1. what is prop drilling

**ChatGPT said:**

Alright — let’s make **prop drilling** so simple that even someone who’s never touched React can get it.

**Prop Drilling — Layman Explanation**

Think of **props** in React as **passing notes between people** in a line.  
If the note needs to reach the last person, but you can only hand it to the next person in line, **everyone in between** has to pass it along — even if they don’t care about the note.

That **passing through people who don’t need it** is called **prop drilling**.

**Visual Example**

Grandparent → Parent → Child → Baby

We want to send favoriteColor = "blue" from **Grandparent** to **Baby**.  
But in React, the only way is to pass it through Parent and Child:

function Baby({ favoriteColor }) {

return <h2>Baby loves {favoriteColor}</h2>;

}

function Child({ favoriteColor }) {

return <Baby favoriteColor={favoriteColor} />;

}

function Parent({ favoriteColor }) {

return <Child favoriteColor={favoriteColor} />;

}

export default function Grandparent() {

return <Parent favoriteColor="blue" />;

}

* Here, Parent and Child **don’t actually care** about favoriteColor — they just pass it along.
* That’s **prop drilling**.

**Why is it a problem?**

1. Code becomes **messy** — you pass props everywhere.
2. Hard to **maintain** — if the middle components change, you might break the flow.
3. In big apps, it’s **easy to get lost** in the chain.

**How to avoid it?**

We can use:

* **React Context API** → store data in a “global storeroom” so components can grab it directly.
* State management tools like **Redux**, **Zustand**, **MobX**.

**Fix with Context API**

import React, { createContext, useContext } from "react";

const ColorContext = createContext();

function Baby() {

const favoriteColor = useContext(ColorContext);

return <h2>Baby loves {favoriteColor}</h2>;

}

function Child() {

return <Baby />;

}

function Parent() {

return <Child />;

}

export default function Grandparent() {

return (

<ColorContext.Provider value="blue">

<Parent />

</ColorContext.Provider>

);

}

Now, **no passing through Parent & Child** — Baby just gets it from the **context**.

If you like, I can draw you a **diagram showing prop drilling vs. context** so you instantly get the difference without reading all this again. Would you like me to do that?

1. ReactHooks

React hooks examples

Got it — I’ll walk you through **React hooks** with simple examples so it feels like learning with LEGO bricks.

## ****1.**** useState ****– Remembering Things****

Think of useState as a **sticky note** inside your component.  
It stores a value and lets you change it, and when you change it, React re-renders the component.

jsx

CopyEdit

import React, { useState } from "react";

export default function Counter() {

const [count, setCount] = useState(0); // count=0 initially

return (

<div>

<p>You clicked {count} times</p>

<button onClick={() => setCount(count + 1)}>Click me</button>

</div>

);

}

📌 Every click updates the count **and** redraws the component with the new value.

## ****2.**** useEffect ****– Doing Things After Render****

useEffect is like telling React:  
"After you build the house (render), go water the plants (run side effects)".

Example: Fetching data after the component loads.

jsx

CopyEdit

import React, { useEffect, useState } from "react";

export default function Users() {

const [users, setUsers] = useState([]);

useEffect(() => {

fetch("https://jsonplaceholder.typicode.com/users")

.then(res => res.json())

.then(data => setUsers(data));

}, []); // [] = run once after first render

return (

<ul>

{users.map(u => <li key={u.id}>{u.name}</li>)}

</ul>

);

}

📌 Here, the API is called **once** after the first render.

## ****3.**** useContext ****– No More Prop Drilling****

useContext lets you **pull data from a Provider** without passing props.

jsx

CopyEdit

import React, { createContext, useContext } from "react";

const ThemeContext = createContext();

function Button() {

const theme = useContext(ThemeContext);

return <button style={{ background: theme }}>Click me</button>;

}

export default function App() {

return (

<ThemeContext.Provider value="lightblue">

<Button />

</ThemeContext.Provider>

);

}

📌 Button doesn’t get theme from props — it reads it directly from the ThemeContext.

## ****4.**** useRef ****– Keeping Track Without Re-rendering****

useRef is like a **box** that can store a value and not lose it when the component updates — and it won’t trigger a re-render.

Example: Focusing on an input.

jsx

CopyEdit

import React, { useRef } from "react";

export default function FocusInput() {

const inputRef = useRef();

const focusInput = () => {

inputRef.current.focus();

};

return (

<div>

<input ref={inputRef} />

<button onClick={focusInput}>Focus the input</button>

</div>

);

}

## ****5.**** useMemo ****– Remember Expensive Calculations****

useMemo is like saying:  
"Only re-calculate this if the inputs change".

jsx

CopyEdit

import React, { useMemo, useState } from "react";

export default function ExpensiveCalc() {

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

const expensiveValue = useMemo(() => {

console.log("Calculating...");

return count \* 2;

}, [count]);

return (

<div>

<p>Value: {expensiveValue}</p>

<button onClick={() => setCount(count + 1)}>Increase</button>

</div>

);

}

📌 Prevents recalculation unless count changes.

## ****6.**** useCallback ****– Stable Function References****

Stops functions from being **re-created** every render — useful when passing callbacks to child components.

jsx

CopyEdit

import React, { useState, useCallback } from "react";

function Child({ onClick }) {

console.log("Child rendered");

return <button onClick={onClick}>Child Button</button>;

}

export default function Parent() {

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

const memoizedClick = useCallback(() => {

console.log("Clicked");

}, []);

return (

<>

<p>{count}</p>

<button onClick={() => setCount(count + 1)}>Increase</button>

<Child onClick={memoizedClick} />

</>

);

}

If you want, I can make a **single React app** that demonstrates **all these hooks together** so you can see them in action without jumping between files.  
That way you’ll have a **playground** for hooks learning.

##### You said:

}, []); // [] = run once after first render explain this in useEffect

###### ChatGPT said:

Alright — let’s break this down like we’re explaining it to a curious friend who’s never touched React.

In your code:

jsx

CopyEdit

useEffect(() => {

// side effect here

}, []);

### useEffect ****Recap****

useEffect runs **after** the component renders.  
It’s like telling React:

"Once you finish painting the house, come back and install the curtains."

### ****What the**** [] ****Means****

That empty array [] is called the **dependency array**.

It tells React:

* **"Only run this effect if these values change."**
* If the array is **empty**, there’s **nothing to watch** — so the effect runs **just once** after the very first render.

### ****Why Run Only Once?****

Some actions should happen only when the component **first appears**, not every time it updates.  
Example:

* Fetching data from an API when the page loads
* Setting up a subscription or event listener

### ****Timeline Example****

Let’s imagine:

1. Component renders for the **first time** → Effect runs.
2. You update state (re-render) → Effect **does NOT** run again because [] says “no dependencies”.
3. The effect runs again **only if** the component **unmounts & mounts again**.

### ****With and Without**** []

* **No array** → Runs **after every render**

useEffect(() => { console.log("Runs every time"); });

* **Empty array []** → Runs **once** after first render

useEffect(() => { console.log("Runs only once"); }, []);

* **Array with values** → Runs when **those values change**

jsx

CopyEdit

useEffect(() => { console.log("Runs when count changes"); }, [count]);

### ****Mini Example****

import React, { useState, useEffect } from "react";

export default function App() {

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

useEffect(() => {

console.log("I run only once after first render!");

}, []); // Empty dependency array

return (

<div>

<p>{count}</p>

<button onClick={() => setCount(count + 1)}>Increase</button>

</div>

);

}

Even if you click the button 100 times, that console log will appear **just once**.