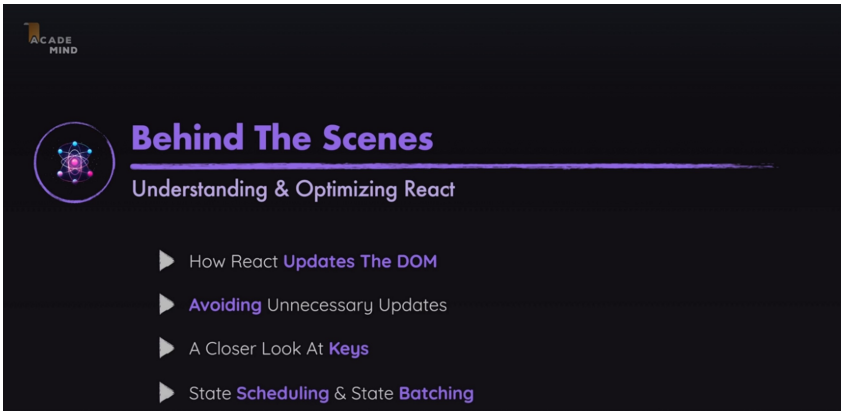
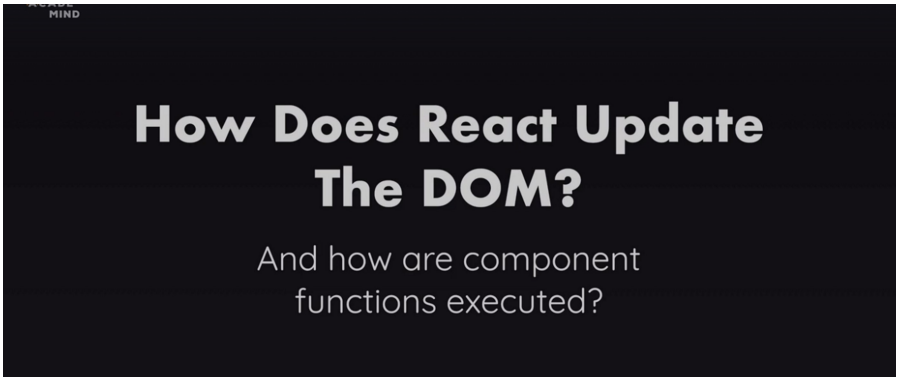
Section 13: A Look Behind The Scenes Of React & Optimization Techniques

Saturday, October 12, 2024 11:04 AM





Having a good understanding of how React actually check your components and derives the actual DOM from those components, Having that understanding is crucial as a React Developer because it will allow you to write better code and better applications.

Rendering the component in the screen means that React goes ahead and executes the component function .

```
import App from './App.jsx';
import './index.css';

ReactDOM.createRoot(document.getElementById('root')).render(<App />);
```

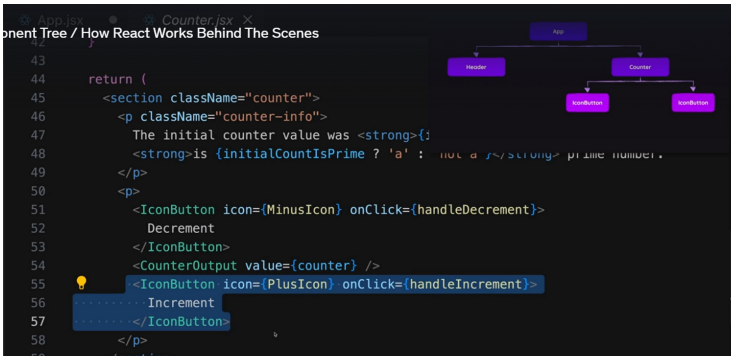
Two pieces of state are registered for this component. And these two functions here get created. (not executed yet)

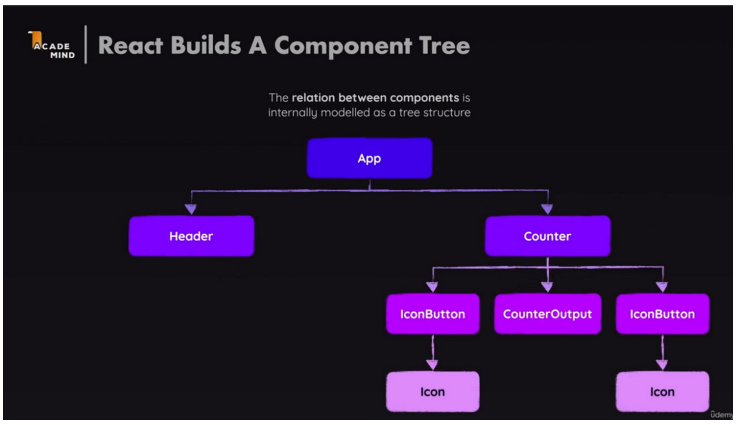
```
function App() {
  log('<App /> rendered');

  const [enteredNumber, setEnteredNumber] = useState(0);
  const [chosenCount, setChosenCount] = useState(0);

  function handleChange(event) {
    setEnteredNumber(+event.target.value);
  }

  function handleSetClick() {
    setChosenCount(enteredNumber);
    setEnteredNumber(0);
  }
```

Then this JSX code here gets executed. And in the end it can be returned by this component.
Every component function must return something that can be rendered. Typically JSX code, sometimes also a portal. This JSX code in the end translated to JavaScript code and translated to actual elements that can be rendered on the screen. That's what React is doing.

```
  return (
    <>
      <Header />
      <main>
        <section id="configure-counter">
          <h2>Set Counter</h2>
          <input type="number" onChange={handleChange} value={enteredNumber} />
          <button onClick={handleSetClick}>Set</button>
        </section>
        <Counter initialCount={chosenCount} />
      </main>
    </>
  );
}
```

Header and Counter are custom component. React, first execute <Header/> Component

