















Welcome back to probably the most complicated and most confusing lecture of the entire course. But I don't say this to scare you because this might also be the most interesting and fascinating lecture of the course. Now, if you don't immediately understand % of what I'm going to teach you here, that's absolutely fine. So don't stress about it at all. Again, you can use React without knowing any of this stuff, but if you have a curious mind and feel the need to understand how react does what it does, then this video is for you. Now, before we even start, let's first go back to where we first learned about the mechanics of state. Remember this diagram? Well, back then I told you that we can conceptually imagine this as a new view being rendered on the screen so into the Dom. However, now we know that this was technically not true because rendering is not about the screen or the Dom or the View. It's just about calling component functions. I also told you that whenever there is a rerender, react discards the old component view and replaces it with a brand new one. However, that's also technically not true. So the Dom will actually not be updated for the entire component instance. So if those things are not true, then let's now learn what happens instead and how rendering actually works. So in the previous lecture we talked about how renders are triggered. In this lecture, we're going to learn all about how renders are actually performed in the render phase. So at the beginning of the render phase, React will go through the entire component tree, take all the component instances that triggered a rerender and actually render them, which simply means to call the corresponding component functions that we have written in our code. This will create updated react elements which all together make up the so-called virtual dom. And this is a term that you might have heard before. And so let's dig a little bit deeper now into what the virtual Dom actually is. So on the initial render, react will take the entire component tree and transform it into one big react element, which will basically be a react element tree like this one. And this is what we call the virtual Dom. So the virtual dom is just a tree of all react elements created from all instances in the component tree. And it's relatively cheap and fast to create a tree like this, even if we need many iterations of it. Because in the end it's just a JavaScript object. Now, Virtual Dom is probably the most hyped and most used term when people describe what React is and how it works. But if we think about it, if the virtual Dom is just this simple object, it's actually not such a big deal. Right? And that's why the React team has really downplayed the meaning of this name. And the official documentation actually no longer mentioned the term virtual dom anywhere. But I'm still using this term here because everyone still uses it and also because it just sounds a bit nicer than React Elementary. Also, some people confuse the term with the shadow dom, even though it has nothing to do with the virtual dom in React. So the shadow dom is actually just a browser technology that is used in stuff like web components. But anyway, let's now suppose that there is going to be a state update in component D, which will of course trigger a rerender. That means that react will call the function of component D again and place the new react element in a new react element tree. So in a new virtual dom. But now comes the very important part, which is this. Whenever React renders a component, that render will cause all of its child components to be rendered as well. And that happens no matter if the props that we pass down have changed or not. So again, if the updated components returns one or more other components, those nested components will be re rendered as well all the way down the component tree. This means that if you update the highest component in a component tree in this example component A, then the entire application will actually be re rendered. Now, this may sound crazy, but React uses this strategy because it doesn't know beforehand whether an update in a component will affect the child components or not. And so by default, React prefers to play it safe and just render everything. Also, keep in mind once again that this does not mean that the entire Dom is updated. It's just a virtual dom that will be recreated, which is really not a big problem in small or medium sized applications. Okay, So with this, we now know what this new virtual dom here means. And so let's keep moving forward. So what happens next is that this new virtual Dom that was created after this date update will get reconciled with the current so-called fiber tree as it exists before the state update. Now this reconciliation is done in react's reconciler, which is called fiber, and that's why we have a fiber tree. Then the results of this reconciliation process is going to be an updated fiber tree. So a tree that will eventually be used to write to the Dom. So this is a high level overview of the inputs and the outputs of reconciliation. But of course now we need to understand what reconciliation is and how it works. So you might be wondering why do we even need stuff like the virtual dom? A reconciler and those fiber trees. Why not simply update the entire dom whenever state changes somewhere in the app? Well, the answer is simple. Remember how we said that creating the virtual Dom? So the react element tree for the entire app is cheap and fast because it's just a JavaScript object. Well, writing to the Dom is not cheap and fast. It would be extremely inefficient and wasteful to always write the entire virtual dom to the actual dom each time that a render was triggered. Also, usually when the state changes somewhere in the app, only a small portion of the Dom needs to be updated and the rest of the Dom that is already present can be reused. Now, of course, on the initial render, there is no way around creating the entire Dom from scratch, but from there on, doing so doesn't make sense anymore. So imagine that you have a complex app like Udemy.com, and when you click on some button there, then show Modal is set to true, which in turn will then trigger a modal to be shown. So in this situation only the Dom elements for that modal need to be inserted into the Dom and the rest of the dom should just stay the same. And so that's what React tries to do. Whenever a render is triggered, React will try to be as efficient as possible by reusing as much of the existing Dom tree as possible. But that leads us to the next question. So how does React actually do that? How does it know what changed from one render to the next one? Well, that's where a process called reconciliation comes into play. So reconciliation is basically deciding exactly which Dom elements need to be inserted, deleted or updated