React hooks

More hooks, custom hooks

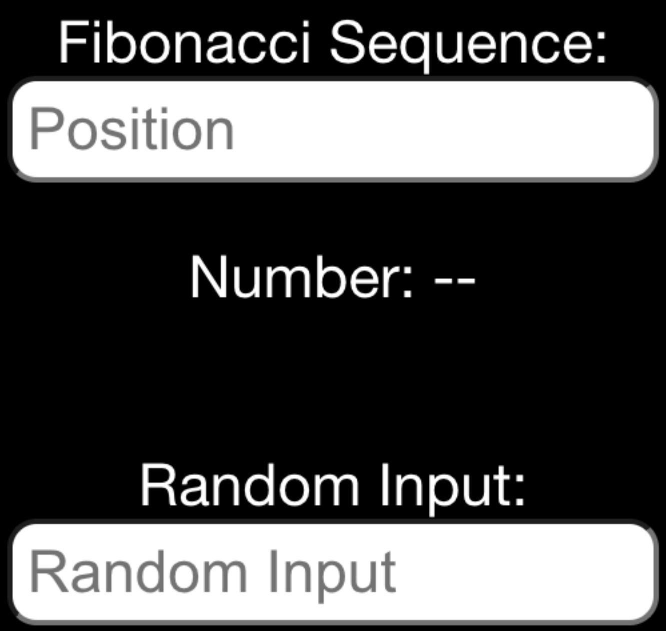
Day 2 - Fundamentals of Frontend Development

Overview

- `useState`
- `useEffect`
- `useCallback`
- `useMemo`
- `useReducer`
- `useTransition`
- `useDeferredValue`
- `useRef`

useMemo

memoise the output of a function


Fibonacci Sequence:

Position

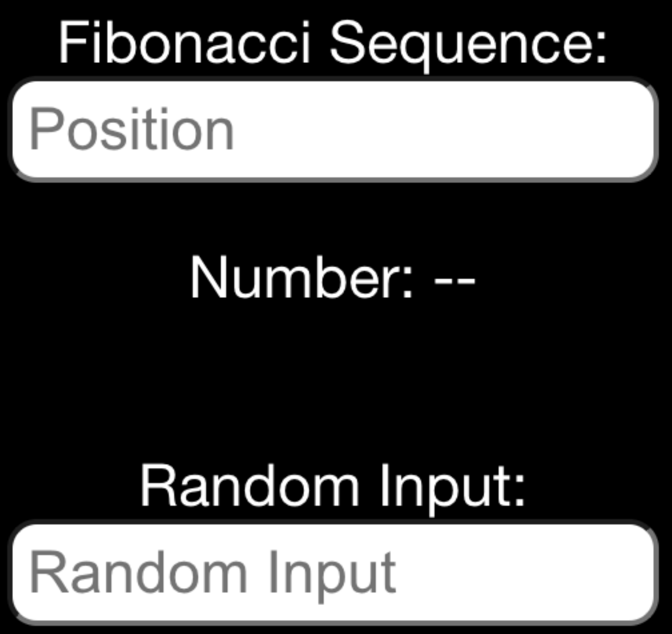
Number: --

Random Input:

Random Input

Without useMemo

- `fib()` re-computed each render
- Not responsive


Fibonacci Sequence:

Position

Number: --

Random Input:

Random Input

Recap: Referential equality

primitive value vs array

```
const fibNumber = fib(userNumber);
const myArray = getArray();

 useEffect(() => {
   console.log("New array");
 }, [myArray]);


 useEffect(() => {
   console.log("new number");
 }, [fibNumber]);
```

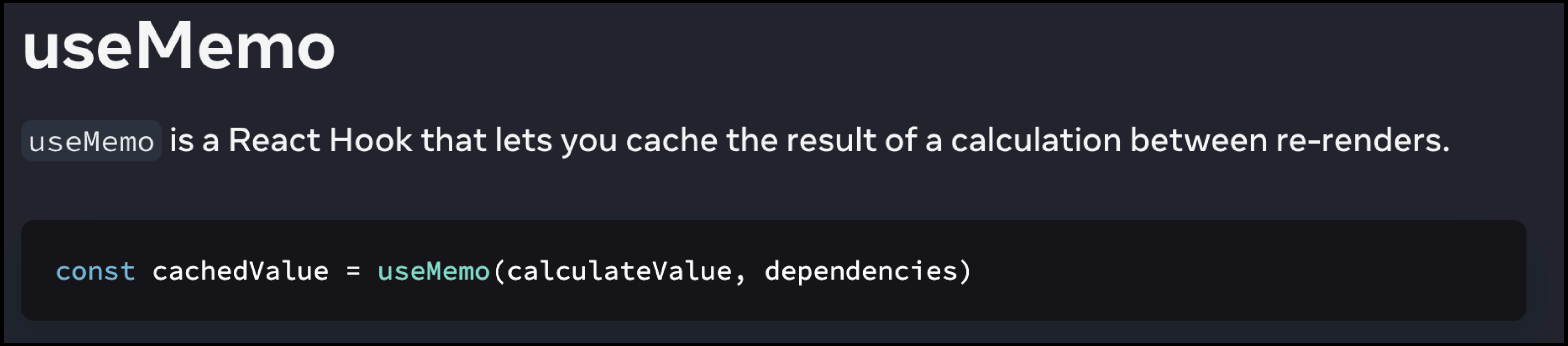
Recap: Referential equality

memoize function defined outside of `App()`

```
const myArray = useMemo(() => getArray());
useEffect(() => {
  console.log("New array");
}, [myArray]);
```

vs

```
const myArray = useMemo(() => getArray(), []);
useEffect(() => {
  console.log("New array");
}, [myArray]);
```



useMemo

`useMemo` is a React Hook that lets you cache the result of a calculation between re-renders.

```
const cachedValue = useMemo(calculateValue, dependencies)
```

With useMemo

- **function** `fib()` **defined outside** `App()`
- `fibNumber` is not re-computed each render
- `fibNumber` is the **output** of `fib()`

With useMemo

`fib()` inside `App()`

```
function App() {
  const [userNumber, setUserNumber] = useState("");
  const [randomInput, setRandomInput] = useState("");

  const fib = (n) => {
    return n <= 1 ? n : fib(n - 1) + fib(n - 2);
  };

  const fibNumber = useMemo(() => fib(userNumber), [userNumber, fib]);
....


}
```

With useMemo + useCallback

`fib()` **inside** `App()` **must** be paired with `useCallback()`

```
function App() {
  const [userNumber, setUserNumber] = useState("");
  const [randomInput, setRandomInput] = useState("");

  const fib = useCallback((n) => {
    return n <= 1 ? n : fib(n - 1) + fib(n - 2);
  }, []);


  const fibNumber = useMemo(() => fib(userNumber), [userNumber, fib]);
....


}
```

useMemo in a loop

```
function ReportList({ items }) {
  return (
    <article>
      {items.map(item => {
        const data = useMemo(() => calculateReport(item), [item]);
        return (
          <figure key={item.id}>
            <Chart data={data} />
          </figure>
        );
      })}
    </article>
  );
}
```