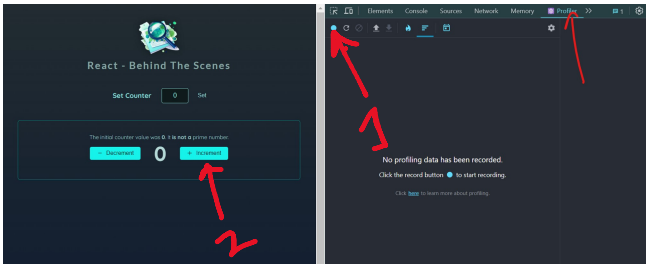




Analyzing Component Function Executions via React's DevTools Profiler

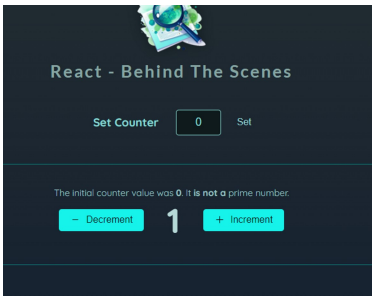
Install React Developer Tools

Then press F12 and go to **Profiler section.**

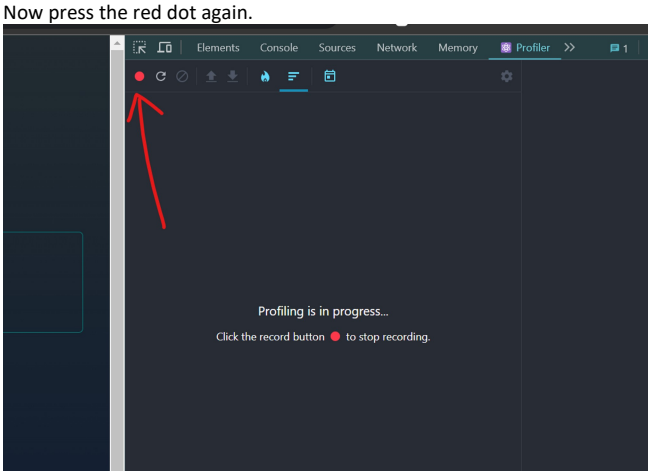


Follow step 1 and 2

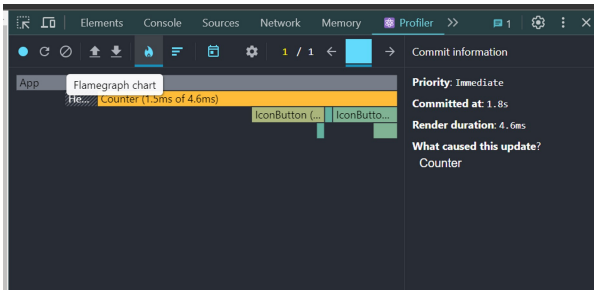
And clicking increment button, you can see something like this

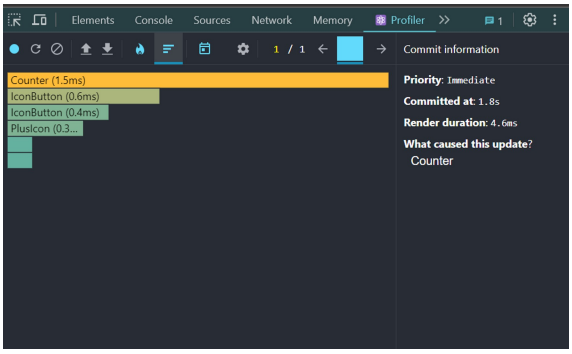


Now press the red dot again.

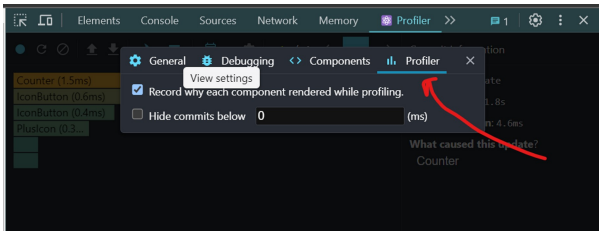


Now you should see which component has been rendered or not.

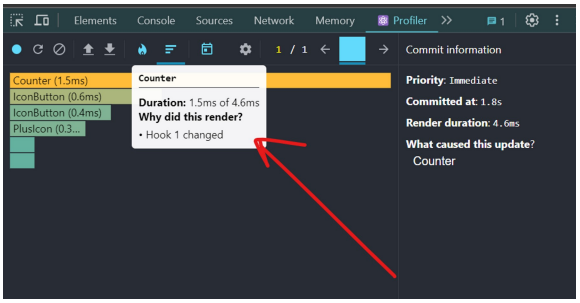




Go to settings from and make sure this checkbox is checked



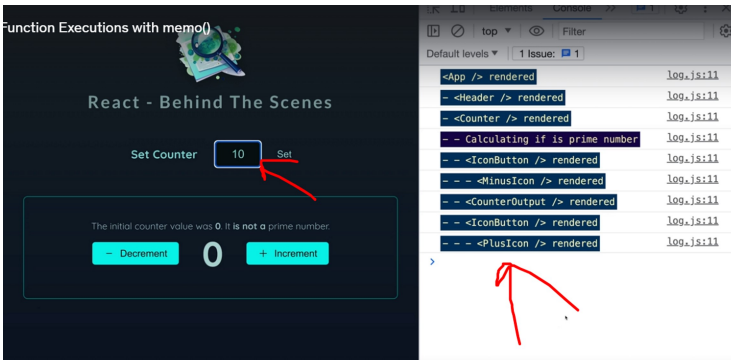
Now, you should be able to see the reason behind each rendered component.



Avoiding Component Function Executions with memo()

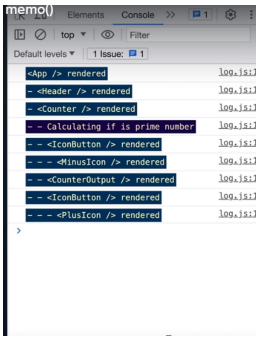
How can we optimize React Application with that knowledge of how different components are related and are being executed?

In our demo application, if we write something, we can see in the console, which components are being triggered.



