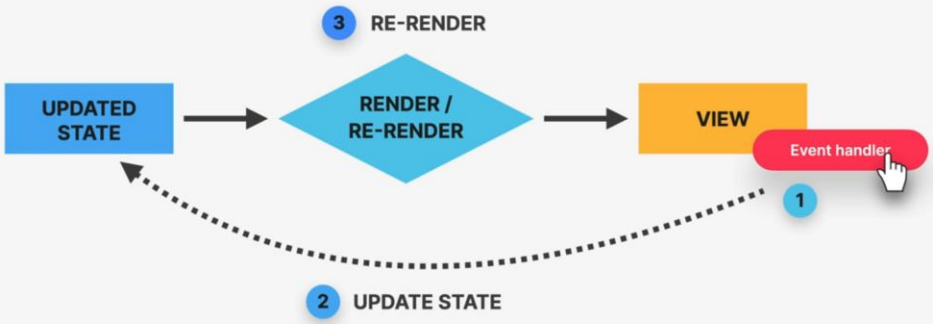
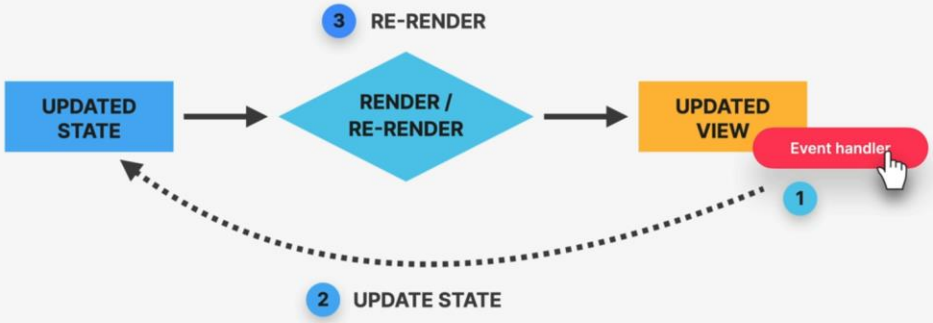
REVIEW: THE MECHANICS OF STATE IN REACT