in order to reflect the latest state changes. So the result of the reconciliation process is going to be a list of Dom operations that are necessary to update the current Dom with the new state. Now reconciliation is processed by a reconciler, and we can say that the reconciler really is the engine of react. It's like the heart of react. So it's this reconciler that allows us to never touch the Dom directly and instead simply tell react what the next snapshot of the UI should look like based on state. And as I mentioned before, the current reconciler in React is called fiber. And this is how it works. So during the initial render of the application, fiber takes the entire React element tree. So the virtual Dom and based on it builds yet another tree, which is the fiber tree. The fiber tree is a special internal tree where for each component instance and Dom element in the app there is one so-called fiber. Now, what's special about this tree is that unlike react elements in the virtual Dom, fibers are not recreated on every render. So the fiber tree is never destroyed. Instead, it's a mutable data structure. And once it has been created during the initial render, it simply mutated over and over again in future reconciliation steps. And this makes fibers the perfect place for keeping track of things like the current component state props, side effects, list of used hooks and more. So the actual state and props of any component instance that we see on the screen are internally stored inside the corresponding fiber in the fiber tree. Now, each fiber also contains a queue of work to do, like updating state updating refs, running registered side effects, performing Dom updates and so on. This is why a fiber is also defined as a unit of work. Now if we take a quick look at the fiber tree, we will see that the fibers are arranged in a different way than the elements in the React element tree. So instead of a normal parent child relationship, each first child has a link to its parent and all the other children then have a link to their previous sibling. And this kind of structure is called a linked list, and it makes it easier for react to process the work that is associated with each fiber. We also see that both trees include not only react elements or components, but also regular Dom elements such as the H and button elements in this example. So both trees really are a complete representation of the entire Dom structure, not just of React components. Now going back to the idea that fibers are units of work. One extremely important characteristic of the fiber reconciler is that work can be performed asynchronously. This means that the rendering process, which is what the reconciler does, can be split into chunks. Some tasks can be prioritized over others and work can be paused, reused or thrown away if not valid anymore. Just keep in mind that all this happens automatically behind the scenes. It's completely invisible to US developers. There are, however, also some practical uses of this asynchronous rendering because it enables modern so-called concurrent features like suspense or transitions starting in React . It also allows the rendering process to pause and resume later so that it won't block the browser's JavaScript engine with too long renders, which can be problematic for performance in large applications. And again, this is only possible because the render phase does not produce any visible output to the Dom yet. Okay. So at this point we know what the fiber reconciler is and how the fiber tree works, but now it's time to talk about what fiber actually does, which is the reconciliation process. And the best way to explain how reconciliation works is by using a practical example. So let's take the virtual Dom and the corresponding fiber tree from the last slide which correspond to this piece of code right here. So in the app component, there is a piece of state called Show Modal, which is currently set to true. And you can pause the video here to analyze it, but it's not really necessary. So let's say now that the state is updated to false. This will then trigger a rerender which will create a new virtual dom. And in this tree, the modal and all its children are actually gone because they are no longer displayed when show modal is not true. Also, all remaining react elements are yellow, meaning that all of them were re rendered. And do you remember why that is? That's right. It's because all children of a re rendered element are re rendered as well as we just learned a few minutes ago. But anyway, this new virtual dom now needs to be reconciled with the current fiber tree, which will then result in this updated tree, which internally is called the Work in Progress Tree. So whenever reconciliation needs to happen, fiber walks through the entire tree step by step and analyzes exactly what needs to change between the current fiber tree and the updated fiber tree based on the new virtual Dom. And this process of comparing elements step by step based on their position in the tree is called Diffing and we will explore exactly how Diffing works a bit later in this section because that's actually pretty important in practice. But anyway, let's quickly analyze or updated fiber tree where a marked new work that is related to Dom mutations. So first the BTN component has some new text, and so the work that will need to be done in this fiber is a Dom update. So in this case, swapping text from height to rate. Then we have the modal overlay, H and button. So these were in the current fiber tree but are no longer in the virtual dom and therefore they are marked as Dom deletions. Finally, we have the interesting case of the video component. So this component was re rendered because it's a child of the app component, but it actually didn't change. And so as a result of reconciliation, the Dom will not be updated in this case. Now, once this process is over, all these mutations will be placed into a list called the list of effects, which will be used in the next phase. So in the commit phase to actually mutate the Dom. Now, what I showed you here was actually still a bit oversimplified. If you can believe that. But I think that this is more than enough for you to understand how this process works. Okay. So that was quite a deep dive. But now we're back here in the high level overview of the render phase. So we learned that the results of the reconciliation process is a second updated fiber tree, plus basically a list of updates that need to be performed in the next phase. So React still hasn't written anything to the Dom yet, but it has figured out this so called list of effects. So this is the final result of the render phase as it includes the Dom operations that will finally be made in the commit phase, which is the topic of our next video. So if you don't want to break the flow, then let's move on there right now.