



Complex state



# Agenda

- Repetition - Event Handling
- Kahoot
- Repetition - Övning
- Passera state till barn-komponenter
- Komplex state
- Övning
- Hantera arrays
- Async
- Övning



# Repetition - Event Handling

Vi hanterar state med useState funktion

Testa...

- counter: state
- setCounter: mekanism för att modifiera

```
import { useState } from 'react'

const App = () => {
  const [ counter, setCounter ] = useState(0)

  setTimeout(
    () => setCounter(counter + 1),
    1000
  )

  return (
    <div>{counter}</div>
  )
}

export default App
```



# Repetition - Event Handling

Vi kan låta “user event” trigga förändring

```
const App = () => {  
  const [ counter, setCounter ] = useState(0)  
  
  const handleClick = () => {  
    console.log('clicked')  
  }  
  
  return (  
    <div>  
      <div>{counter}</div>  
      <button onClick={handleClick}>  
        plus  
      </button>  
    </div>  
  )  
}
```



## Repetition - Event Handling

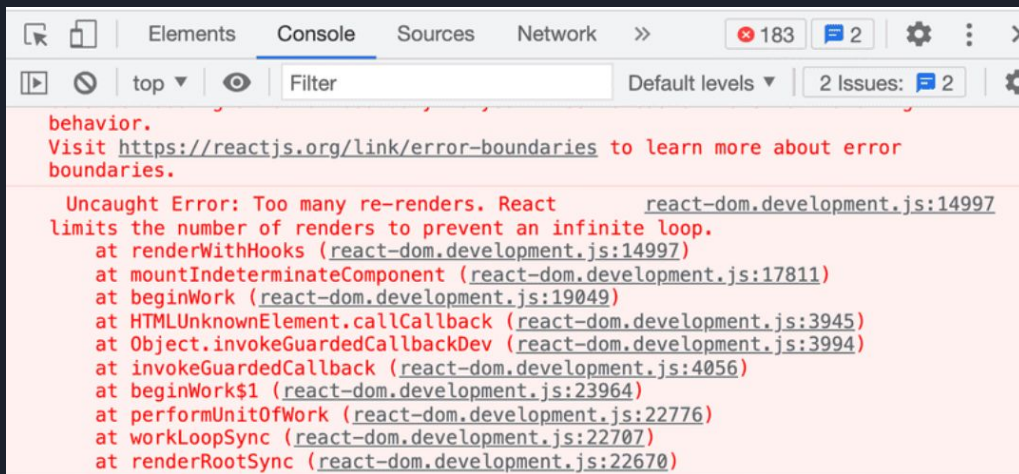
Vad skulle hända om vi skrev så här istället?

```
<button onClick={setCounter(counter + 1)}>  
  plus  
</button>
```

# Repetition - Event Handling

En event handler ska vara en funktion inte funktionsanrop!

- Varje gång appen renderas om så anropas eventHandlern och triggas åter en ny omrendering i infinitum..



The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays a red error message: 'Uncaught Error: Too many re-renders. React limits the number of renders to prevent an infinite loop.' The error is followed by a stack trace listing various React internal functions and their corresponding file paths, such as 'renderWithHooks (react-dom.development.js:14997)', 'mountIndeterminateComponent (react-dom.development.js:17811)', and 'renderRootSync (react-dom.development.js:22670)'. Above the error message, there is a link to 'https://reactjs.org/link/error-boundaries' with the text 'Visit' and 'to learn more about error boundaries.' The console also shows a 'behavior.' label and a '2 Issues: 2' indicator in the top right corner.

```
behavior.  
Visit https://reactjs.org/link/error-boundaries to learn more about error  
boundaries.  
  
Uncaught Error: Too many re-renders. React react-dom.development.js:14997  
limits the number of renders to prevent an infinite loop.  
    at renderWithHooks (react-dom.development.js:14997)  
    at mountIndeterminateComponent (react-dom.development.js:17811)  
    at beginWork (react-dom.development.js:19049)  
    at HTMLUnknownElement.callCallback (react-dom.development.js:3945)  
    at Object.invokeGuardedCallbackDev (react-dom.development.js:3994)  
    at invokeGuardedCallback (react-dom.development.js:4056)  
    at beginWork$1 (react-dom.development.js:23964)  
    at performUnitOfWork (react-dom.development.js:22776)  
    at workLoopSync (react-dom.development.js:22707)  
    at renderRootSync (react-dom.development.js:22670)
```