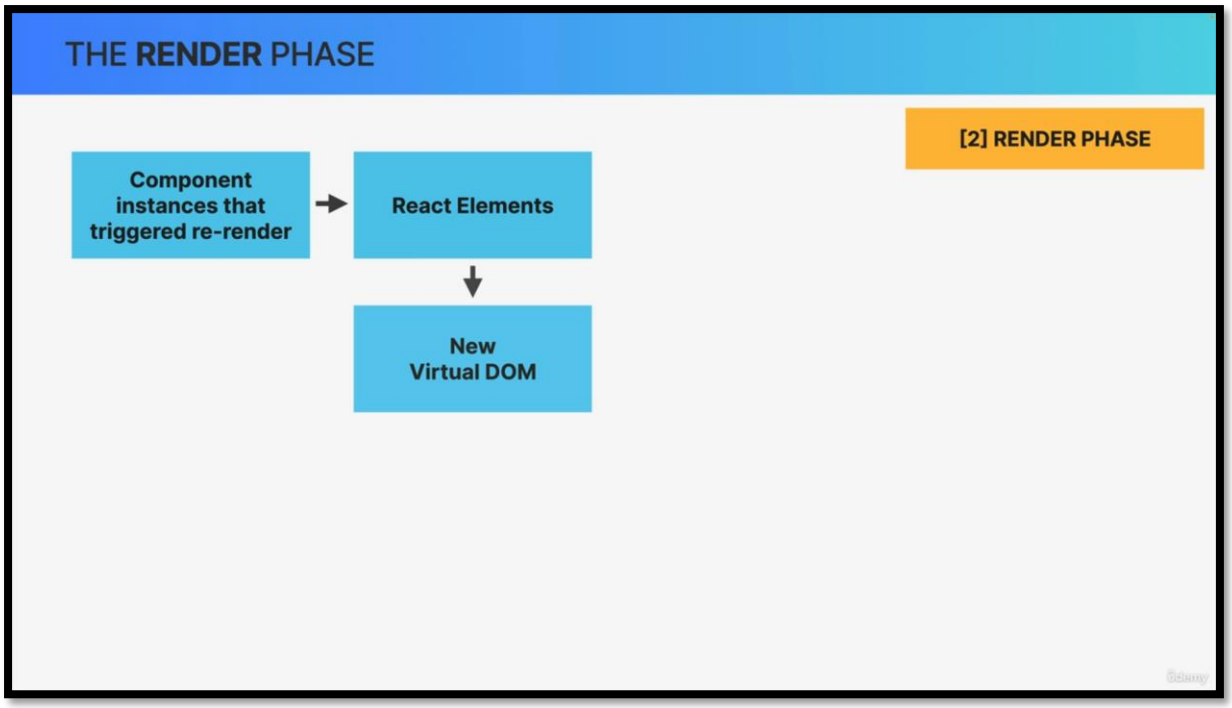
NOT TRUE #1: RENDERING IS UPDATING THE SCREEN / DOM

REVIEW: THE MECHANICS OF STATE IN REACT



NOT TRUE #1: RENDERING IS UPDATING THE SCREEN / DOM

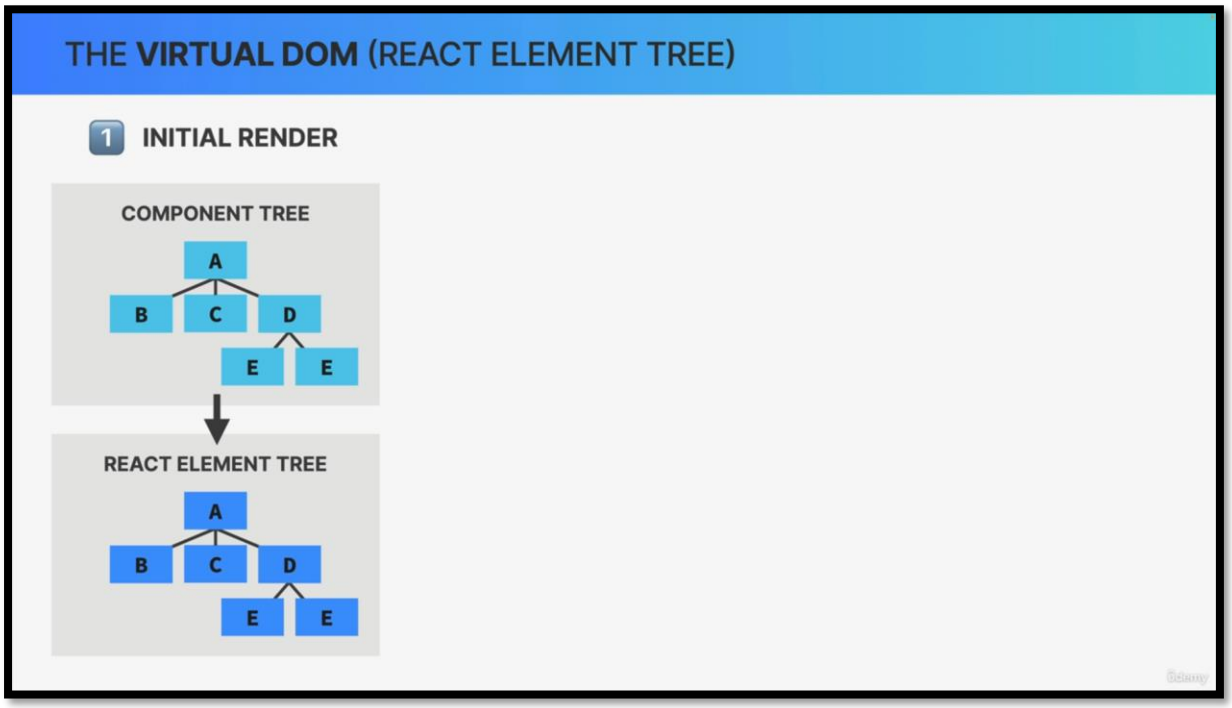
NOT TRUE #2: REACT COMPLETELY DISCARDS OLD VIEW (DOM) ON RE-RENDER



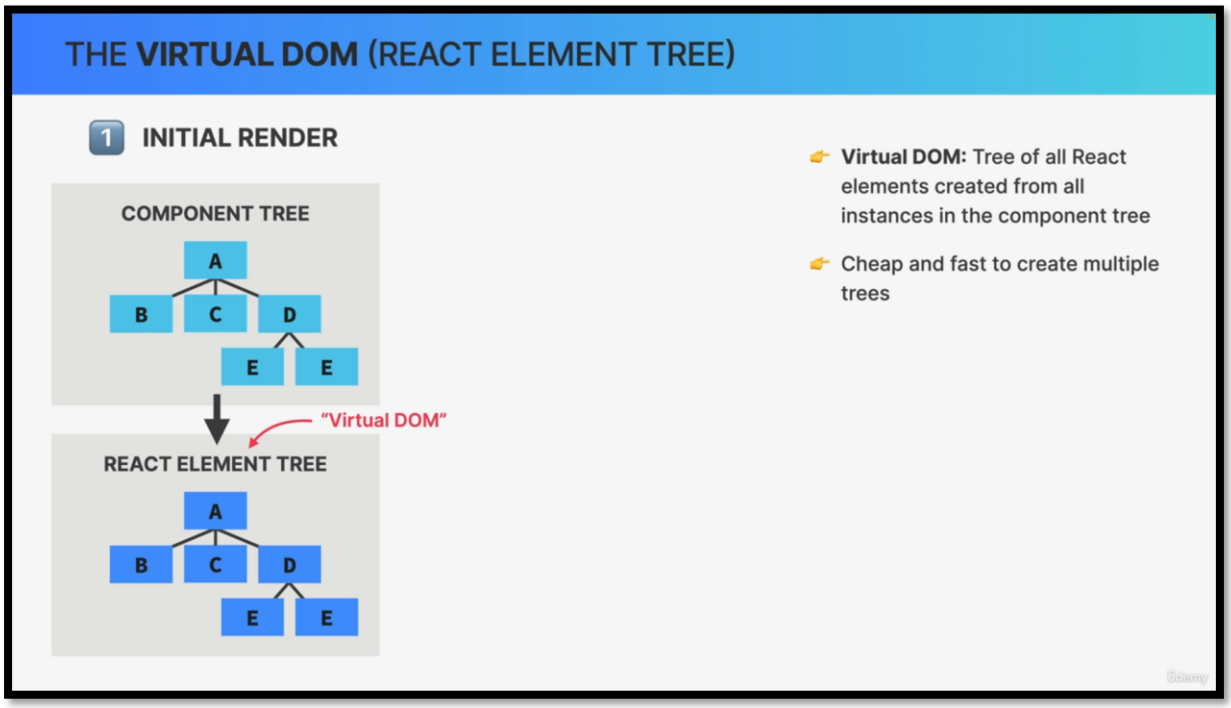
At the beginning of the render phase React will go through the entire component tree, take all the component instances that triggered a re-render and actually render them, which simply means to call the corresponding component functions that we have written in our code.

This will create updated React elements which altogether make up the so-called virtual DOM.

