







In the lecture about the renderer phase, we left out one critical piece, which was the Diffing algorithm. That is part of the reconciliation process. So we mentioned thing back then, but we didn't really go into how Diffing works. And since that's really important, let's do that now. So this thing is, first of all, based on two fundamental assumptions. The first one is that two elements of different types will produce different trees. The second assumption is that elements with a stable key. So a key that is consistent over time will stay the same across renders. Now, these assumptions may seem pretty obvious, especially the first assumption, but they allow the algorithm to be orders of magnitude faster. Going, for example, from a billion operations per thousand processed elements to just a thousand operations. Now remember that Diffing is comparing elements step by step between two renders based on their position in the tree. And there are basically two different situations that need to be considered. First, having two different elements at the same position in the tree between two renders and second, having the same element at the same position in the tree. So those are the only two situations that matter. And so let's now start with the first situation. So let's say that at some point an application is rerendered. And in the diffing process we find that an element has changed in a certain position of the tree. Now here we're actually not looking at a tree, but at the JS code, which leads to that tree because I find that it's a bit easier to understand this way. But anyway, in the case of a Dom element changing like this, changing simply means that the type of the element has changed. Like in this example from a div to a header. So in a situation like this, React will assume that the element itself plus all its children are no longer valid. Therefore all these elements will actually be destroyed and removed from the Dom and that also includes their state, which is really important to remember. So as we see in this example, both the div element and the search bar component will be removed from the Dom and will then be rebuilt as a header with a brand new search bar component instance as a child. So if the child elements stayed the same across renders, the tree will actually get rebuilt. But it gets rebuilt with brand new elements. And so if there were any components with state, that state will not be recovered. So this effectively resets state and this has huge implications for the way that react applications work in practice. And that's why we will see some examples of this behavior in the next lecture. Now. Everything we just learned works the exact same way for React elements. So basically for component instances, as we can see in this second example. So here the search bar component changed to a profile menu component and therefore the search bar is again completely destroyed, including its state and removed from the Dom. Okay. So this is the first situation. The second situation is when between two renders, we have the exact same element at the same position in the tree. And this one is actually way more straightforward. So if after a render an element at a certain position in the tree is the same as before, like in these examples right here, the element will simply be kept in the Dom and that includes all child elements and more importantly, the component's state. Now, this may sound pretty obvious, but it actually has some important implications in practice. So again, the same element at the same position in the tree stays the same and preserves state. And it works like this for Dom elements and for React elements as well. Now, looking at these examples, we actually see that something has changed. However, it was not the element type that has changed, but simply the class name attribute in the div and the weight prop in the search bar component. And so what React is going to do is to simply mutate the Dom element attributes and in the case of react elements, it will pass in the new props, but that's it. So React tries to be as efficient as possible. And so the Dom elements themselves will stay the same. They are not removed from the Dom and even more importantly, the state will not be destroyed. Now, sometimes we actually don't want this standard behavior, but instead to create a brand new component instance with new state. And so that's where the key prop comes into play as we will learn about after seeing these rules that we just learned about in action.