Cannot useMemo in a loop

```
function ReportList({ items }) {
  return (
    <article>
      {items.map(item =>
        <Report key={item.id} item={item} />
      )}
    </article>
  );
}

function Report({ item }) {
  // ✅ Call useMemo at the top level:
  const data = useMemo(() => calculateReport(item), [item]);
  return (
    <figure>
      <Chart data={data} />
    </figure>
  );
}
```

just memo

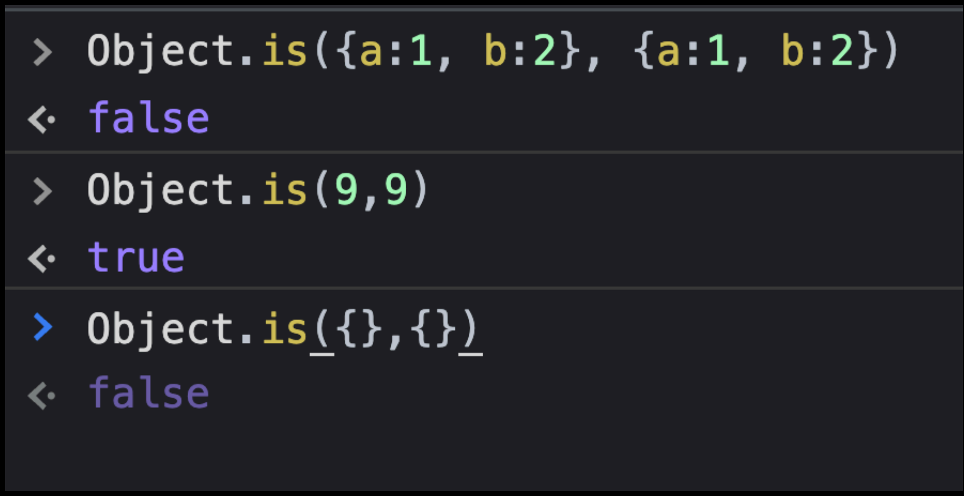
Skip re-rendering when props are unchanged

```
const Fibonacci = memo(function ({ userNumber, setUserNumber }) {
  const fib = useCallback((n) => {
    return n <= 1 ? n : fib(n - 1) + fib(n - 2);
  }, []);

  // log when it's rendered
  console.log("Component rendered at ", Date.now());

  // recomputed each render
  const fibNumber = fib(userNumber);

  return (
    <>
      <label>Fibonacci Sequence:</label>
      <input
        type="number"
        value={userNumber}
        placeholder="Position"
        onChange={(e) => setUserNumber(e.target.value)}
      />
      <p>Number: {fibNumber || "--"}</p>
    </>
  );
});

export default Fibonacci;
```

just memo

Usage

- Still re-render when it's own state change
- Still re-render when a context that it's using changes
- Still re-render when any prop is **not shallowly equal** to what it was previously
- Should accept **minimum necessary info** in the props to optimise



```
> Object.is({a:1, b:2}, {a:1, b:2})
<· false
> Object.is(9,9)
<· true
> Object.is({},{})
<· false
```

```
function Page() {
  const [name, setName] = useState('Taylor');
  const [age, setAge] = useState(42);

  const person = useMemo(
    () => ({ name, age }),
    [name, age]
  );

  return <Profile person={person} />;
}

const Profile = memo(function Profile({ person }) {
  // ...
});
```

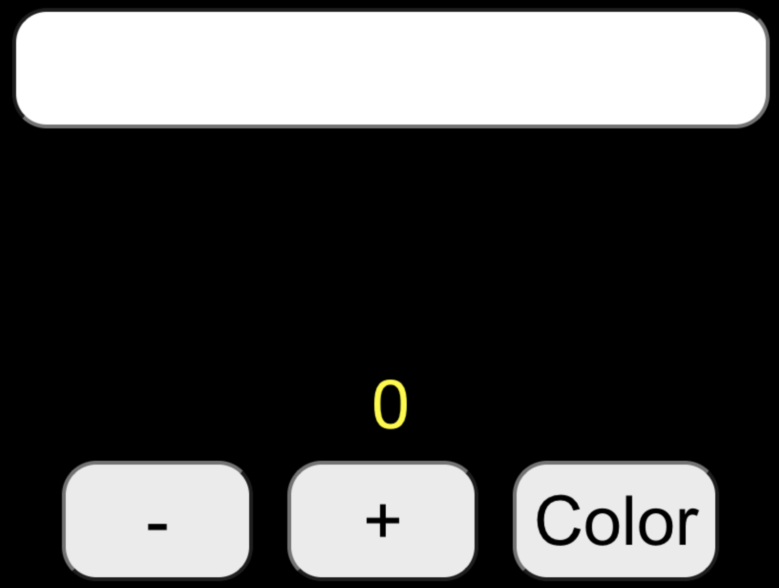
just memo

Custom comparison function

```
const Example = memo(function Example({ dataPoints }) {
  // ...
}, arePropsEqual);


function arePropsEqual(oldProps, newProps) {
  return (
    oldProps.dataPoints.length === newProps.dataPoints.length &&
    oldProps.dataPoints.every((oldPoint, index) => {
      const newPoint = newProps.dataPoints[index];
      return oldPoint.x === newPoint.x && oldPoint.y === newPoint.y;
    })
  );
```

useReducer

consolidate all the state update logic outside your component in a single function



Migrate useState to useReducer

```jsx
// initially use these states, must pass 6 props down
const [userInput, setuserInput] = useState("");
const [count, setCount] = useState(0);
const [color, setColor] = useState(false);

return (
  <main className="App" style={{ color: color ? "#FFF" : "#FFF952" }}>
    <input
      type="text"
      value={userInput}
      onChange={(e) => setuserInput(e.target.value)}
    />
    <p>{count}</p>
    <section>
      <button onClick={() => setCount((prev) => prev - 1)}>-</button>
      <button onClick={() => setCount((prev) => prev + 1)}>+</button>
      <button onClick={() => setColor((prev) => !prev)}>Color</button>
    </section>
    <br />
    <br />
    <p>{userInput}</p>
  </main>
);
```

Define Actions

and Initial State (outside)

```
const initialState = { count: 0, userInput: "", color: false };
```

```
const ACTION = {
  INCREMENT: "increment",
  DECREMENT: "decrement",
  NEW_USER_INPUT: "newUserInput",
  TG_COLOR: "tgColor",
};
```