Virtual Dom

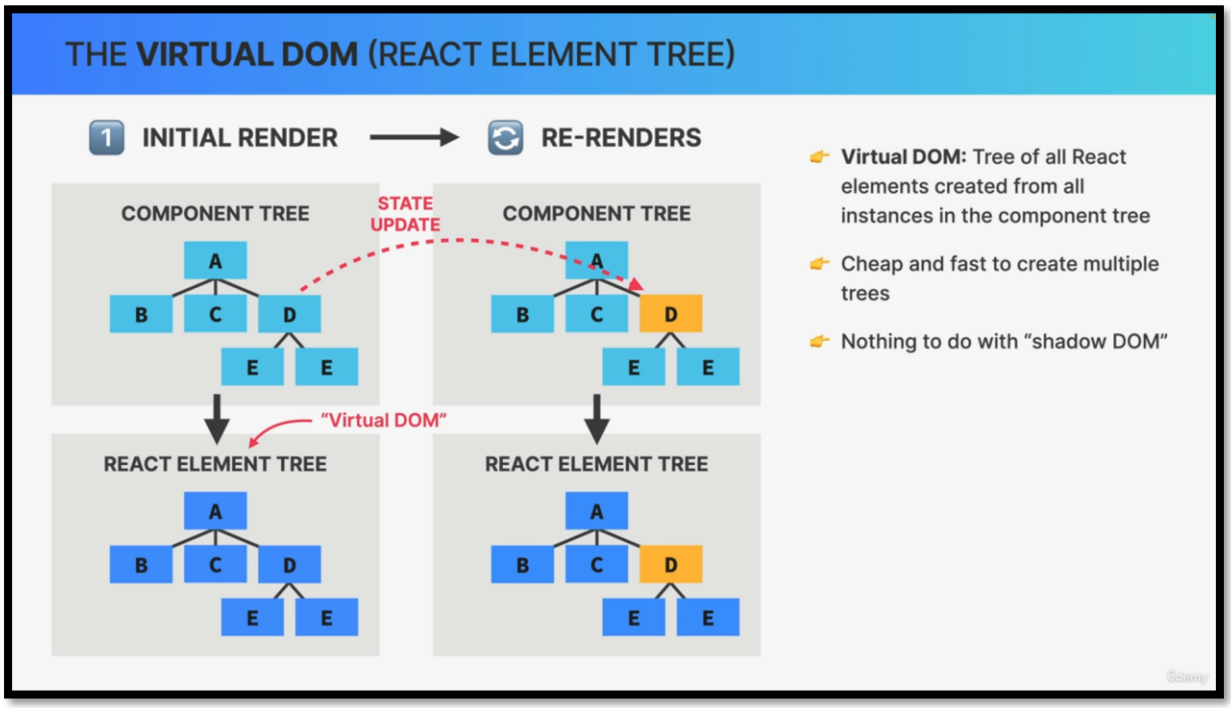


On the initial render, React will take the entire component tree and transform it into one big React element which will basically be a React element tree like this one. This is what we call the virtual DOM.

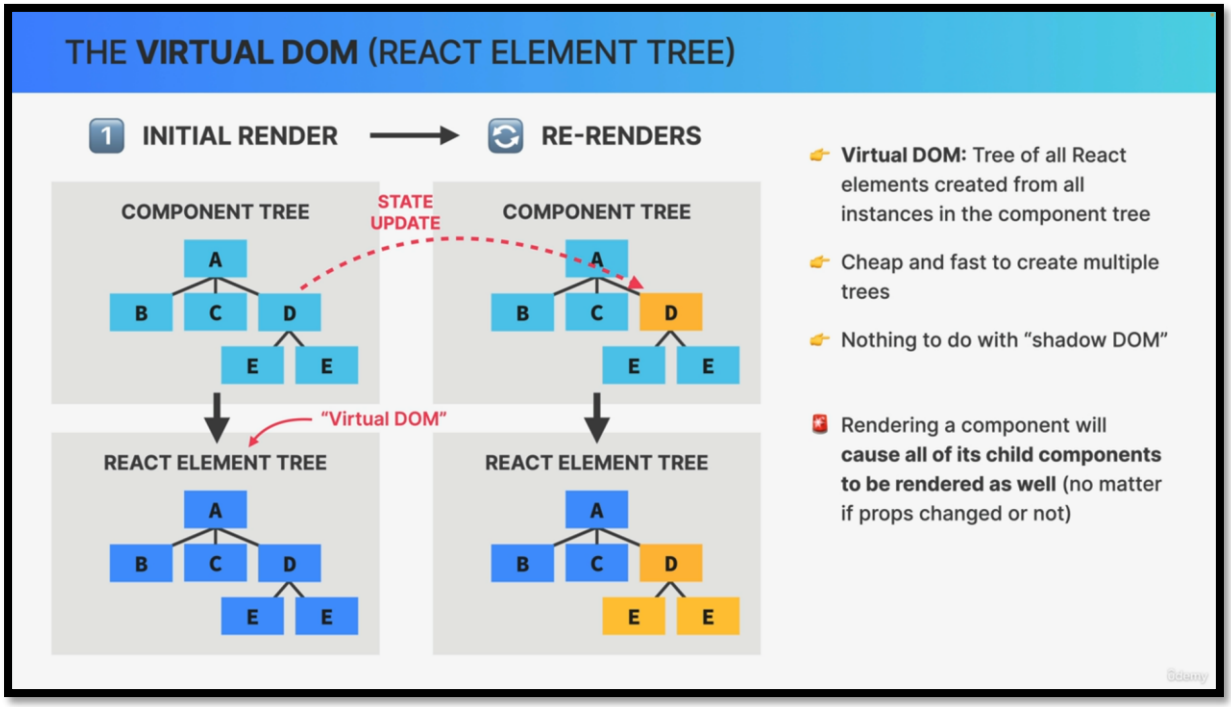


The virtual DOM is just a tree of all React elements created from all instances in the component tree.

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.

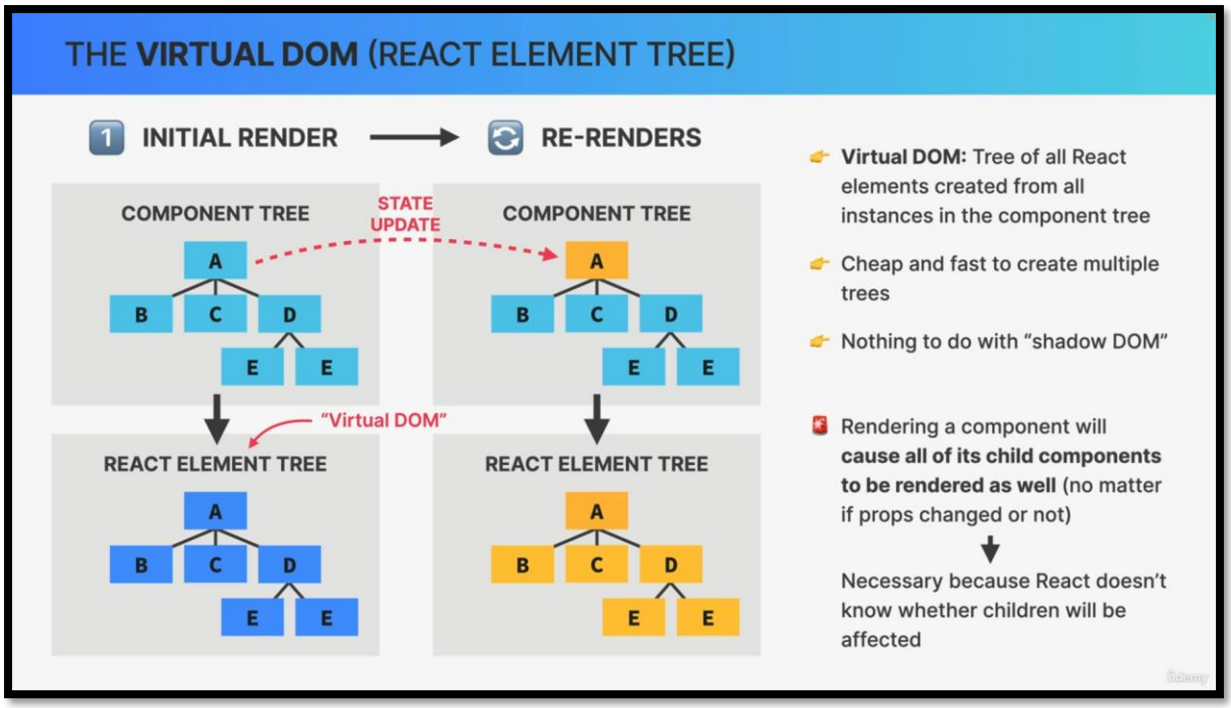


Let's now suppose that there is a state update in component D which will of course trigger a re-render. That means that React will call the function of component D again and place the new React element in a new React element tree or virtual dom. So, in a new virtual DOM.