This happens because, state changes in App component and in a result child component gets executed again.

```
  function handleSetClick() {
    setChosenCount(enteredNumber);
    setEnteredNumber(0);
  }

  return (
    <>
      <Header />
      <main>
        <section id="configure-counter">
          <h2>Set Counter</h2>
          <input type="number" onChange={handleChange} value={enteredNumber} />
          <button onClick={handleSetClick}>Set</button>
        </section>
        <Counter initialCount={chosenCount} />
      </main>
    </>
```

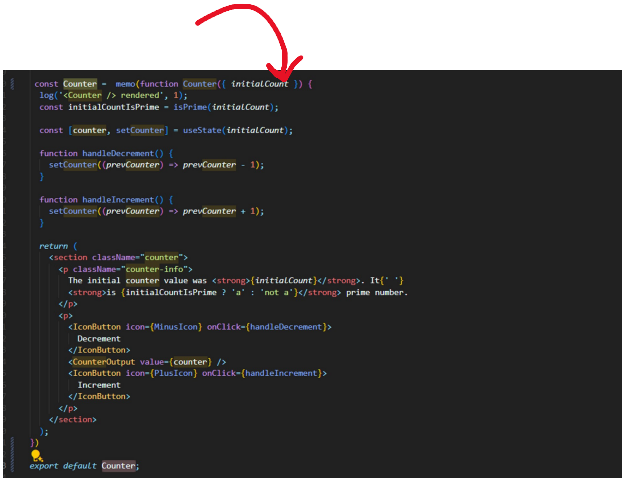


This does not necessarily have an impact on the actual DOM that's rendered.
So it's not that bad. But still it is not optimal because it is a bunch of code that's executed.

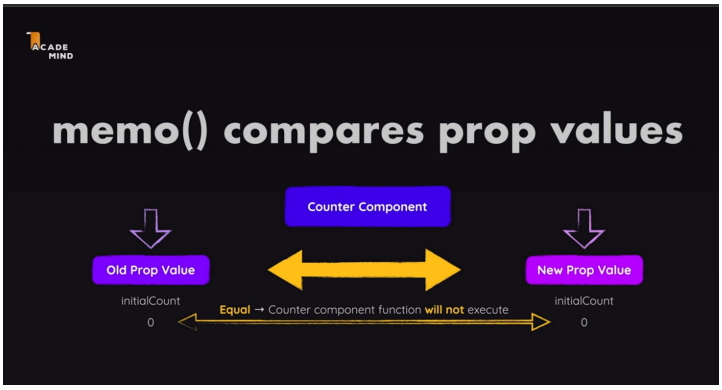
How can we fix this problem?

One solution could be using memo function. What memo does ?
It will take a look at the props of your component function.

Whenever the component function would normally execute again, For example, app component function executes
Memo will take a look at the old prop value, and the new prop value that would be receive now, if those prop value are exactly the same,

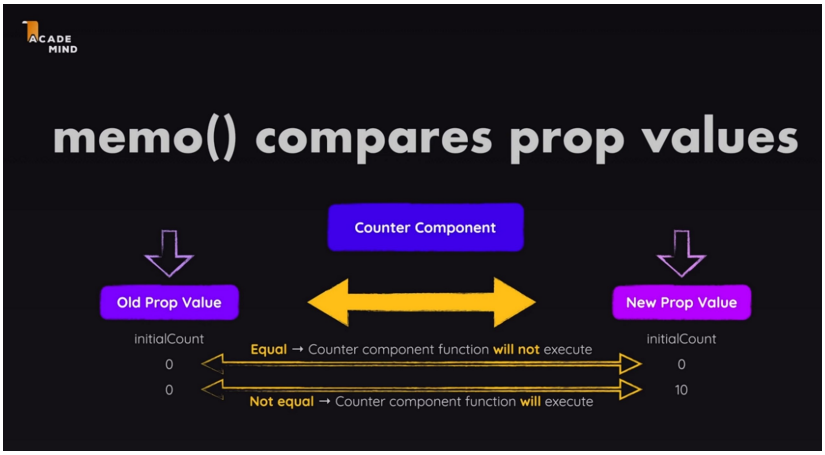


```
const Counter = memo(function Counter({ initialCount }) {
  log('<Counter /> rendered', 1);
  const initialCountIsPrime = isPrime(initialCount);

  const [counter, setCounter] = useState(initialCount);

  function handleDecrement() {
    setCounter((prevCounter) => prevCounter - 1);
  }

  function handleIncrement() {
    setCounter((prevCounter) => prevCounter + 1);
  }

  return (
    <section className="counter">
      <p className="counter-info">
        The initial counter value was <strong>{initialCount}</strong>. It{' '}
        <strong>is {initialCountIsPrime ? 'a' : 'not a'}</strong> prime number.
      </p>
      <p>
        <IconButton icon={MinusIcon} onClick={handleDecrement}>
          Decrement
        </IconButton>
        <CounterOutput value={counter} />
        <IconButton icon={PlusIcon} onClick={handleIncrement}>
          Increment
        </IconButton>
      </p>
    </section>
  );
})
export default Counter;
```



Counter Component function execution will be prevented by memo.

```
  return true;
});

const Counter = memo(function Counter({ initialCount }) {
  log('<Counter /> rendered', 1);
  const initialCountIsPrime = isPrime(initialCount);

  const [counter, setCounter] = useState(initialCount);

  function handleDecrement() {
    setCounter((prevCounter) => prevCounter - 1);
  }

  function handleIncrement() {
    setCounter((prevCounter) => prevCounter + 1);
  }
```

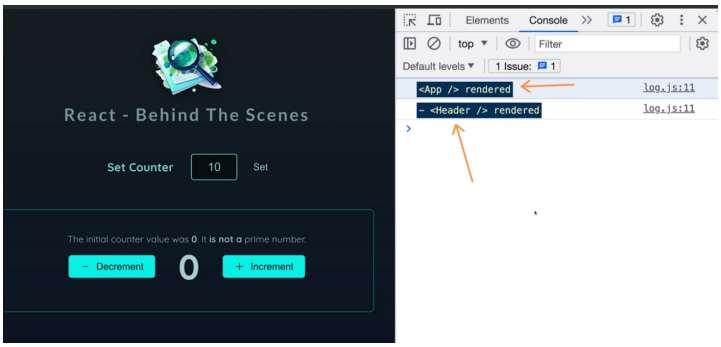


This <Counter/> component function will only be executed if the 'initialCount' changed or the state changed.

