

So we just created an important piece of state and lifted it up to a parent component that is common to both components that need to use or to update that state. Right. However, this whole idea might still be a bit confusing because in fact, it can seem quite counterintuitive. And so let's now look at another example and some diagrams to really understand how lifting up state works and why it's so important. And as an example, let's use the checkout part of the Udemy interface that we have seen in a previous lecture. And let's say that we started by building this promotions component where the user can input coupon codes that will then be added to a list of applied coupons. So that sounds like we need a piece of state called coupons right here, right. So that coupon state is now local to the promotions component along with a set coupons function coming from use state. Now next, we set out to build the total component, but here we quickly realize that the total component also needs access to the coupon state. Otherwise, without knowing which coupons have been applied, how would the total component know what discounts to apply and what price to display? And so here we encounter a problem. How do we give the total component access to the coupon state? Because in React we have one way data flow. So data can only flow down from parents to children, but not sideways to sibling components. Therefore, we cannot simply pass the coupons data as props to the total component. That's just not possible. And so we need a way of sharing state with other components that are further up or sideways in the component tree. But luckily for us, we already did exactly that in the last lecture with the item state by lifting it up. And so we already know that lifting up state is the technique that will solve this problem. But what does that mean and how exactly does it work? Well, lifting state up simply means to place some state in a component that is a parent of both components that need the piece of state in question. So in this example, we would remove the coupon state from promotions and place it in the checkout component. And just like this, we have lifted state up to the closest common parent of both total and promotions. And now giving both these components access to the state is as easy as passing it down using props. And that's it. So by lifting State up, we have just successfully shared one piece of state with multiple components in different positions in the component tree, which is something that we need to do all the time in React apps. And so it's really important that you get used to this pattern and remember that we need this pattern in the first place as a direct consequence of react's one way data flow. But anyway, all this now seems to be working just fine at this point. But now what happens when we want to add a new coupon to the coupon state? Or in other words, what happens when the user inputs a new coupon and clicks on the apply button? Well, we want to update the coupon state, right. But how do we do that now? Because after lifting the state up, it now lives in the parent component. So not in the promotions component anymore. Promotions only receive this data via props. But as you know, we cannot mutate props. So that's one of the hard rules of react. So what we're asking here is if we have one way data flow, so if data can only flow from parents to children, then how can the child component promotions update the state that lives in the parent component checkout? Well, actually, the solution is quite simple. All we have to do is to also pass the set coupons function down as a prop to the components who need to update the state. And so now that we have these set coupons function in promotions, once a new coupon is added, we can simply use set coupons to update the state that lives in the parent component. And this is actually exactly what we also did in the previous lecture with the difference that we didn't directly pass set items, but a function that uses set items to update the items, which is essentially the same thing. But anyway, we can call this technique of passing down a setter function child to parent communication or also inverse data flow inverse, because usually data only flows down. But here we basically have a trick that allows us to basically have the data flowing up as well. Now, of course, this is not truly flowing up, but this workaround of passing down the setter function and use it to update the parent state is pretty close to actually having data flowing up the tree. Now, when I first learned about this technique years ago, it actually took me quite some time to wrap my head around this whole idea because this can actually be quite confusing. And so that's why I created an extra lecture with all these diagrams for you, hoping that you will have an easier time understanding child to parent communication. But that's enough talk. Let's now go back to our code and use this in practice a few more times.