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 passed down have changed or not.

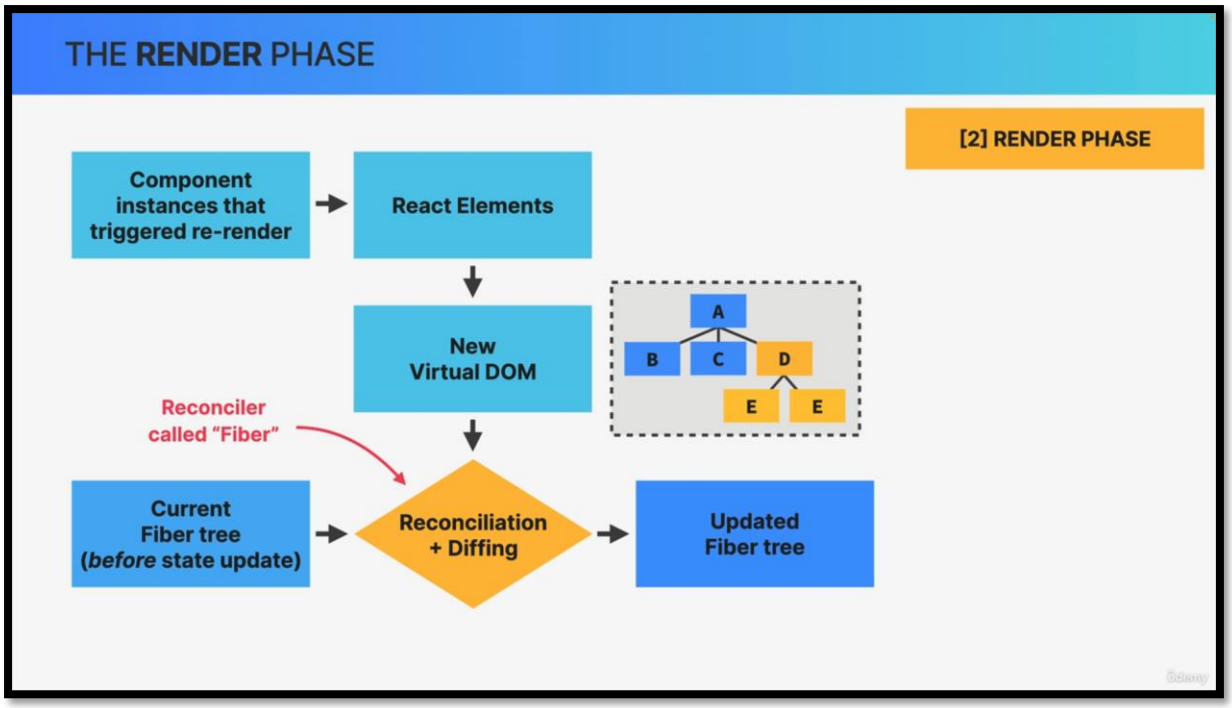
If the updated components return one or more other components, those nested components will be re-rendered as well, all the way down the component tree.



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.

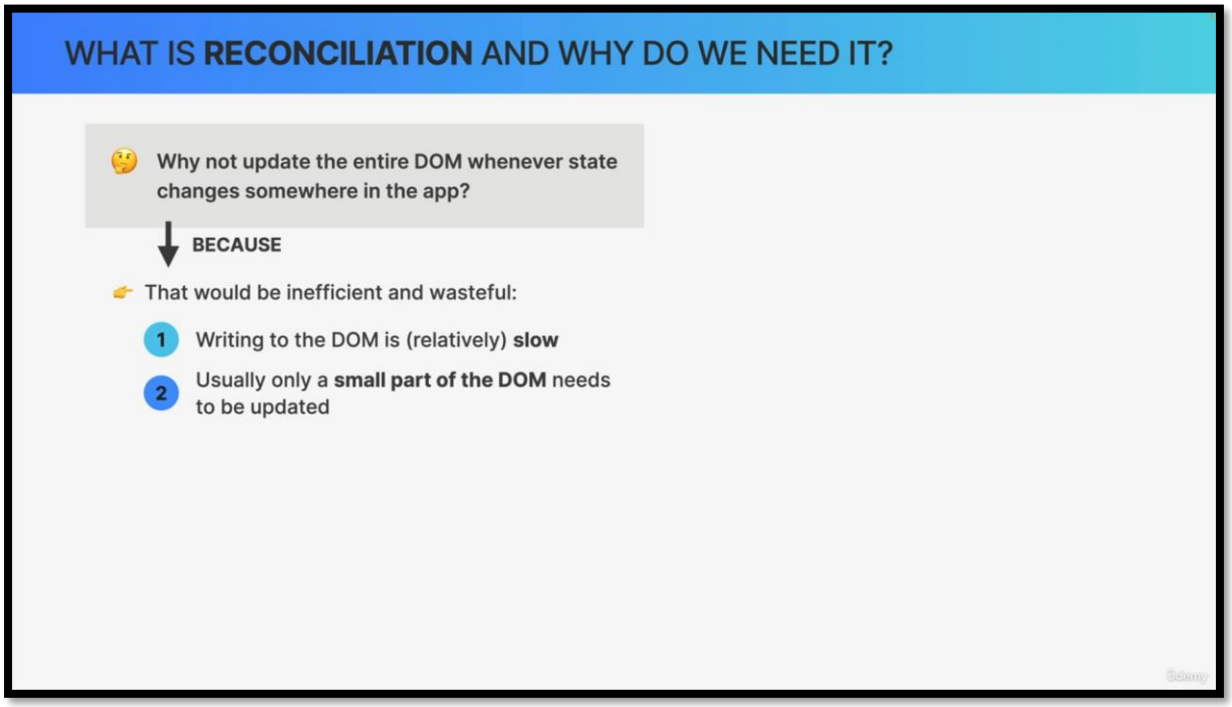
React uses this strategy because it doesn't know beforehand whether an update in a component will affect the child components or not. By default, React prefers to play it safe and just render everything.