Action Object can have any shape, typically it's a `string` type

Write **reducer** function

```
const reducer = (state, action) => {
  // do something to our state, based on the action dispatched
  switch (action.type) {
    case ACTION.INCREMENT:
      return { ...state, count: state.count + 1 };
    case ACTION.DECREMENT:
      return { ...state, count: state.count - 1 };
    case ACTION.NEW_USER_INPUT:
      return { ...state, userInput: action.payload };
    case ACTION.TG_COLOR:
      return { ...state, color: !state.color };
    default:
      throw new Error(); // to handle unexpected action
  }
};
```

- Reducers must be **pure**
- Each action describes a **single** user interaction

useReducer

```
// dispatch an action (sending actions with dispatch)
const [state, dispatch] = useReducer(reducer, initialState);
```

```
  value={state.userInput}
  onChange={(e) =>
    dispatch({ type: ACTION.NEW_USER_INPUT, payload: e.target.value })
  }
/>
<br />
<br />
<p>{state.count}</p>
<section>
  <button onClick={() => dispatch({ type: ACTION.DECREMENT })}>-</button>
  <button onClick={() => dispatch({ type: ACTION.INCREMENT })}>+</button>
  <button onClick={() => dispatch({ type: ACTION.TG_COLOR })}>
    Color
  </button>
```

useReducer

Initialization

```
function createInitialState(username) {
  // ...
}

function TodoList({ username }) {
  const [state, dispatch] = useReducer(reducer, createInitialState(username));
  // ...
```

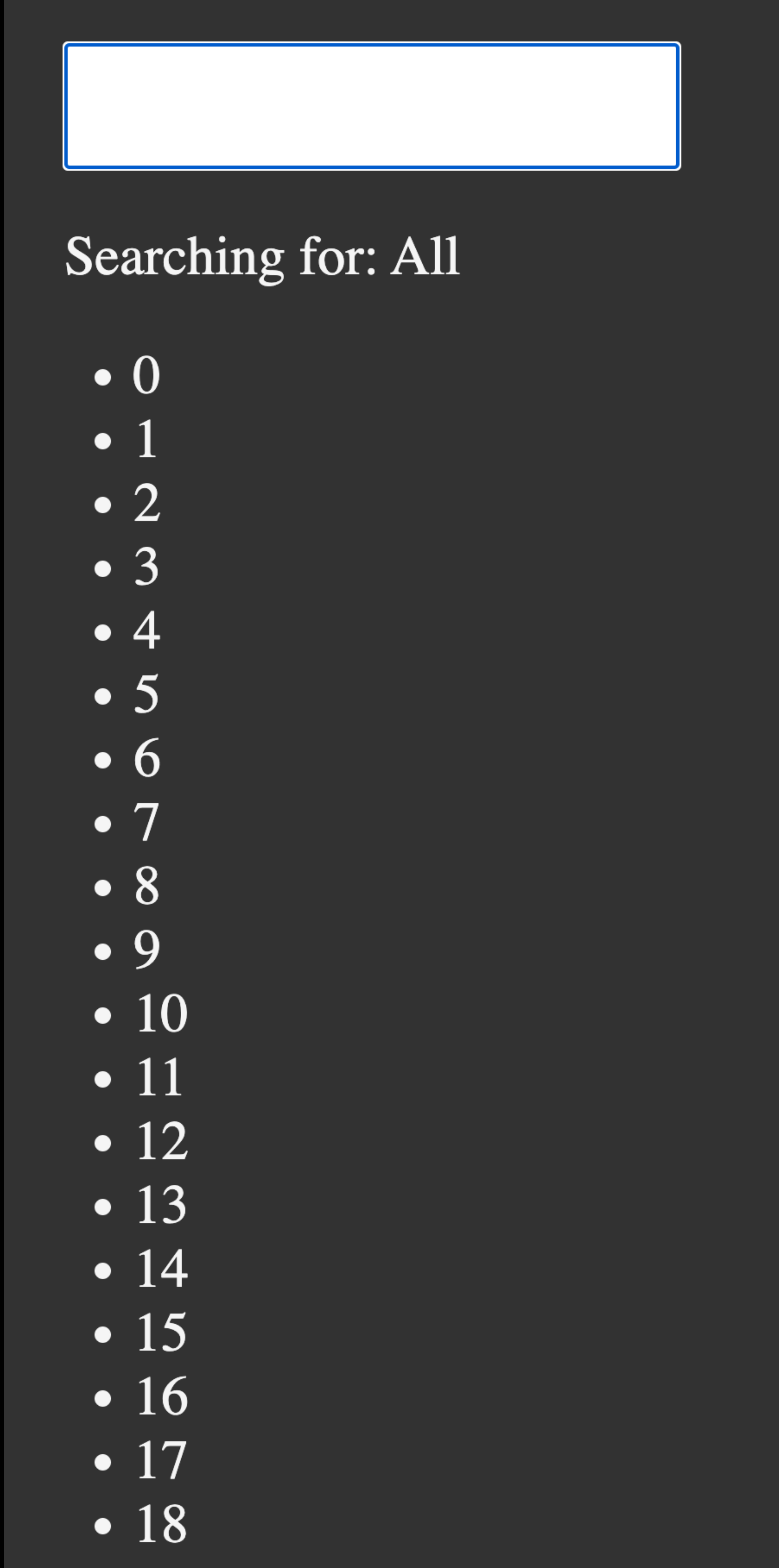
❌

```
function createInitialState(username) {
  // ...
}

function TodoList({ username }) {
  const [state, dispatch] = useReducer(reducer, username, createInitialState);
  // ...
```

✅

useTransition useDeferredValue

For a more responsive App



Searching for: All

- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18

useTransition

update state without blocking the UI.

```
const [count, setCount] = useState(0);
const [items, setItems] = useState([]);
const [isPending, startTransition] = useTransition();

const handleClick = () => {
  // urgent update
  setCount(count + 1);

  // start transition update
  startTransition(() => {
    const myArr = Array(20000)
      .fill(1)
      .map((el, i) => count + 20000 - i);
    setItems(myArr);
  });
};

const content = (
  <div className="App">
    <button onClick={handleClick}>{count}</button>
    {isPending ? <p>Loading...</p> : null}
    <ul>
      {items.map((item) => (
        <li key={item}>{item}</li>
      ))}
    </ul>
  </div>
);
```

useTransition

Can we use it to update input?

```jsx
const [text, setText] = useState('');
// ...
function handleChange(e) {
  startTransition(() => {
    setText(e.target.value);
  });
}
// ...
return <input value={text} onChange={handleChange} />;
```

useTransition

Mixing it with async?

```
startTransition(() => {
  setTimeout(() => {
    setPage('/about');
  }, 1000);
});
```

```
startTransition(async () => {
  await someAsyncFunction();
  setPage('/about');
});
```

The setTimeout() method calls a function after a number of milliseconds.