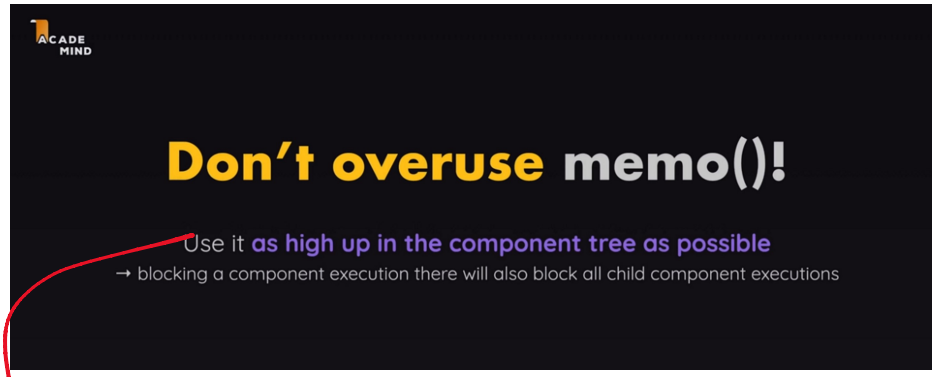
```jsx
const Counter = memo(function Counter({ initialCount }) {
  log('<Counter /> rendered', 1);
  const initialCountIsPrime = isPrime(initialCount);

  const [counter, setCounter] = useState(initialCount);

  function handleDecrement() {
    setCounter((prevCounter) => prevCounter - 1);
  }

  function handleIncrement() {
    setCounter((prevCounter) => prevCounter + 1);
  }

  return (
    <section className="counter">
      <p className="
        The initial    const initialCountIsPrime: boolean    nt}</strong>. It{' '}
        <strong>is {initialCountIsPrime ? 'a' : 'not a'}</strong> prime number.
      </p>
      <p>
        <IconButton icon={MinusIcon} onClick={handleDecrement}>
          Decrement
        </IconButton>
        <CounterOutput value={counter} />
        <IconButton icon={PlusIcon} onClick={handleIncrement}>
          Increment
        </IconButton>
      </p>
    </section>
  );
})

export default Counter;
```

That's not affected by memo. Memo only prevents function execution that are triggered by the parent component. Here in case the <App/> component.

Now only the App Componet and Header component is being executed.

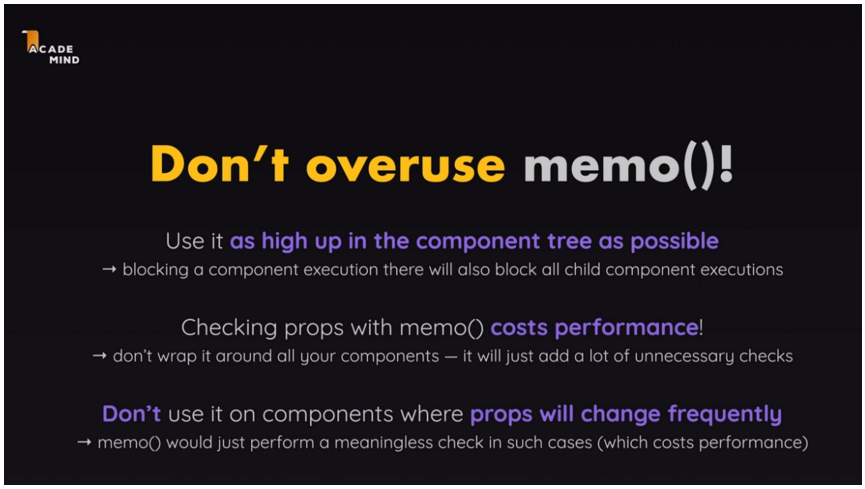


Now you might be thinking to wrap all your component with memo() hook? Because it can be useful. Right?
But you should not wrap around all your components.



Here in our case, it is <Counter/> component. In a result, all the child and sub-child of <Counter/> component will not be executed. If you would wrap memo around all your components, that would simply mean, React always has to check the props before it executes the component function.
And of course checking the prop values for equality also costs some performance.

And that's specially a problem if you wrap a function that almost always gets changed props.



So, you should use it with care. And simple app like this one, you don't need to use it at all.

Avoiding Component Function Executions with Clever Structuring

So, 'memo if use with care, can be useful but it's not the only way of preventing unnecessary render.
Another technique that is often more powerful than memo is a clever component composition in your application.

For example, in the <App/> component we simply could put this into separate component.

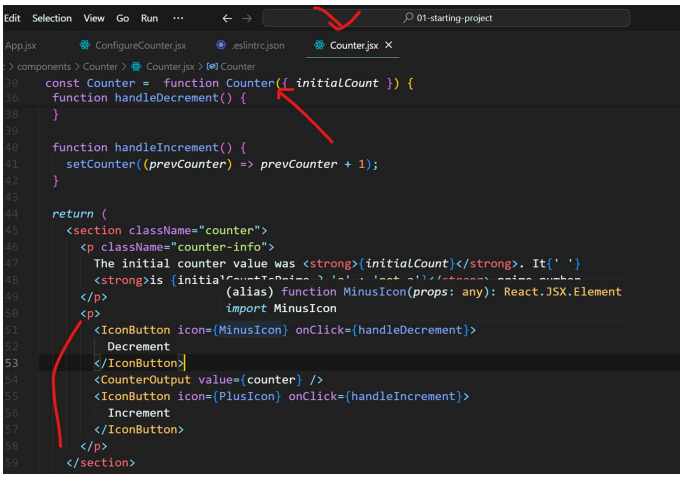
```
      setEnteredNumber(0);
    }

    return (
      <>
        <Header />
        <main>
          <section id="configure-counter">
            <h2>Set Counter</h2>
            <input type="number" onChange={handleChange} value={enteredNumber} />
            <button onClick={handleSetClick}>Set</button>
          </section>
          <Counter initialCount={chosenCount} />
        </main>
      </>
```

Then this state that changes on every keystroke would also live in that separate component and
Would not affect the <App/> component.