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.



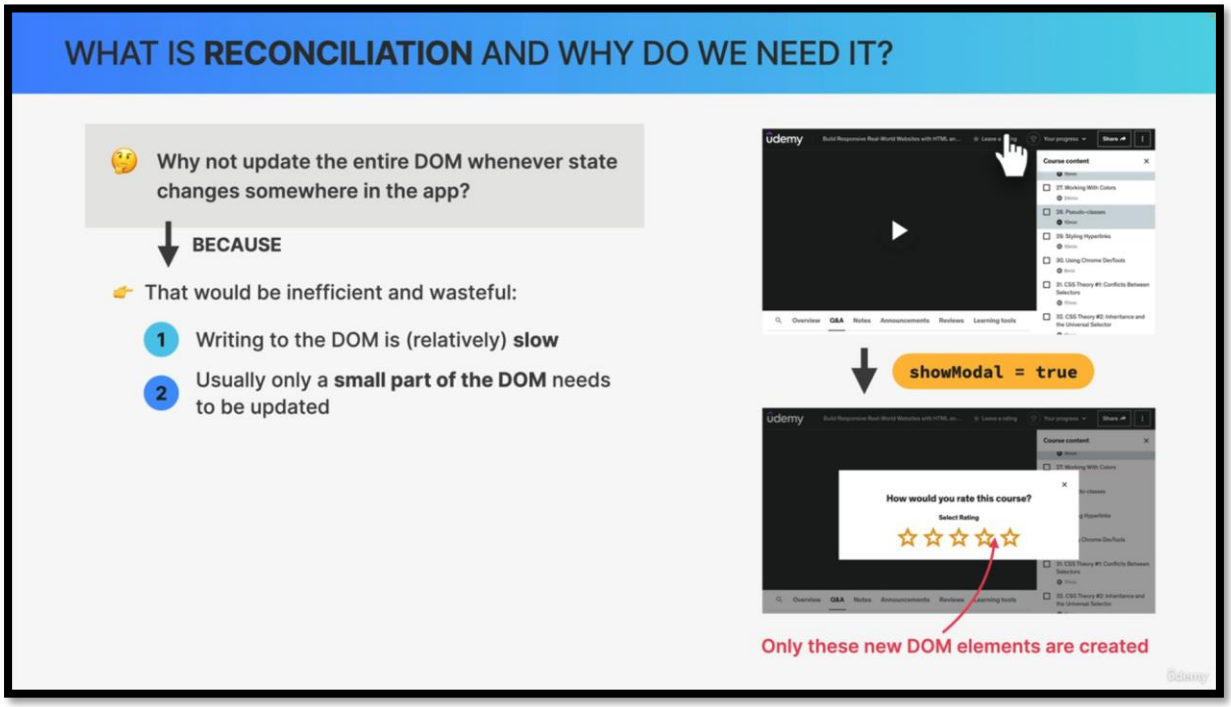
This new virtual DOM that was created after the state update will get reconciled with the current so-called Fiber tree as it exists before the state update.

Now this reconciliation is done in React's reconciler which is called Fiber. The results of this reconciliation process is gonna be an updated Fiber tree, so, a tree that will eventually be used to write to the DOM.



1. Creating the virtual DOM or the React element tree for the entire app is cheap and fast because it's just a JavaScript object. 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.

2. 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. 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.



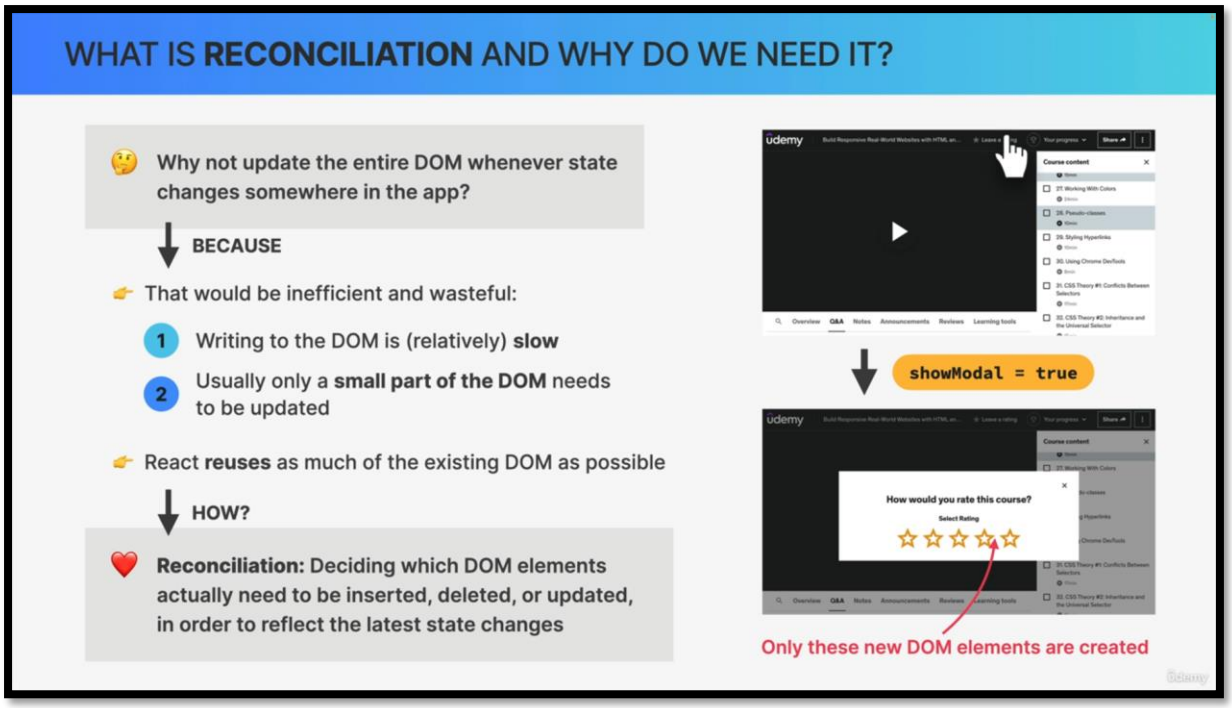
So, imagine that you have a complex app like udemy.com and when you click on some button there then showModal is set to true, which in turn will then trigger a modal to be shown.