setTimeout() is an asynchronous function, meaning that the timer function will not pause execution of other functions in the functions stack

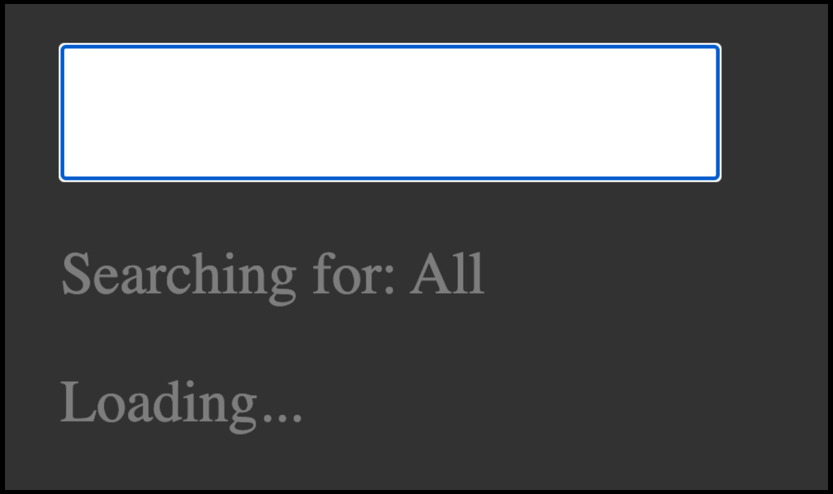
useDeferredValue

```
const [count, setCount] = useState(0);
const [items, setItems] = useState([]);
const deferredCount = useDeferredValue(count);
const deferredItems = useDeferredValue(items);


const handleClick = () => {
   ...
    // do expensive computation of items array at each handleClick()

 };



 <button onClick={handleClick}>{count}</button>
 {isPending ? <p>Loading...</p> : null}
 <p>Deferred: {deferredCount}</p>
 <ul>
   {deferredItems.map((item) => (
     <li key={item}>{item}</li>
   ))}
 </ul>
```

does this make computation of items faster?

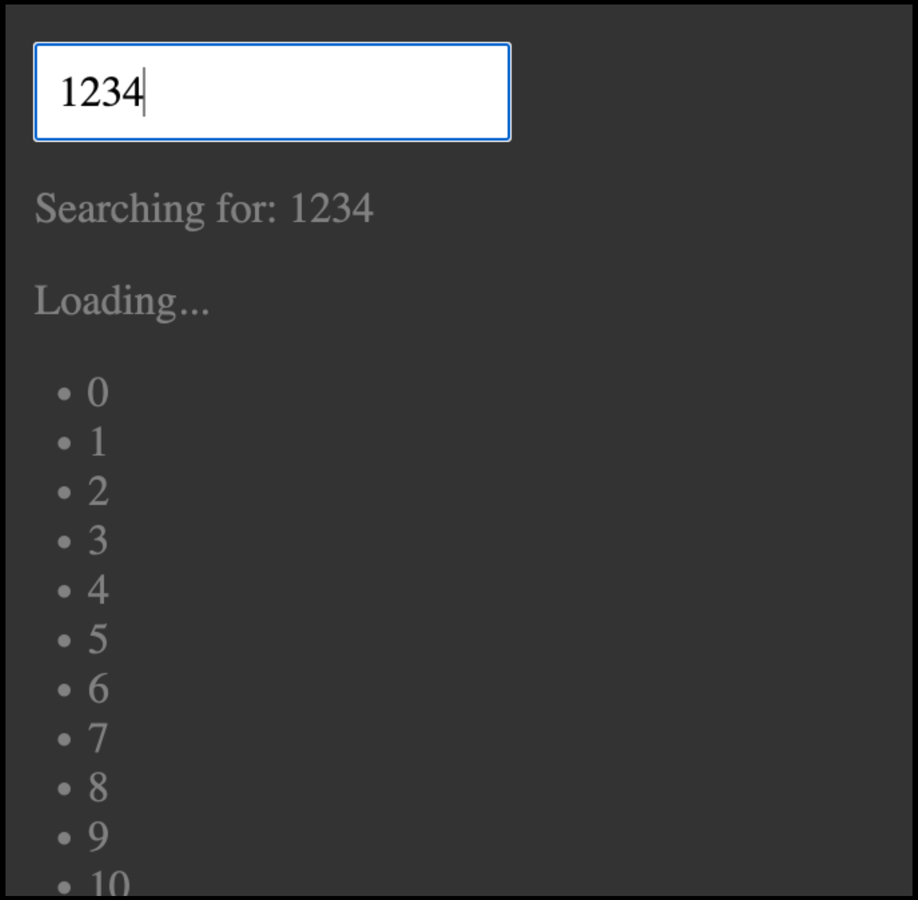
Increase Efficiency

+ should indicate if data is stale



Searching for: All

Loading...



1234

Searching for: 1234

Loading...

- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

```
useEffect(() => {
  startTransition(() => {
    console.log(deferredInput);
    const filtered = bigArray.filter((item) =>
      item.toString().includes(deferredInput)
    );
    setList(filtered);
  });
}, [deferredInput]);
```

```
       <section style=
 {isPending ? { opacity: 0.4 } : null}>
```