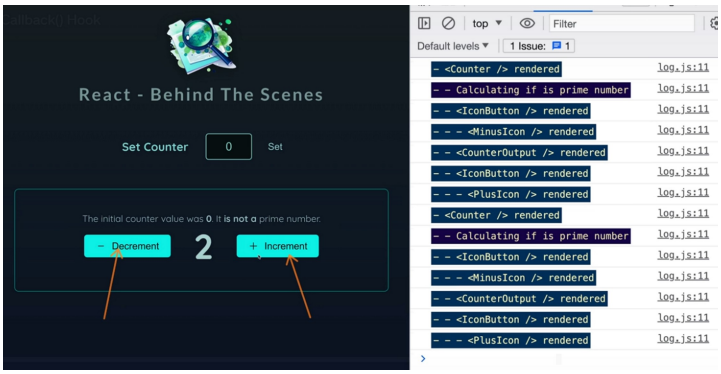
```
function App() {
  log('<App /> rendered');

  const [enteredNumber, setEnteredNumber] = useState(0);
  const [chosenCount, setChosenCount] = useState(0);

  function handleChange(event) {
    setEnteredNumber(+event.target.value);
  }

  function handleSetClick() {
    setChosenCount(enteredNumber);
    setEnteredNumber(0);
  }
```

Understanding the useCallback() hook

We are in the Counter component.

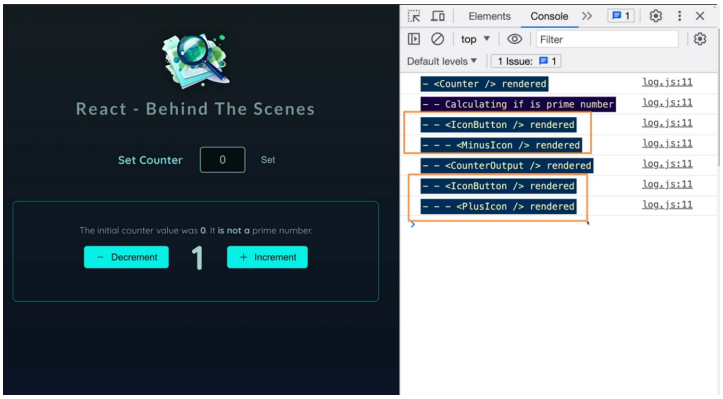


When I click the ***increment*** or ***decrement*** button, those buttons stays same and does not change. Only Then number is changing. So why should we re-render the Increment or Decrement button?



So, let's wrap <IconButton/> component with memo to prevent unnecessary render.

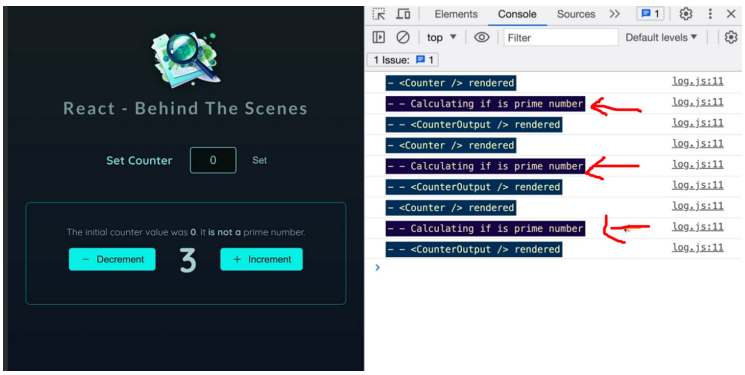
Save it and reload, still you would see this in the console. Meaning, our 'memo' somehow is not working.
From our understanding, *memo* *wraps component. It re-execute component if the prop value changes with the old prop value.*



So, let's analyze our prop value.

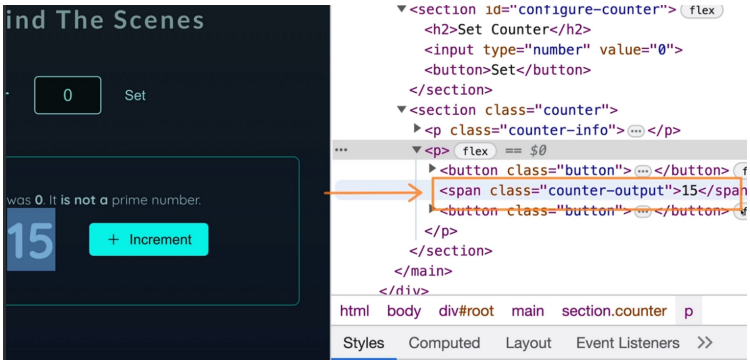
Understanding the useMemo() Hook

If you look at the console, you will notice that, 'Calculating if is prime number' is executing every time we click on the 'Decrement' or 'Increment' button.

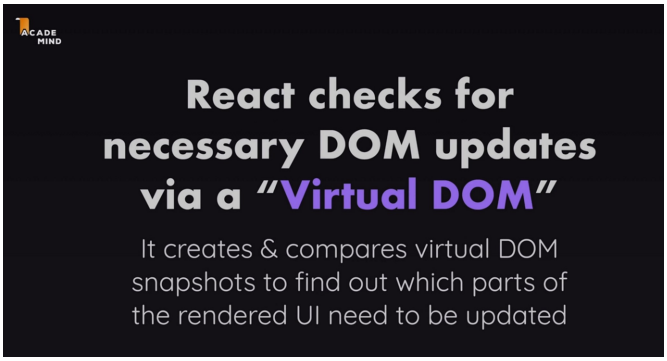


React Uses a Virtual DOM - Time to Explore it!