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. 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.

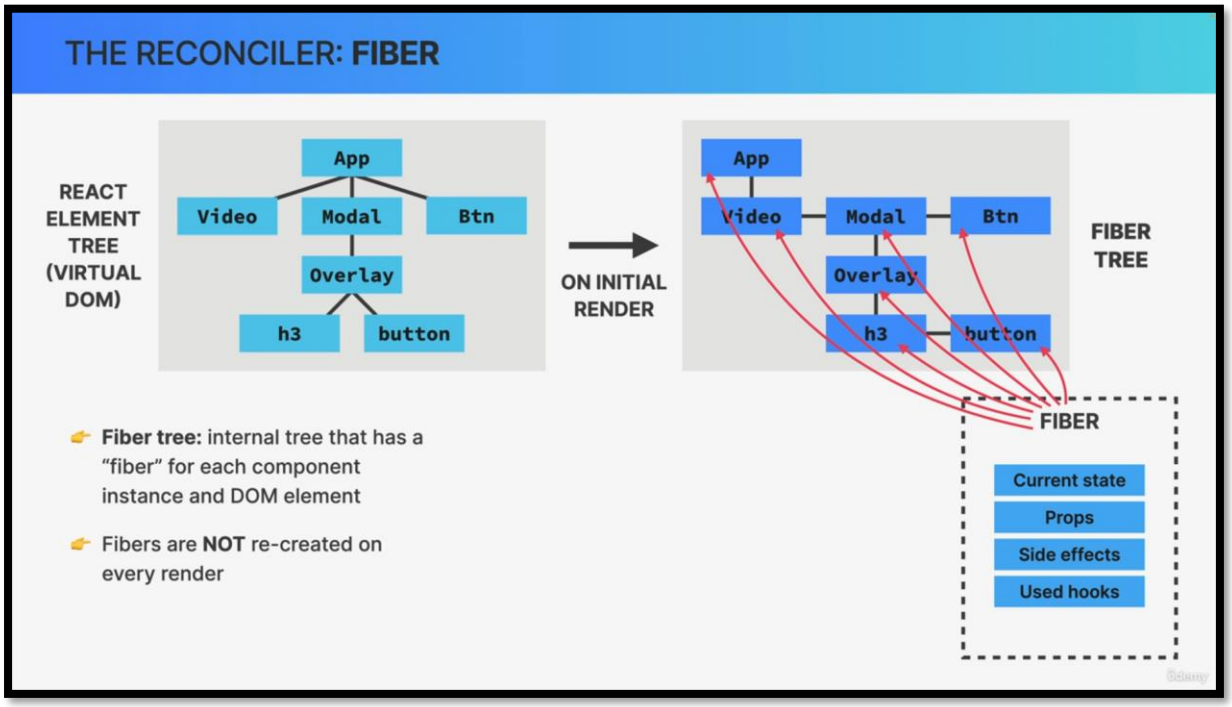
That leads us to the next question. So how does React actually do that? How does it know what changed from one render to the next one?

That's where a process called reconciliation comes into play.



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 gonna be a list of DOM operations that are necessary to update the current DOM with a new state.

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.



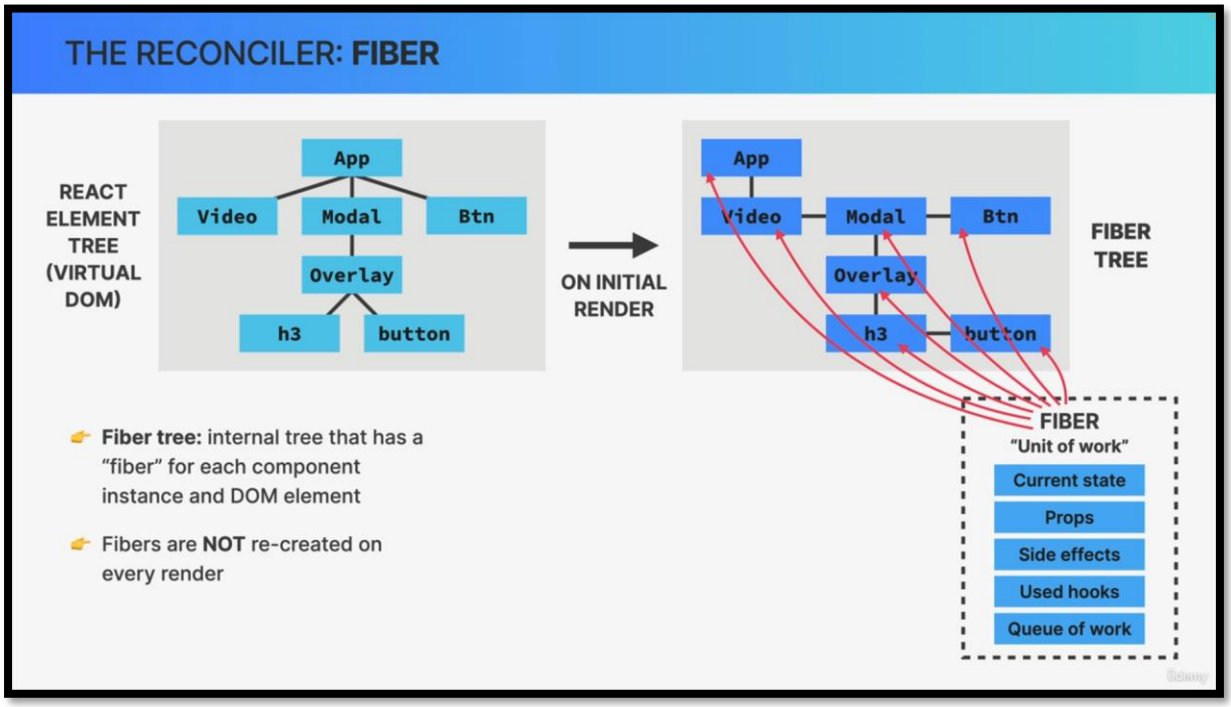
The current reconciler in React is called Fiber.

During the initial render of the application Fiber takes the entire React element tree or 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 have a Fiber.

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's simply mutated over and over again in future reconciliation steps.

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.