useDeferredValue

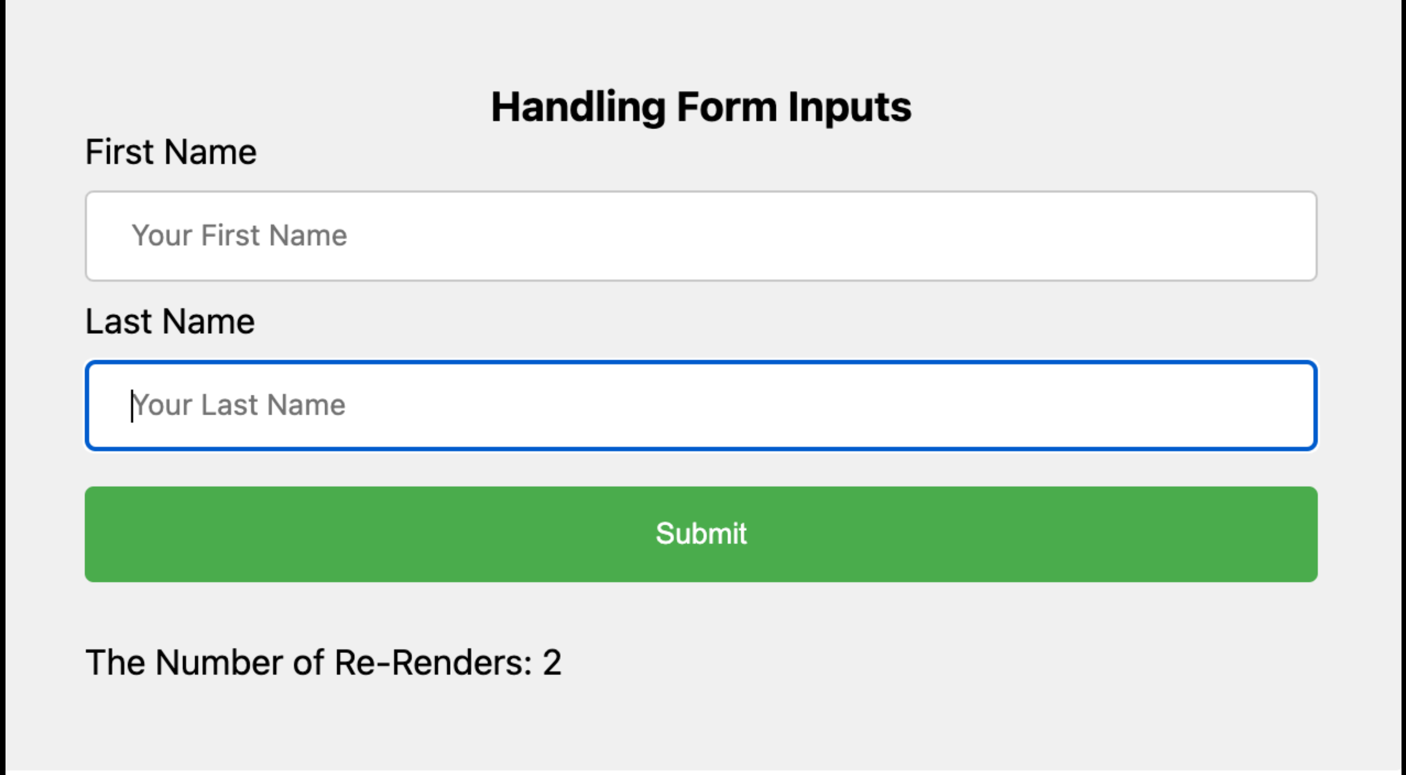
Usage

- Show **stale** content when fresh content is loading
- Indicate that content is **stale**
- Defer re-rendering part of UI (with `memo`)

```
const SlowList = memo(function SlowList({ text }) {
  // ...
});

function App() {
  const [text, setText] = useState('');
  const deferredText = useDeferredValue(text);
  return (
    <>
      <input value={text} onChange={e => setText(e.target.value)} />
      <SlowList text={deferredText} />
    </>
  );
}
```

useRef

Reference a value that's **not** needed for rendering



Handling Form Inputs

First Name

Your First Name

Last Name

Your Last Name

Submit

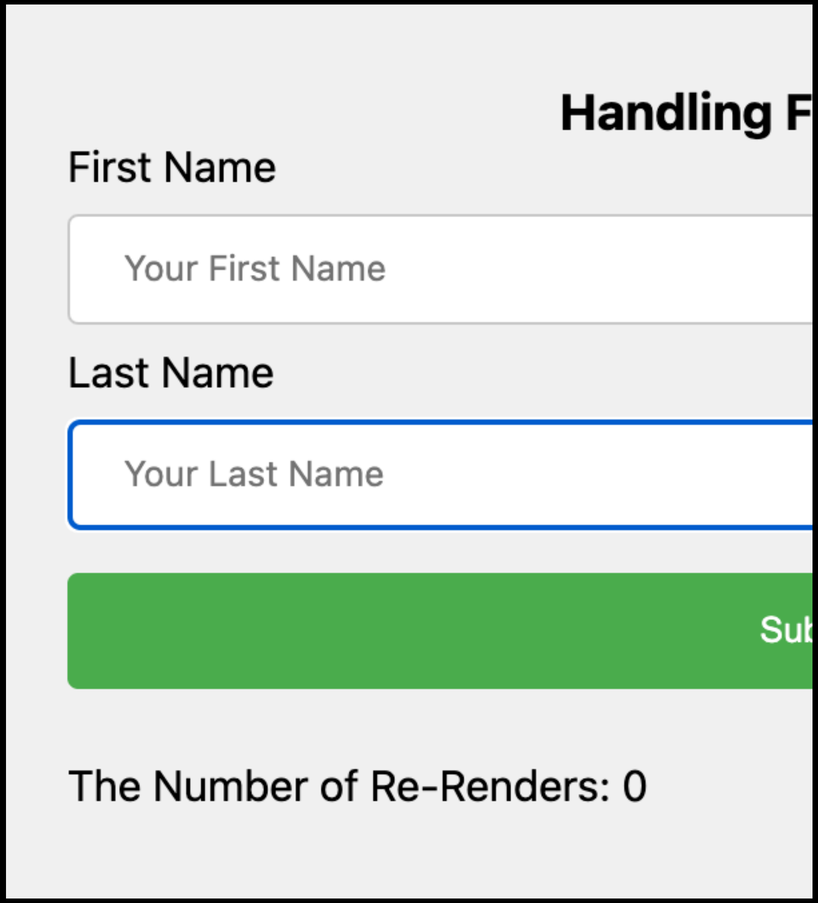
The Number of Re-Renders: 2

useRef

The useRef hook persists values between re-renders

```
<input
  type="text"
  id="lastName"
  placeholder="Your Last Name"
  ref={lastNameInput}
/>
```

```
useEffect(() => {
  // focus the first time on mount
  lastNameInput.current?.focus();
}, []);
```



Handling F

First Name

Your First Name

Last Name

Your Last Name

The Number of Re-Renders: 0

useRef

null check?

```
function FormInputs() {
  const renderCount = useRef(0);
  const lastNameChangeCount = useRef(0);
    // ...
```

✅

```
function VideoPlayer() {
  this.name = "Video Player";
  this.format = ".mp4";
  this.video = "movie";
}

const Video = () => {
  const videoPlayerRef = useRef(new VideoPlayer());
```

❌

useRef

null check?

```
function VideoPlayer() {
  this.name = "Video Player";
  this.format = ".mp4";
  this.video = "movie";
}

const Video = () => {
  const videoPlayerRef = useRef(null);
```

```
const getPlayer = () => {
  if (videoPlayerRef.current !== null) {
    return videoPlayerRef.current;
  }
  const player = new VideoPlayer();
  videoPlayerRef.current = player;
  return player;
};
```