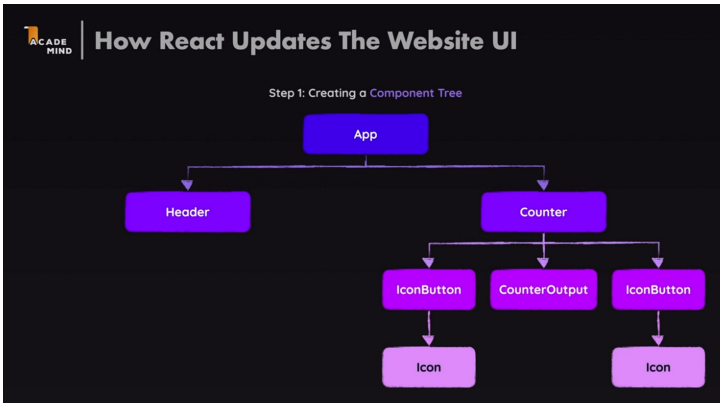
Whenever, I click on the increment or decrement button, of course in react component re-evaluates.
But it does not mean that old code is thrown away. React only updates where the change is needed.
Because it is efficient. All other DOM elements are not touched by React.



This is happening because React works with so-called virtual DOM.



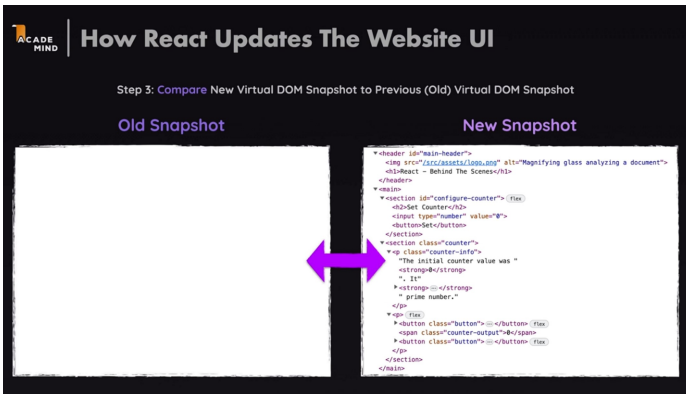
When I reload my react app, react creates this component tree and that actually derives the actual HTML code that Should be rendered.



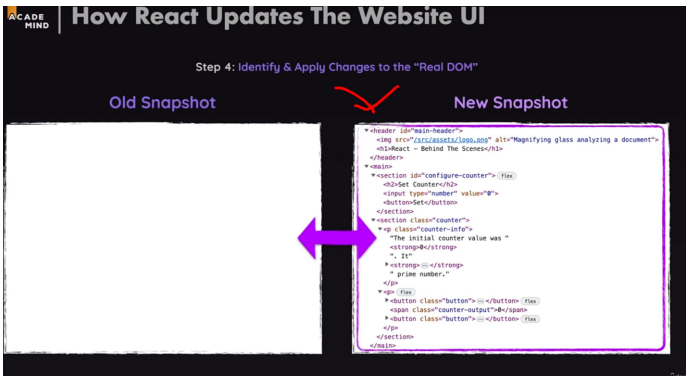
And it then creates a virtual DOM snapshot.
So it's not reaching out to the real DOM yet. Instead, it just creates a virtual representation of how that real DOM should look like.

As a next step, React then compares that to that last virtual DOM snapshot it created.

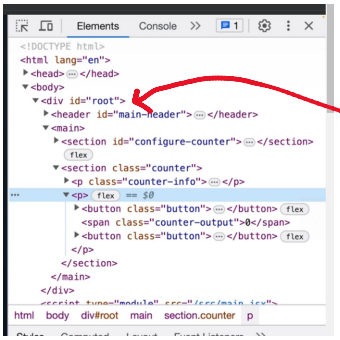
If the app just started, there is no last snapshot.



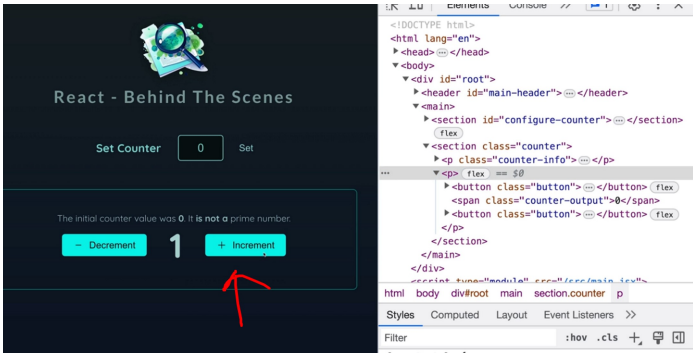
And therefore, React of course, sees that everything changed. And it goes ahead to the real DOM and Makes those changes.



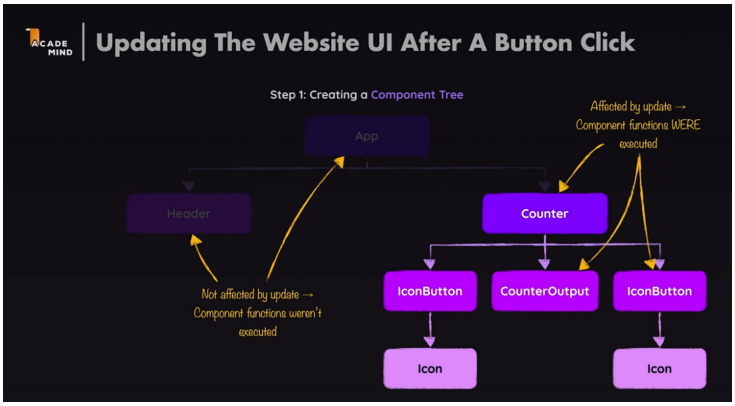
Which means in this case, the entire virtual DOM is inserted into the real DOM into this div with the id ***root.***



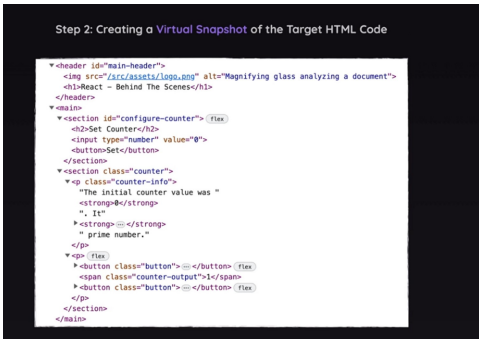
So, then I click this increment button, react actually repeats this process.



It again creates that component tree. It can quickly find out that only a part of that tree changed.
That only some of those component functions were executed.

