

## What is an “Effect” (or a “Side Effect”)?

Main Job: Render UI & React to User Input

Evaluate & Render JSX  
Manage State & Props  
React to (User) Events & Input  
Re-evaluate Component upon State &  
Prop Changes

This all is “baked into” React via the “tools”  
and features covered in this course (i.e.  
useState() Hook, Props etc).

Side Effects: Anything Else

Store Data in Browser Storage  
Send Http Requests to Backend Servers  
Set & Manage Timers  
...

These tasks **must happen outside of the normal component evaluation** and render cycle – especially since they might block/delay rendering (e.g. Http requests)

## Handling Side Effects with the useEffect() Hook

```
useEffect(() => { ... }, [ dependencies ]);
```

A function that should be executed AFTER every component evaluation IF the specified dependencies changed

**Your side effect code goes into this function.**

Dependencies of this effect – the function only runs if the dependencies changed

**Specify your dependencies of your function here**

## Introducing useReducer() for State Management

Sometimes, you have **more complex state** – for example if it got **multiple states, multiple ways of changing** it or **dependencies** to other states



useState() then often **becomes hard or error-prone to use** – it's easy to write bad, inefficient or buggy code in such scenarios



useReducer() can be used as a **replacement** for useState() if you need **“more powerful state management”**

## Understanding useReducer()

```
const [state, dispatchFn] = useReducer(reducerFn, initialState, initFn);
```

The state snapshot used in the component re-render/ re-evaluation cycle

A function that can be used to dispatch a new action (i.e. trigger an update of the state)

The initial state

A function to set the initial state programmatically

`(prevState, action) => newState`

A function that is **triggered automatically** once an action is **dispatched** (via `dispatchFn()`) – it **receives the latest state snapshot** and **should return the new, updated state**.

## useState() vs useReducer()

Generally, you'll know when you need useReducer() (→ when using useState() becomes cumbersome or you're getting a lot of bugs/ unintended behaviors)

### useState()

The main state management "tool"

Great for independent pieces of state/ data

Great if state updates are easy and limited to a few kinds of updates

### useReducer()

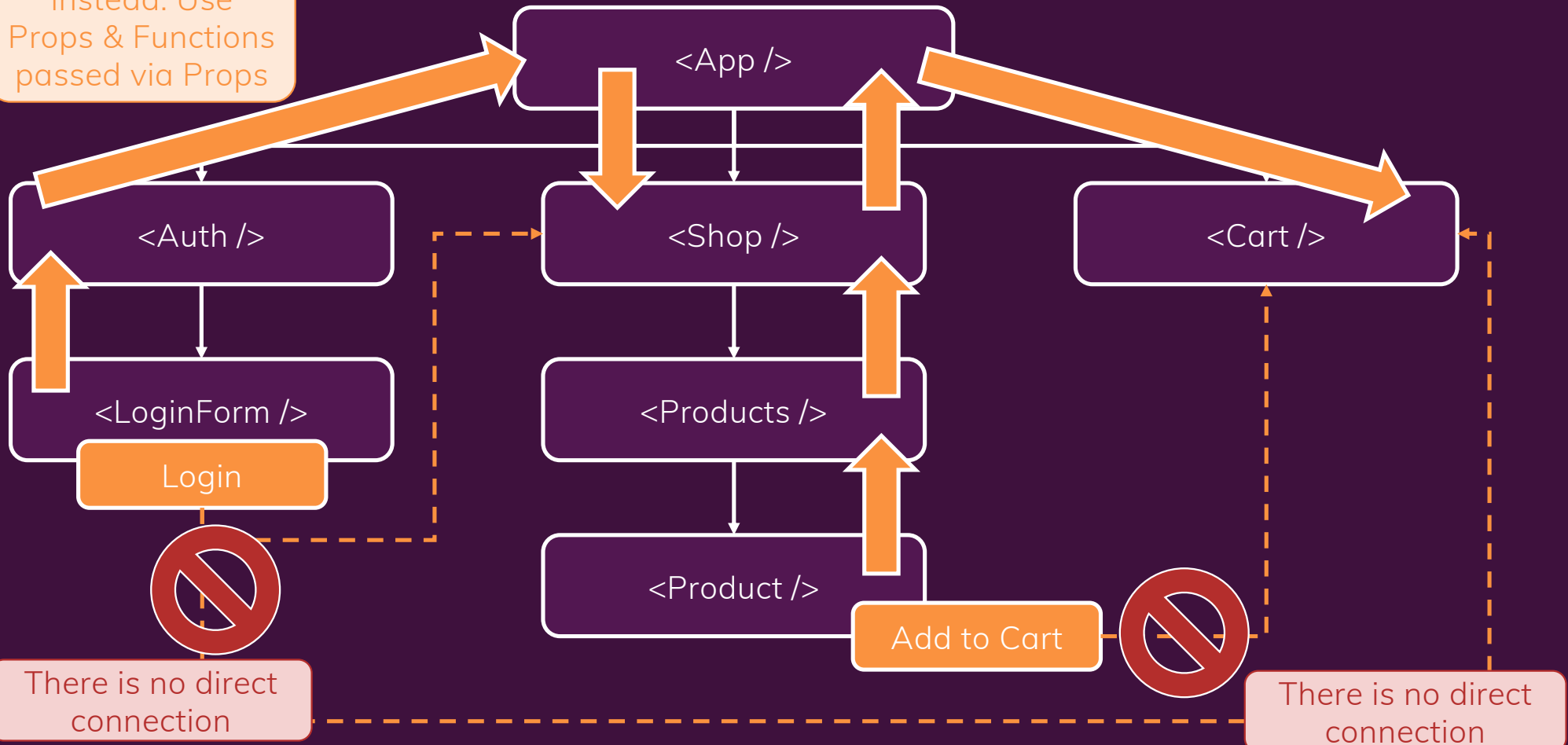
Great if you need "more power"