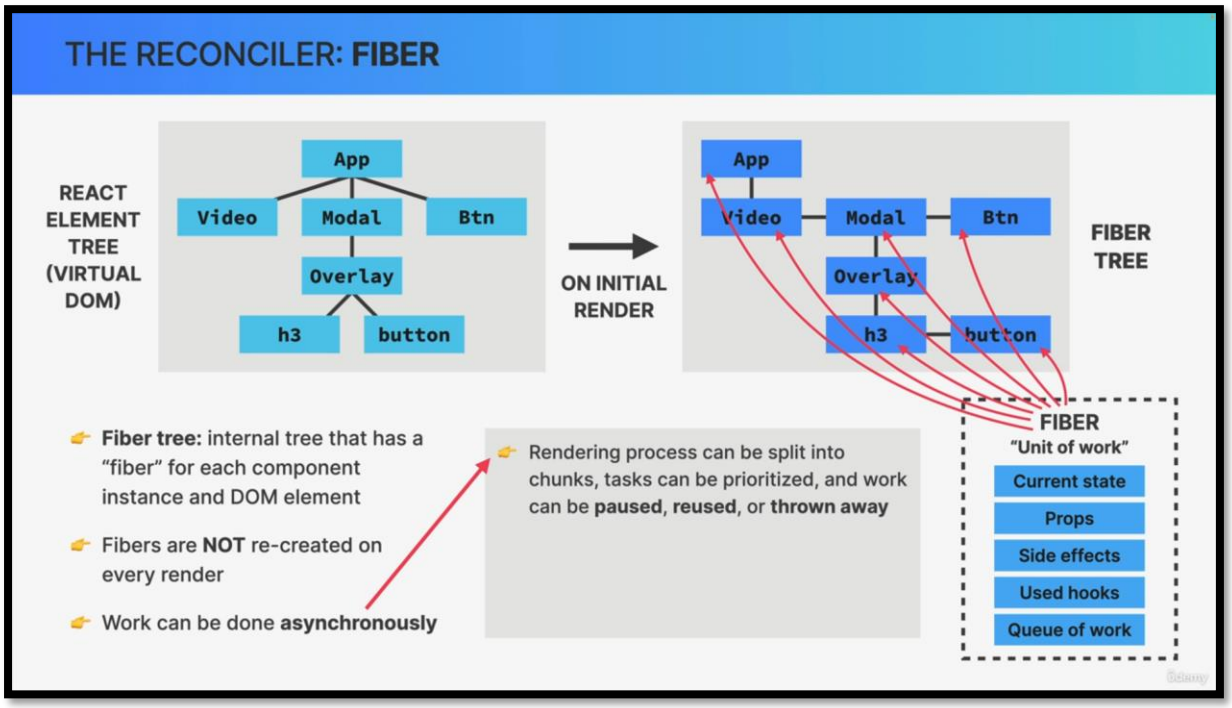


So, the actual state and props of any component instance that we see on the screen are internally stored inside the corresponding Fiber in the Fiber tree. Each Fiber also contains a queue of work to do like updating state, updating refs, running registered side effects, performing DOM updates and so on. This is why a Fiber is also defined as a unit of work.

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.

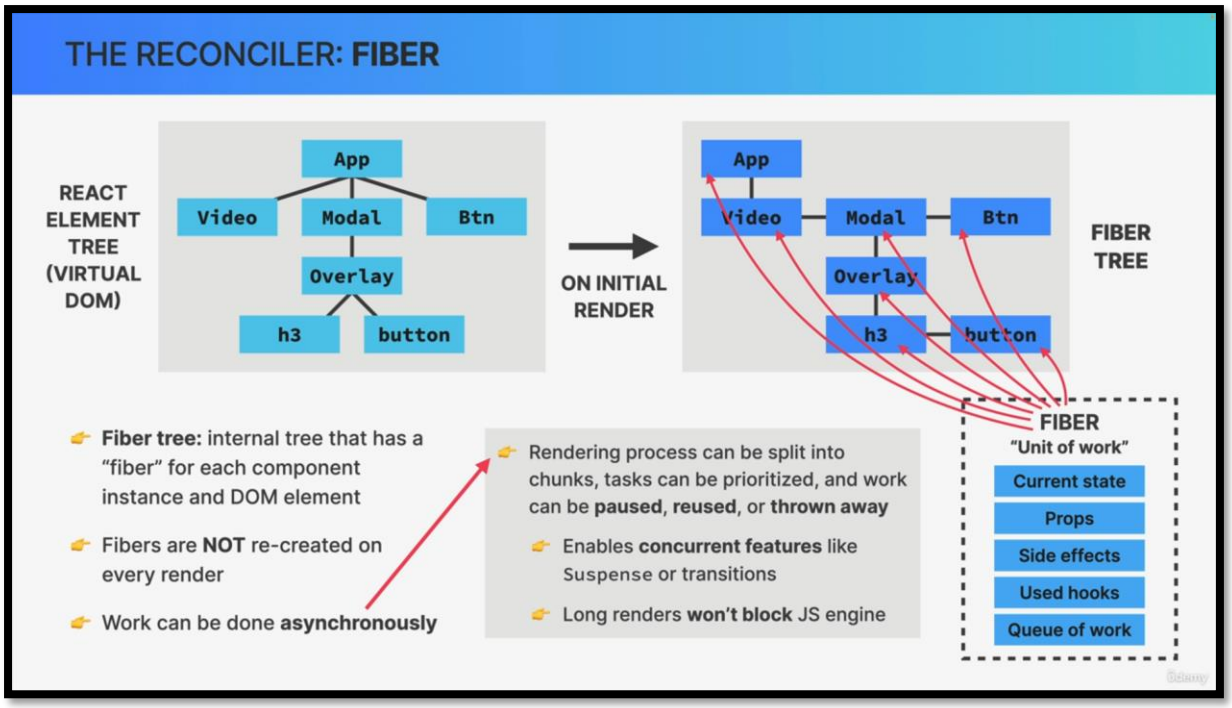
So instead of a normal parent-child relationship, each first child has a link to its parent and all the other children then have a link to their previous sibling.

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.