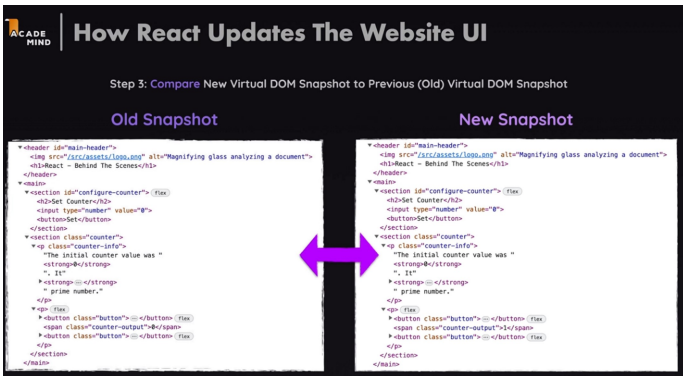


Then React derives the updated HTML code

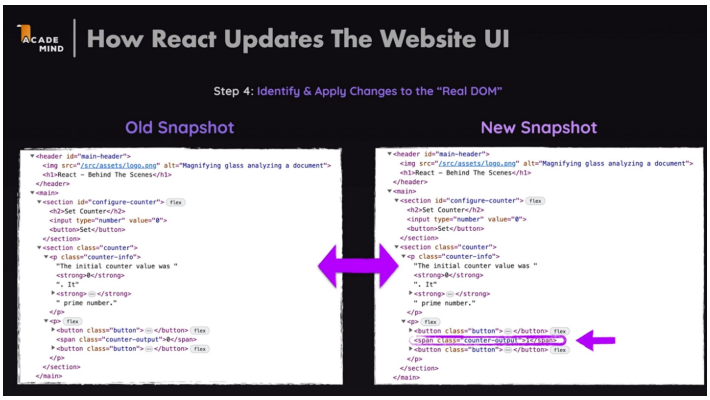


Then compares the updated code with the old code.

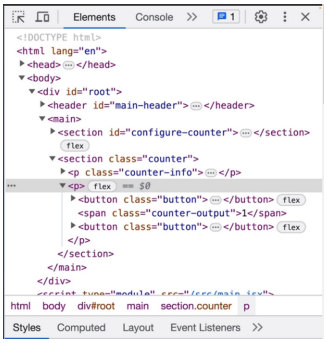
Still only virtually, it does not compare anything from the real DOM. It just compares the new virtual DOM Snapshot with the new virtual DOM snapshot. It then quickly finds out which parts changed.



In our case, only the text of the

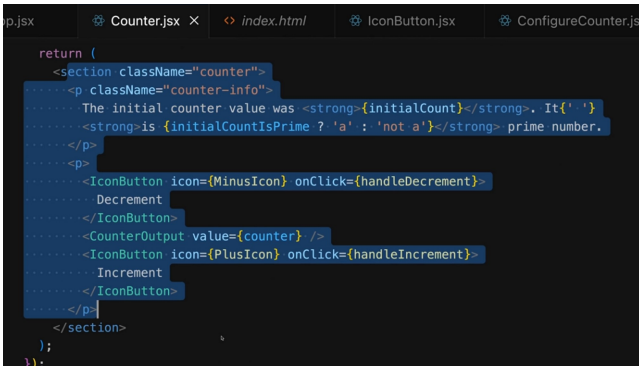


And as a next step, React goes ahead and applies those changes to the Real DOM.
In this case, it only goes to that span in the real DOM and replaces it's old text with the next Text and nothing else.



This is super important to understand!
Just because a component function executes and it's JSX code therefore is evaluated does not mean that this code is inserted or updated in the real DOM.

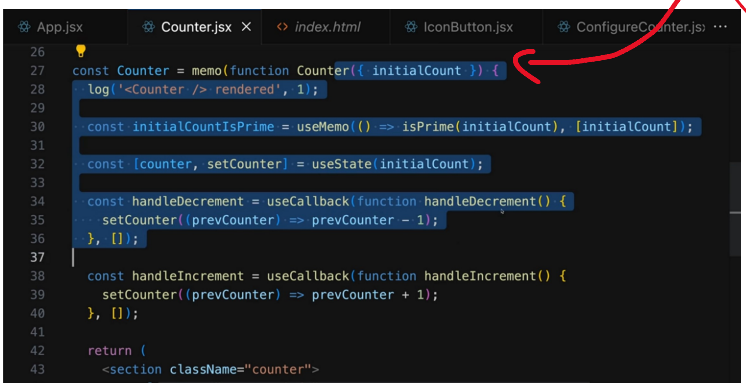
Because all those real DOM operation are quite performance intensive!
And therefore React does not change the real dom all the time and instead it just
Creates these snapshots and compares them and only makes the necessary changes to get to the target Result.



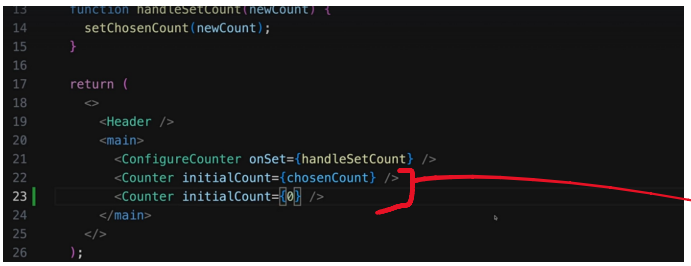
Why Keys Matter When Managing State!

The state you register in a component function is scoped to that component you could say and also recreated when you reuse this component.

The state you register in a component function is scoped to that component you could say and also recreated when you reuse this component.

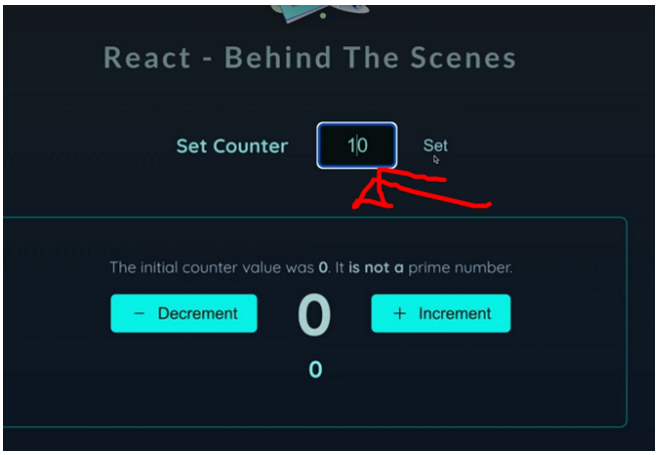


For example, in the <App/> component, every counter receives its own counter state.