# Repetition - Event Handling

onChange event

```
1  import React, { useState } from "react";
2  import ReactDOM from "react-dom/client";
3
4  const App = () => {
5    const [text, setText] = useState("");
6    return (
7      <form>
8        <input
9          type="text"
10         value={text}
11         onChange={(e) => setText(e.target.value)}
12        />
13        <p>{text}</p>
14      </form>
15    );
16  };
17
18  ReactDOM.createRoot(document.getElementById("root")).render(<App />);
```



# Repetition - Event Handling

## onChange event (checkbox)

```
1  import React, { useState } from "react";
2  import ReactDOM from "react-dom/client";
3
4  const App = () => {
5    const [isChecked, setIsChecked] = useState(false);
6    const changeCheckBox = () => {
7      setIsChecked(!isChecked);
8    };
9    return (
10     <form>
11       <input checked={isChecked} type="checkbox" onChange={changeCheckBox} />
12       <p>{isChecked ? "checked" : "not"} </p>
13     </form>
14   );
15 };
16
17 ReactDOM.createRoot(document.getElementById("root")).render(<App />);
```



# Repetition - Event Handling

onSubmit event (formulär)

```
const App = () => {
  const [name, setName] = useState("");
  const changeName = (event) => {
    setName(event.target.value);
  };
  const sayHello = (event) => {
    event.preventDefault();
    alert("Hello, " + name);
  };
  return (
    <form onSubmit={sayHello}>
      <input type="text" value={name} onChange={changeName} />
      <input type="submit" value="save"></input>
    </form>
  );
};
```

Kahoot time!!!





# Repetition - Övning

Övning: Skapa ett formulär

- En text input (onChange)
- En spara knapp (som triggar formulär onSubmit)
- När man skrivit något och tryckt “spara” läggs text i lista
- Visa listan i div



# Passera state till barn-komponenter

Vi har följande app...

```
const App = () => {  
  const [ counter, setCounter ] = useState(0)  
  
  const increaseByOne = () => setCounter(counter + 1)  
  
  const setToZero = () => setCounter(0)  
  
  return (  
    <div>  
      <div>{counter}</div>  
      <button onClick={increaseByOne}>  
        plus  
      </button>  
      <button onClick={setToZero}>  
        zero  
      </button>  
    </div>  
  )  
}
```



# Passera state till barn-komponenter

Det är rekommenderat i react att applikationer använder komponenter som är

- lättviktiga (små och enkla)
- återanvändbara

Så vi återgår till komponenten och undersöker hur vi kan refaktorera appen!



# Passera state till barn-komponenter

Vi har i return

- 1 div som visar counter
- 2 knappar som ökar/minskar

```
const App = () => {  
  const [ counter, setCounter ] = useState(0)  
  
  const increaseByOne = () => setCounter(counter + 1)  
  
  const setToZero = () => setCounter(0)  
  
  return (  
    <div>  
      <div>{counter}</div>  
      <button onClick={increaseByOne}>  
        plus  
      </button>  
      <button onClick={setToZero}>  
        zero  
      </button>  
    </div>  
  )  
}
```



# Passera state till barn-komponenter

Vi skapar en display komponent

```
const Display = (props) => {  
  return (  
    <div>{props.counter}</div>  
  )  
}
```




# Passera state till barn-komponenter


Vi skapar en knapp komponent

```
const Button = (props) => {  
  return (  
    <button onClick={props.onClick}>  
      {props.text}  
    </button>  
  )  
}
```



- 
- Vi håller kontroll över state i Förälder komponent (App)
  - Barn komponenter
    - Display (visar state)
    - Button (ändrar state)

```
const App = () => {  
  const [ counter, setCounter ] = useState(0)  
  
  const increaseByOne = () => setCounter(counter + 1)  
  const decreaseByOne = () => setCounter(counter - 1)  
  const setToZero = () => setCounter(0)  
  
  return (  
    <div>  
      <Display counter={counter}/>  
      <Button  
        onClick={increaseByOne}  
        text='plus'  
      />  
      <Button  
        onClick={setToZero}  
        text='zero'  
      />  
      <Button  
        onClick={decreaseByOne}  
        text='minus'  
      />  
    </div>  
  )  
}
```