Should be considered if you have related pieces of state/ data

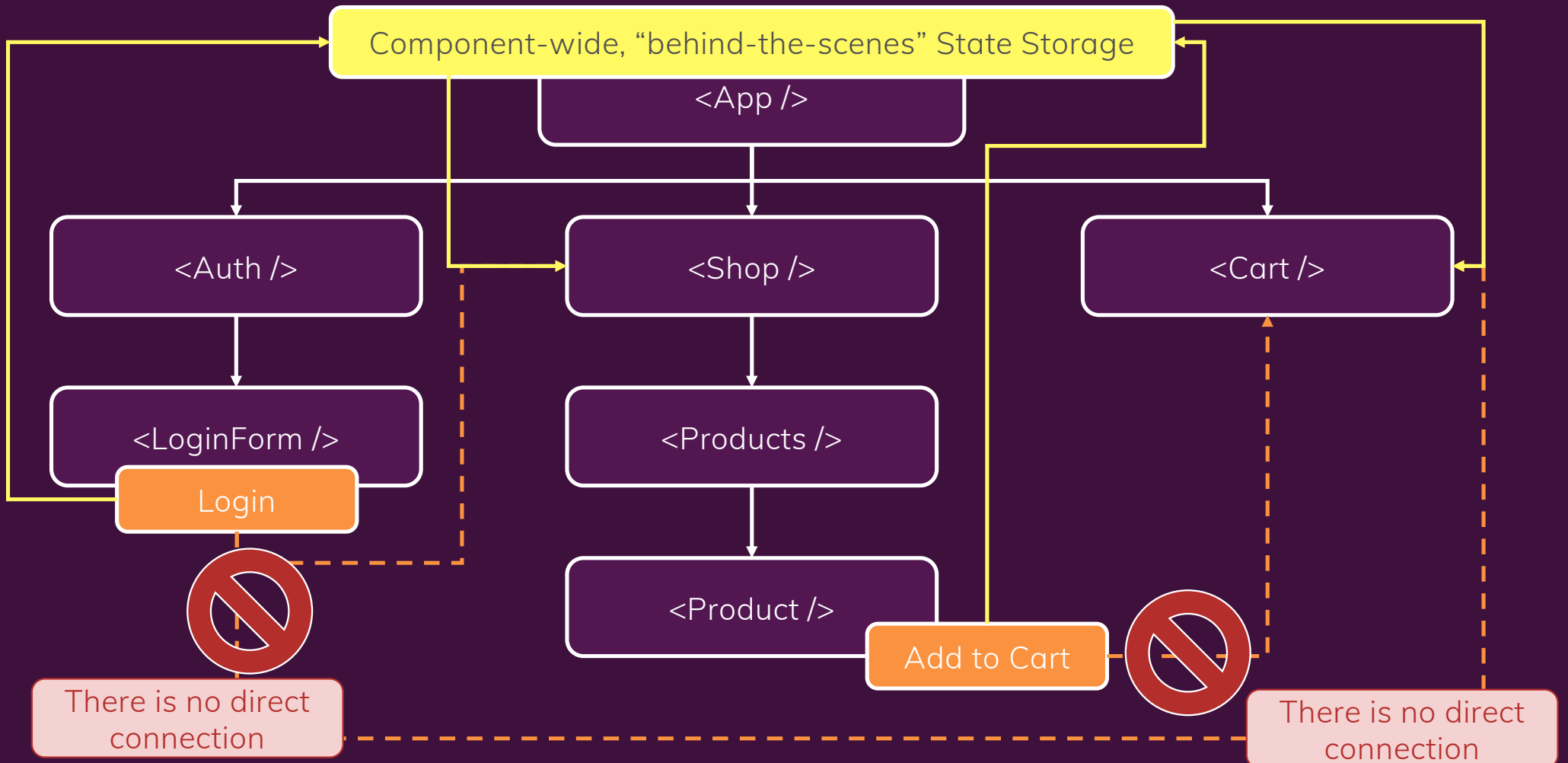
Can be helpful if you have more complex state updates

# Component Trees & Component Dependencies

Instead: Use  
Props & Functions  
passed via Props



## Context to the Rescue!



## Context Limitations

React Context is **NOT optimized** for high frequency changes!



We'll explore a better tool for that, later

React Context also **shouldn't be used to replace ALL** component communications and props



Component should still be configurable via props and short "prop chains" might not need any replacement



# Rules of Hooks

Only call React Hooks in **React Functions**

React  
Component  
Functions

Custom Hooks  
(covered later!)

Only call React Hooks at the **Top Level**

Don't call them  
in nested  
functions

Don't call them  
in any block  
statements

+ extra, unofficial Rule for **useEffect()**: ALWAYS add everything you refer to inside of `useEffect()` as a dependency!