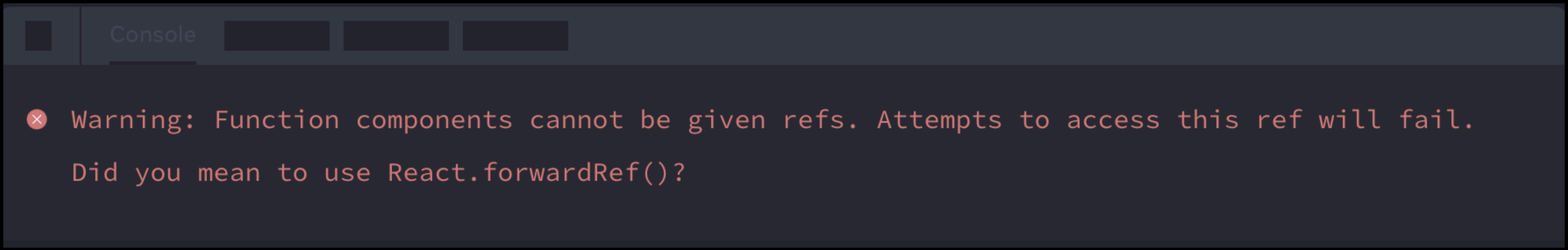
```
  const submit = (e) => {
    e.preventDefault();
    let player = getPlayer();
    // do something with the video player
```

Guaranteed to return a VideoPlayer and never null

useRef

ref to a custom component

```
<div className="App">
  ...
  <OnlinePlayer
    isPlaying={isPlaying}
    setIsPlaying={setIsPlaying}
    handleClick={handleClick}
    ref={onlinePlayerRef}
  />
</div>
```



```
Warning: Function components cannot be given refs. Attempts to access this ref will fail.
Did you mean to use React.forwardRef()?
```

useRef

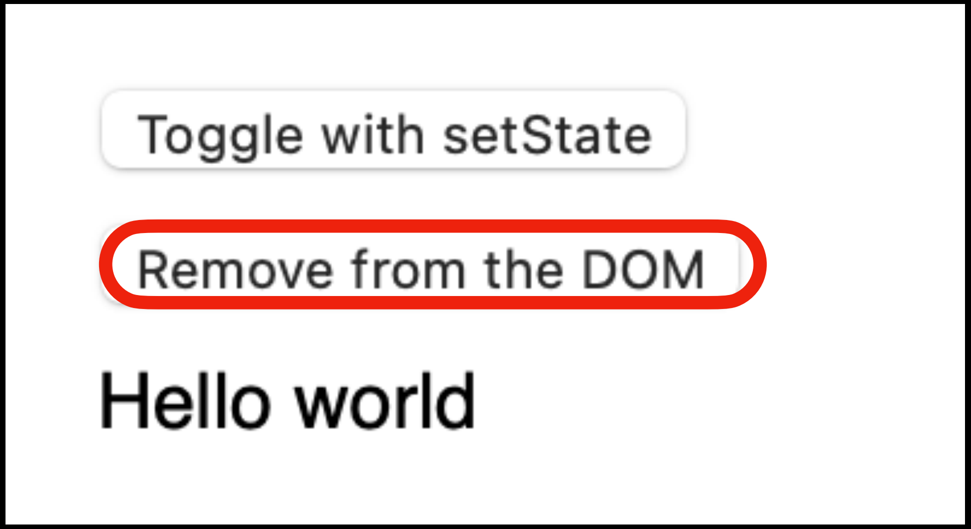
ref to a custom component: `forwardRef`

```
const OnlinePlayer = forwardRef(
  ({ isPlaying, setIsPlaying, handleClick }, ref) => {
    return (
      <>
        <button className="button-player" onClick={handleClick}>
          {isPlaying ? "Pause" : "Play"}
        </button>
        <video
          width="50%"
          ref={ref}
          onPlay={() => setIsPlaying(true)}
          onPause={() => setIsPlaying(false)}
        >
....


}
```

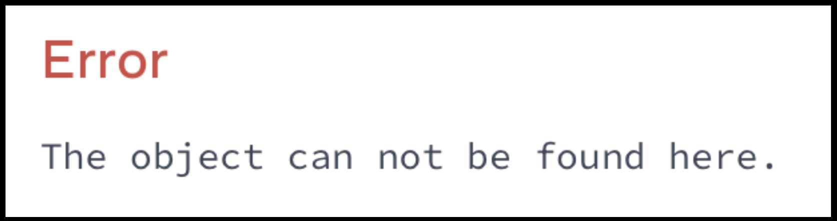
Refs are an escape hatch, use this **sparingly**

Spare usage of refs

```
export default function Counter() {
  const [show, setShow] = useState(true);
  const ref = useRef(null);

  return (
    <div>
      <button
        onClick={() => {
          setShow(!show);
        }}>
        Toggle with setState
      </button>
      <button
        onClick={() => {
          ref.current.remove();
        }}>
        Remove from the DOM
      </button>
      {show && <p ref={ref}>Hello world</p>}
    </div>
  );
}
```



Hello world



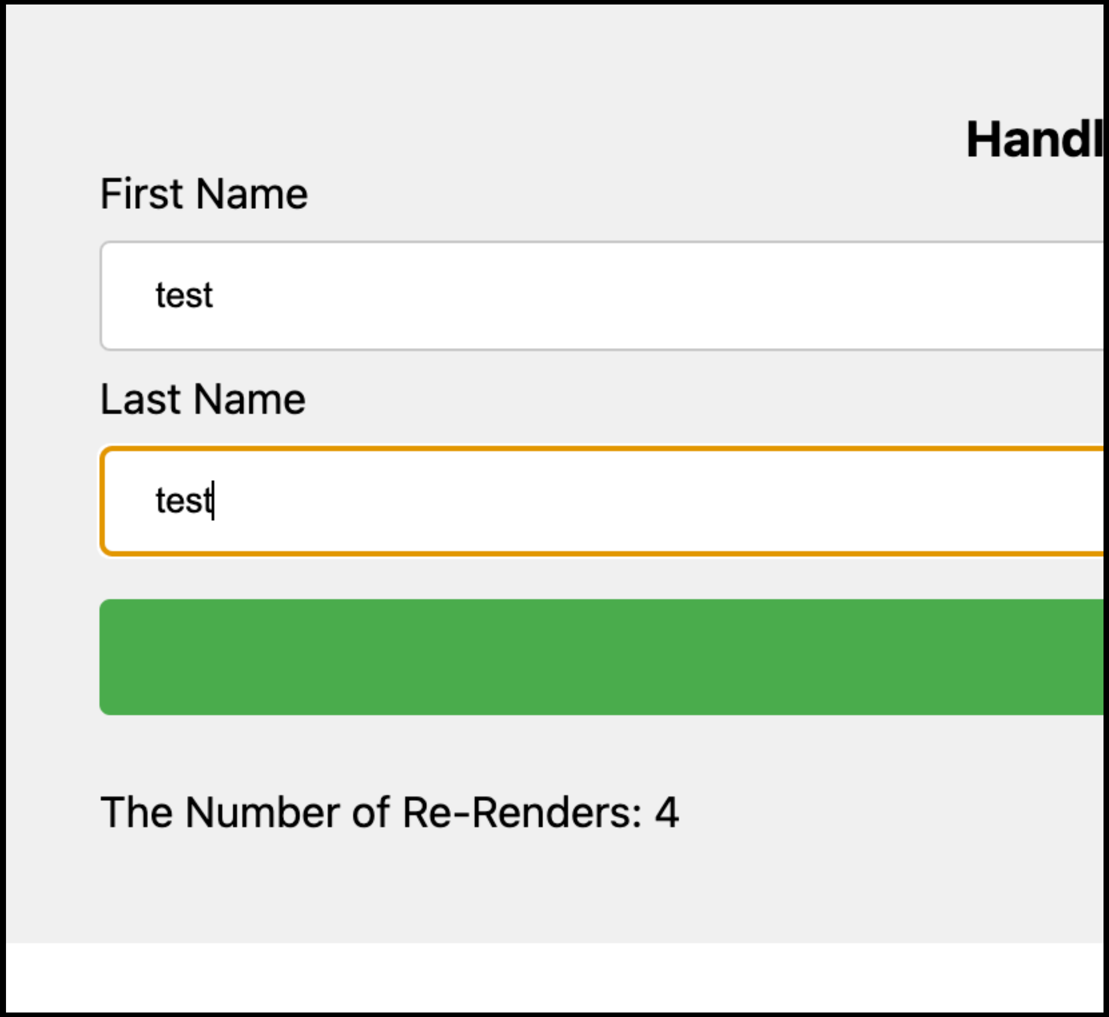
Error

```
The object can not be found here.
```

