │   └── NewMessageInput.js
│   ├── dashboard.css
│   ├── Dashboard.js
│   └── Sidebar
│       ├── DeleteConversationsButton.js
│       ├── ListItem.js
│       ├── NewChatButton.js
│       └── Sidebar.js
├── index.css
└── index.js
```