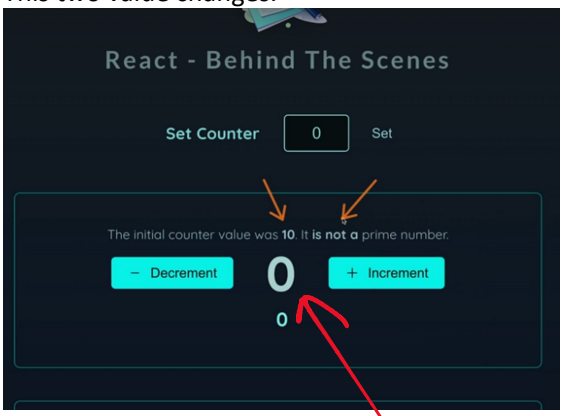


Using keys For Resetting Components

If I change this value and click set,



This two value changes.



But this value does not change. But our plan is also reset the counter when we click set button.

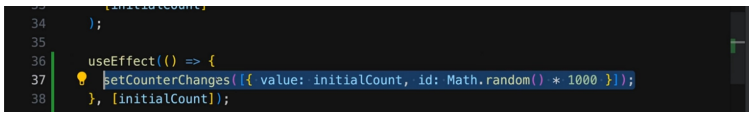
But that's now happening because this initial count value is really just used as an initial value for the state when it's being initialized.

It really just used for initialization, nothing else
And that's why future initial account changes are ignored.

The initial value you pass to useState is never used after the first execution.

But what if you want to reset the counter if this prop changes?

One solution could be use 'useEffect' and reset the counter.



But we should avoid useEffect if possible because it also triggers an extra component function. Because useEffect runs after the component function execution.

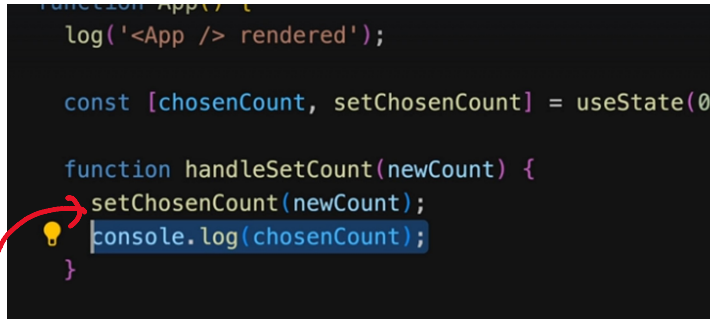
Better ways is:

Adding a key. Because key forces react to unmount the old component and add a new one, It completely creates a new instance of <Counter/> component.

State Scheduling & Batching

When we update the state, state update will be scheduled by React. It will not be executed instantly.

That's why if we console log here, we will get the old state



Because this state update is scheduled by React. It will trigger a new component function execution and the new state will be available the next time this executes.

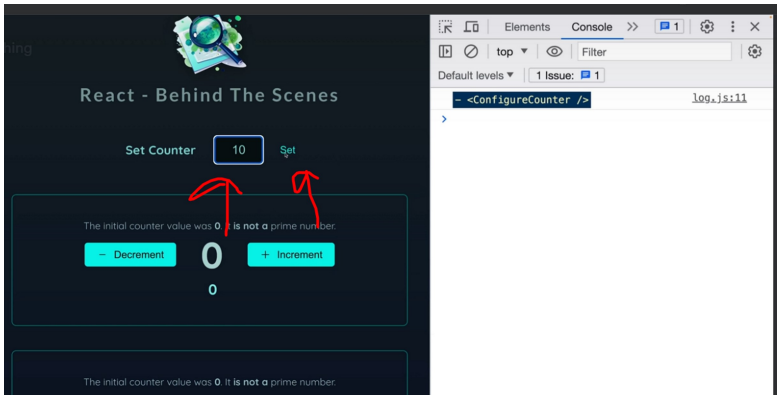
That is why we use this function form to update state which depends on old state.

React guarantees us we will always get the latest state snapshot available.

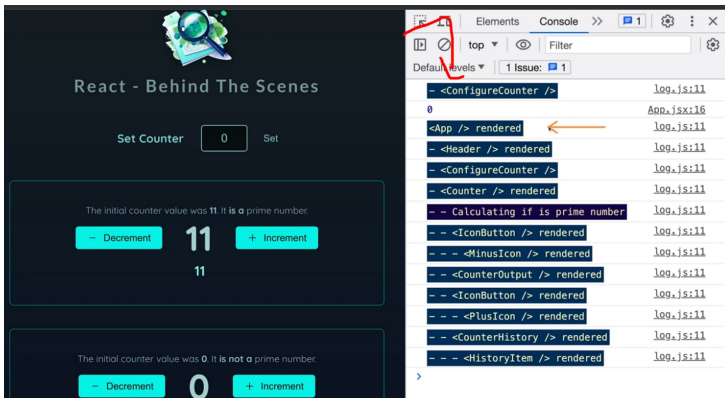
```
function App() {
  log('<App /> rendered');

  const [chosenCount, setChosenCount] = useState(0);

  function handleSetCount(newCount) {
    setChosenCount(newCount);
    setChosenCount((prevChosenCount) => prevChosenCount + 1);
    console.log(chosenCount); // won't work!
  }

  return (
    <>
```

If you have multiple state updates, that are in the end triggered simultaneously. You will not end up with multiple component function executions (In our case it is 2). Because that would of course be pretty inefficient.

If you enter a value and click set



You see the app component function is only executed once. Because React performs state batching. Which simply means multiple state updates that are triggered from the same function are batched together and will only lead to one component function execution.



Optimizing React with MillionJS

You can use this package to optimize the performance of your React applications.