useRef

Mutating `ref.current` property does not trigger re-render

```
const [firstName, setFirstName] = useState("");
const lastNameInput = useRef();
const lastNameChangeCount = useRef(0);

    <input
      type="text"
      id="firstName"
      placeholder="Your First Name"
      value={firstName}
      onChange={(e) => setFirstName(e.target.value)}
    />

    <input
      type="text"
      id="lastName"
      placeholder="Your Last Name"
      ref={lastNameInput}
      onChange={(e) => {
        console.log("Lastname changed");
        lastNameChangeCount.current += 1;
      }}
    />
```



First Name

test

Last Name

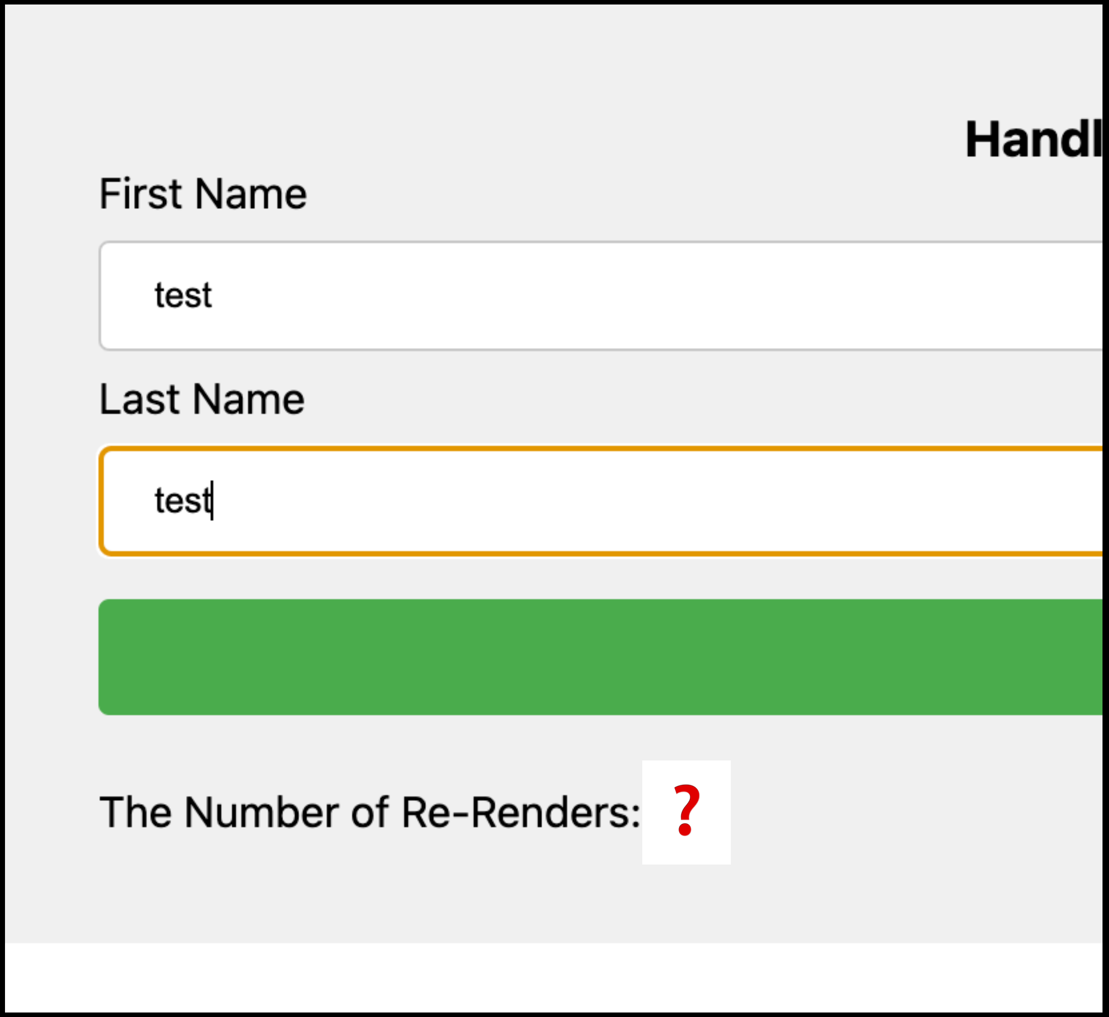
test

The Number of Re-Renders: 4

useRef

Mutating `ref.current` property does not trigger re-render

```
const [firstName, setFirstName] = useState("");
const lastNameInput = useRef();
const lastNameChangeCount = useRef(0);

    <input
      type="text"
      id="firstName"
      placeholder="Your First Name"
      ~~value={firstName}~~
      onChange={(e) => setFirstName(e.target.value)}
    />

    <input
      type="text"
      id="lastName"
      placeholder="Your Last Name"
      ref={lastNameInput}
      onChange={(e) => {
        console.log("Lastname changed");
        lastNameChangeCount.current += 1;
      }}
    />
```



