





Welcome back to part three of how rendering works behind the scenes. So we just finished learning about the render phase, which resulted in a list of Dom updates, and this list will now get used in the commit phase. Now, technically speaking, the current work in progress fiber tree also goes into this commit phase. But let's keep it simple here. So these are more conceptual diagrams so that we can understand what is happening. Not a % accurate description of the algorithms inside react. All right. But anyway, as you know by now, the commit phase is where React finally writes to the Dom. So it inserts, deletes and updates Dom elements. You will sometimes also read that React flushes updates to the Dom in this phase. So basically React goes through the effects list that was created during rendering and applies them one by one to the actual Dom elements that were in the already existing dom tree. Now writing to the Dom happens all in one go. So we say that the commit phase is synchronous, unlike the rendering phase, which can be paused. So committing cannot be interrupted. And this is necessary so that the Dom never shows partial results, which ensures that the UI always stays consistent. In fact, that's the whole point of dividing the entire process into the render phase and the commit phase. In the first place, it's so that rendering can be paused, resumed and discarded, and the result of all that rendering can then be flushed to the Dom in one go. Then once the commit phase is completed, the work in progress fiber tree becomes the current tree for the next render cycle. That's because, remember, fiber trees are never discarded and never recreated from scratch. Instead, they are reused in order to save precious rendering time. And with that, we close up the commit phase. The browser will then notice that the Dom has been changed and as a result it will repaint the screen whenever it has some idle time. So this is where these Dom updates are finally made visible to the user in the form of an updated user interface. Now, I'm not going to go into how this phase works because this is really more about how browsers work internally and not react. However, there is still one more thing that we need to talk about. So the browser paint phase that we just mentioned is of course performed by whatever browser the user is using, and the render phase is obviously performed by the React library. But what about the commit phase? We would think that it's also done by react, right? But actually that's not true. It's actually a separate library that writes to the Dom and it's called React Dom. So not very creative, but that's just what it's called. So in fact, React itself does never touch the Dom and it actually has no idea where the result of the render phase will actually be committed and painted. So react only does the render phase but not the commit phase. And the reason for that is that React itself was designed to be used independently from the platform where elements will actually be shown and therefore react can be used with many different so-called hosts. Now, up until this point, we have only ever thought of react in conjunction with the Dom because we usually use it to build web application. And in % of the cases, that's actually what we do with React. But the truth is that React is used with other hosts as well. For example, we can actually build native mobile applications for iOS and Android using React Native, or we can build videos with React using a package called Remotion, and we can even create all kinds of documents like word or PDF documents, Figma designs, and many more using different so-called renderers. Now, if we think about this renderer is actually a pretty terrible name because according to React's own terminology, renderers do not render, but they commit the results of the render phase. But they think that this renderer name comes from a time before react, divided the render and the commit phase into two separate phases. And so they chose this term of renderer because it fits with the common sense definition of rendering. But anyway, in all these situations, the results of the render phase is not really a list of Dom updates, but a list of updates of whatever elements are used in the host that's being used. So the term virtual Dom then also doesn't really make much sense when we look at it from this angle, which is just one more reason why the React team prefers the more accurate name of React Element Tree. Now, all these details are, of course, not really that important. What I want you to retain from this slide is that the React library is not the one responsible for writing to the Dom because the Dom is just one of many hosts to which React apps can be committed. So to which they can be output basically. And for each of these hosts, we have a different package that we can use. And that's why in our Index.js file we always import both react and react. Dom Right. And so now you know the exact reason why we have to do that. All right. So after looking at all these phases in so much detail, let's do a quick recap here and summarize everything that we have learned. So the whole process of rendering and displaying a React application on the screen starts with a trigger, which can either be the initial render of the app or a state update in one of the component instances. This then triggers the render phase, which does not produce any visual output. So this phase starts by rendering all component instances that need a rerender, and rendering in react simply means to call the components functions. This will create one or more updated react elements which will be placed in a new virtual Dom, which is actually simply a tree of react elements. Now, what's really important to remember about this process is that rendering a component will cause all of its child components to be rendered as well, no matter if props changed or not. This is because React doesn't know whether children have been affected by the parent rerendering or not. Now, next up, this new virtual Dom needs to be reconciled with the current fiber tree. So with the representation of the element tree before the state update, this is necessary because it would be slow and inefficient to destroy and rebuild the entire Dom tree each time that something on the screen must be updated. Instead, reconciliation tries to reuse as much of the Dom as possible by finding the smallest number of Dom updates that reflect the latest state update on the screen. Now this reconciliation process is done using a reconciler called fiber, which works with a mutable data structure called the fiber tree. And in this tree, for each react element and Dom element, there is a fiber. And this fiber holds the actual component state props and a queue of work. After reconciliation. This queue of work will contain the Dom updates that are needed for that element. Now the computation of these updates is performed by a diffing algorithm, which step by step compares the elements in the new virtual Dom with the elements in the current fiber tree. So to see what has changed. So the final result of the render phase. So basically of this reconciliation and diffing process is a second updated fiber tree as well as a list of all necessary Dom updates. Now it's important to note that the render phase is asynchronous, so fiber can prioritize and split work into chunks and pause and resume some work later. And this is necessary for concurrent features and also to prevent the JavaScript engine to be blocked by complex render processes. But anyway, the output of the render phase. So the list of Dom updates will finally actually be written to the Dom in the commit phase. So in this phase, a so-called renderer like React Dom will insert, delete and update dom elements so that we end up with an updated Dom that reflects the new state of the application. And unlike the render phase, the commit phase is actually synchronous. So all the Dom updates are performed in one go in order to ensure a consistent over time. Now, finally, once the browser realizes that the Dom has been updated, it starts a new browser paint. In order to visually update the user interface on the screen. Okay. And there you have it. This is how, in a nutshell, we go from updated react elements all the way to an updated Dom and user interface on the screen. So it sure was a really long process, but I hope that you have learned a lot along the way and that it wasn't too overwhelming. And again, keep in mind that you can build React apps without being aware that most of these things even exist. Now, some of these things do have practical implications and also implications for performance, but we will talk about those later. For now, just take a breath as this was probably the hardest part of the entire course. So good job of sticking with it to the end. And let me know what you think of this series of lectures in the Q&amp;A.