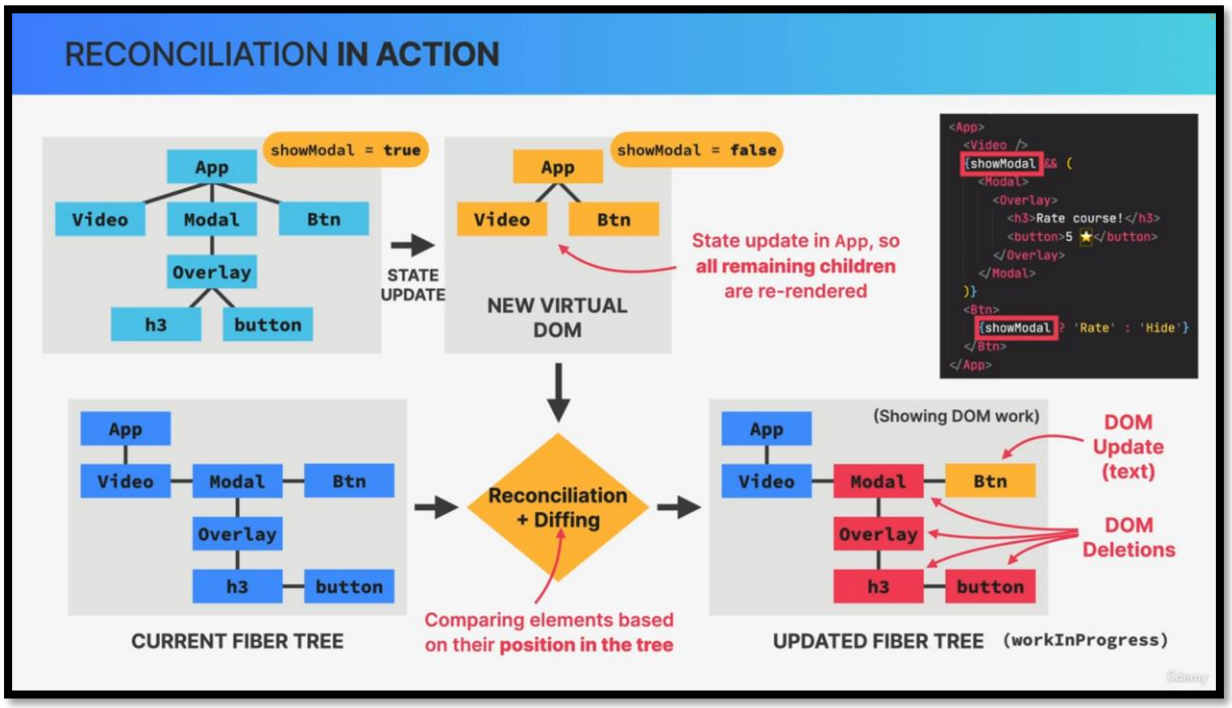
One extremely important characteristic of the Fiber reconciler is that work can be performed asynchronously. This means that the rendering process which is what the reconciler does, can be split into chunks, some tasks can be prioritized over others and work can be paused, reused, or thrown away if not valid anymore.

All this happens automatically behind the scenes. It's completely invisible to us developers.



Some practical uses of this asynchronous rendering enables modern so-called concurrent features like Suspense or transitions starting in React 18.

It also allows the rendering process to pause and resume later so that it won't block the browser's JavaScript engine with two long renders, which can be problematic for performance in large applications. Again, this is only possible because the render phase does not produce any visible output to the DOM yet.



In the app component, there is a piece of state called showModal, which is currently set to true and let's say now that the state is updated to false.

This will then trigger a re-render which will create a new virtual DOM. In this tree, the modal and all its children are actually gone because they are no longer displayed when showModal is not true.

All remaining React elements are yellow, meaning that all of them were re-rendered. It's because all children of a re-rendered element are re-rendered as well.

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. This process of comparing elements step-by-step based on their position in the tree is called diffing.

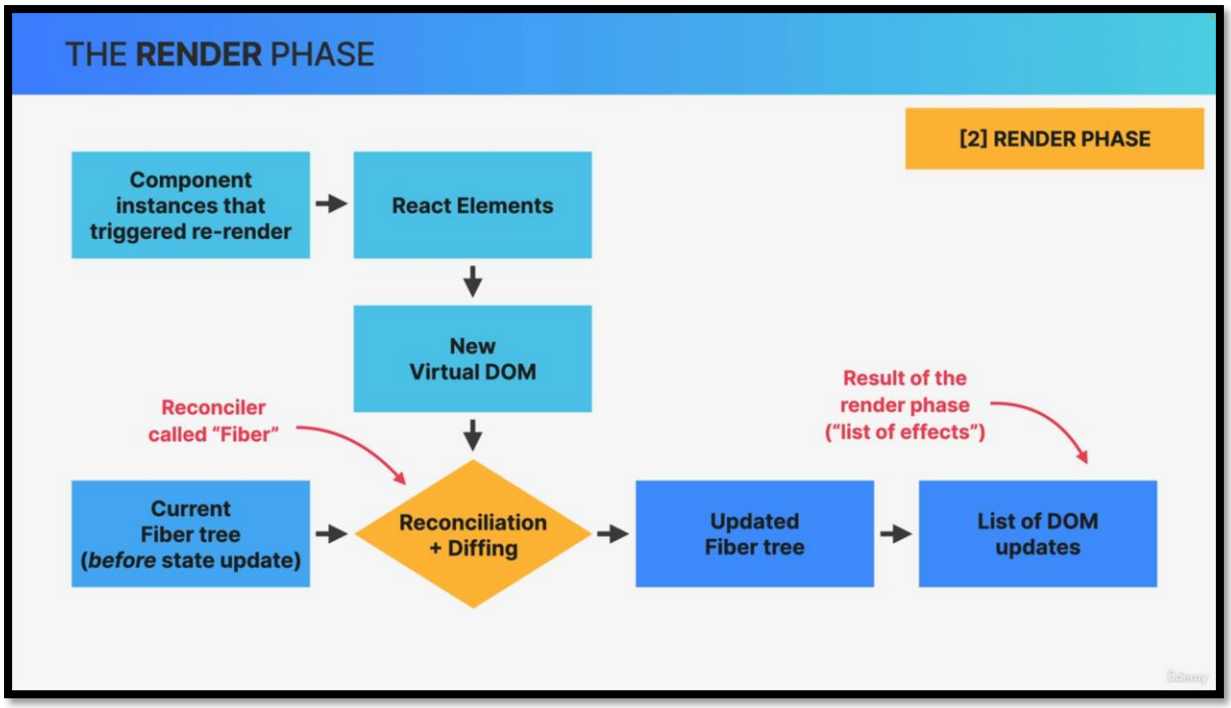
DOM Mutations

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 hide to rate.

Then we have the Modal, Overlay, h3 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.

As a result of reconciliation, the DOM will not be updated in this case. Now, once this process is over, all these DOM mutations will be placed into a list called the list of effects which will be used in the commit phase, to actually mutate the DOM.



The results of the reconciliation process is a second updated Fiber tree, plus basically a list of DOM 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.