Handl

First Name

test

Last Name

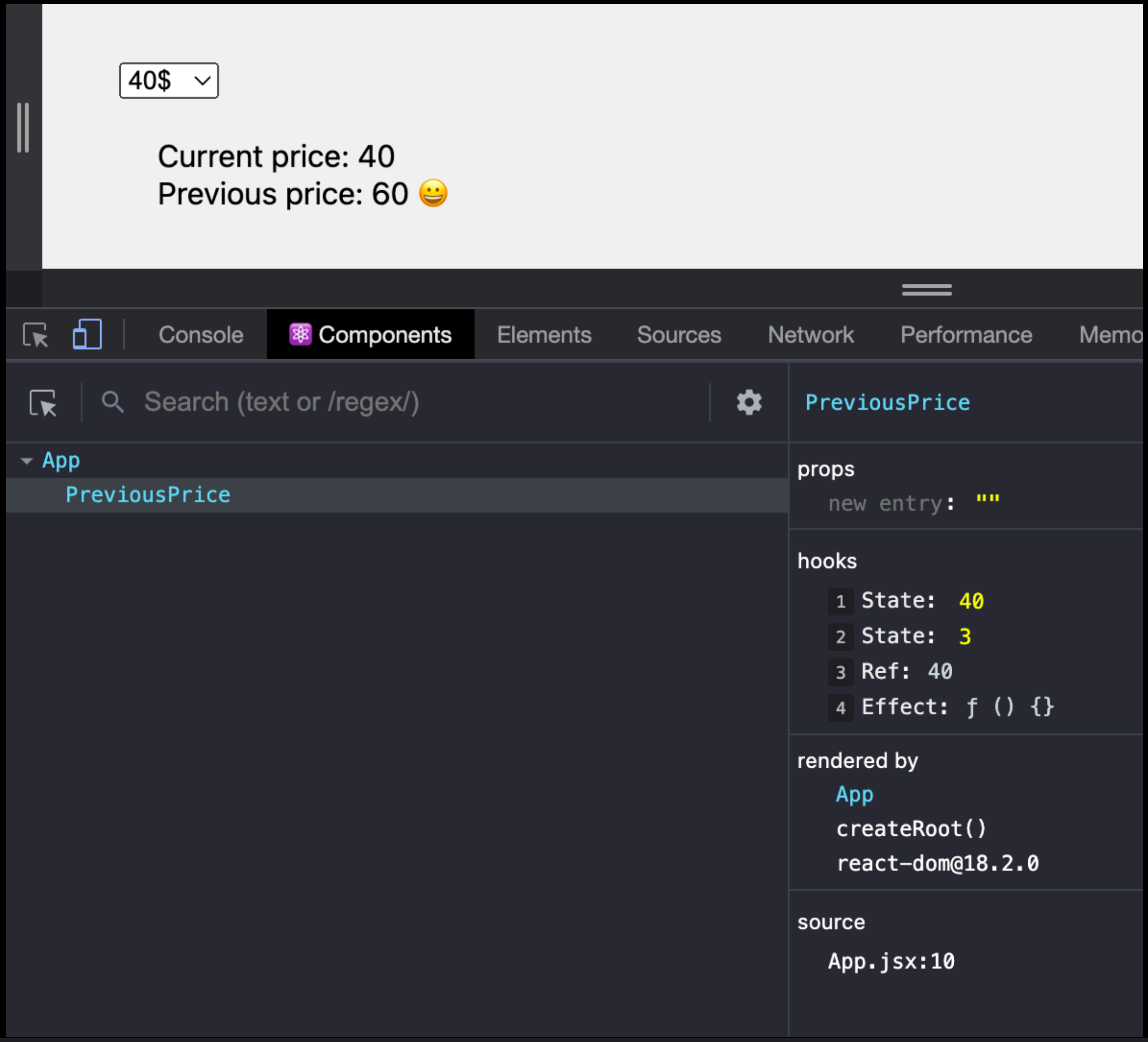
test

The Number of Re-Renders: ?

useRef

Mutating `ref.current` property does not trigger re-render

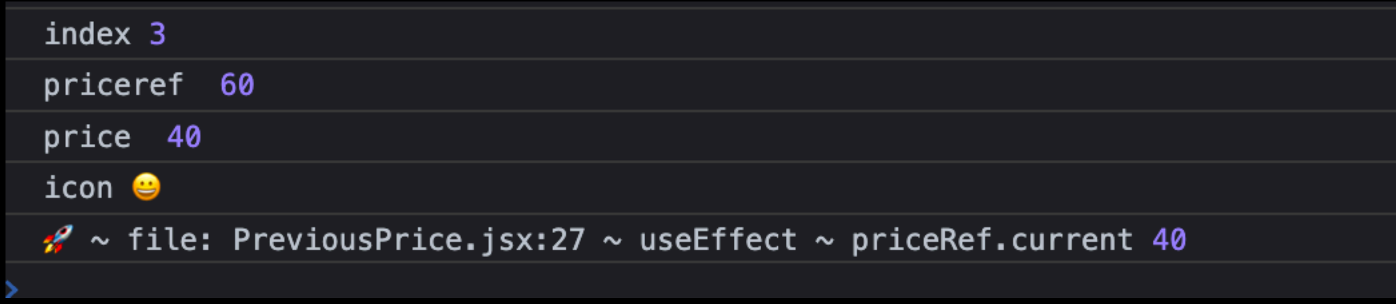
```
  useEffect(() => {
    priceRef.current = price;
  });

  const icon =
    priceRef.current < price ? "😡" :
priceRef.current > price ? "😀" : " 🤓 ";

  return (
    <div>
      <select value={price}
           onChange={onPriceChange}>
        {priceOptions}
      </select>
      <div>
        <p> Current price: {price}</p>
        <p>Previous price: {priceRef.current}</p>
      </div>
    </div>
```



40$

Current price: 40
Previous price: 60 😀

```
PreviousPrice
props
  new entry: ""
hooks
  1 State:  40
  2 State:  3
  3 Ref:  40
  4 Effect: ƒ () {}
rendered by
  App
  createRoot()
  react-dom@18.2.0
source
  App.jsx:10
```



```
index 3
priceref  60
price  40
icon 😀
🚀 ~ file: PreviousPrice.jsx:27 ~ useEffect ~ priceRef.current 40
```

Updating ref

Should not update ref or state inside immediate scope of component's function

- Update reference inside `useEffect()`
- Update reference inside **handlers** (event handlers, timer handlers, etc)

```
function MyComponent({ prop }) {
  const myRef = useRef(0);
  useEffect(() => {
    myRef.current++; // ✅
    setTimeout(() => {
      myRef.current++; // ✅
    }, 1000);
  }, []);
  const handler = () => {
    myRef.current++; // ✅
  };
  myRef.current++; // ❌
  if (prop) {
    myRef.current++; // ❌
  }
  return <button onClick={handler}>
      My button</button>;
}
```

Keeping components **pure**

Don't write or read `ref.current` during rendering

React expects that the body of your component behaves like a **pure** function:

- If the inputs (`props`, `state`, and `context`) are the same, it should return **exactly the same** JSX.
- Calling it in a **different** order or with **different** arguments should **not** affect the results of other calls.

If you have to read or write something during rendering, use `state` instead.