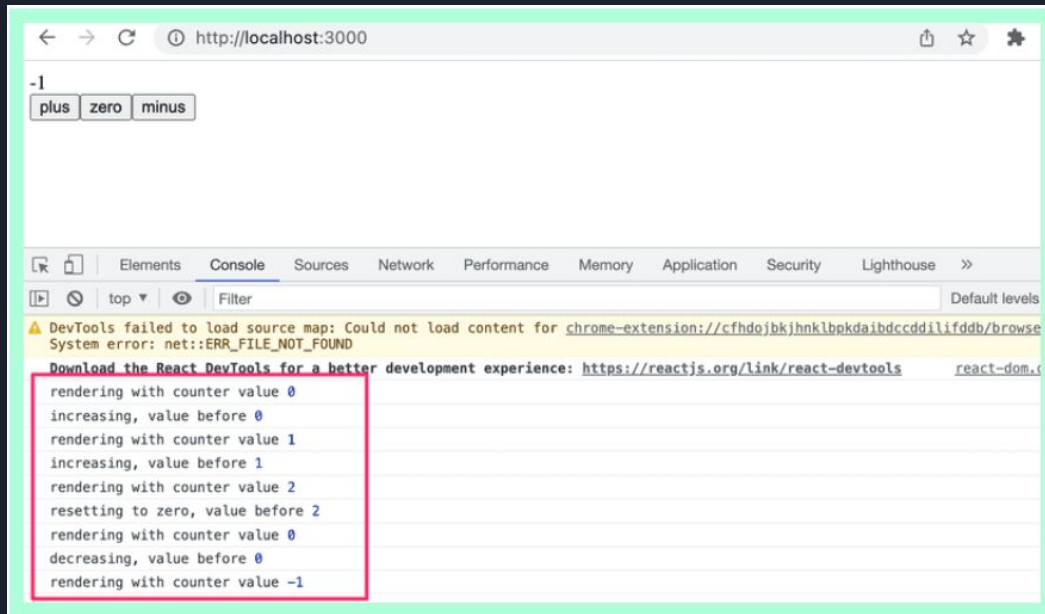


Om vi lägger till lite `console.log` kan vi undersöka vad som triggat omrendering av komponenter....

```
const App = () => {  
  const [counter, setCounter] = useState(0)  
  console.log('rendering with counter value', counter)  
  
  const increaseByOne = () => {  
    console.log('increasing, value before', counter)  
    setCounter(counter + 1)  
  }  
  
  const decreaseByOne = () => {  
    console.log('decreasing, value before', counter)  
    setCounter(counter - 1)  
  }  
  
  const setToZero = () => {  
    console.log('resetting to zero, value before', counter)  
    setCounter(0)  
  }  
  
  return (  
    <div>  
      <Display counter={counter} />  
      <Button onClick={increaseByOne} text="plus" />  
      <Button onClick={setToZero} text="zero" />  
      <Button onClick={decreaseByOne} text="minus" />  
    </div>  
  )  
}
```

# Passera state till barn-komponenter

Vi ser att varje ändring av state  
renderar om App



# Passera state till barn-komponenter

Vi kan också givetvis refaktorera app genom destruktering

```
const Display = (props) => {  
  return (  
    <div>{props.counter}</div>  
  )  
}
```



```
const Display = ({ counter }) => {  
  return (  
    <div>{counter}</div>  
  )  
}
```

# Passera state till barn-komponenter

Vi kan också givetvis refaktorera app genom destruktering

```
const Button = (props) => {  
  return (  
    <button onClick={props.onClick}>  
      {props.text}  
    </button>  
  )  
}
```



```
const Button = ({ onClick, text }) => (  
  <button onClick={onClick}>  
    {text}  
  </button>  
)
```



# Övning

Refaktorera ert formulär

- En text input => Gör till komponent (Input)
- En spara knapp => Gör till komponent (Button)
- Listan i div => Gör till komponent (List)



# Komplex state

Vi har hittills bara tittat på enkel state (ett tal eller lista)

- Om vi har ett mer komplext state?

Många gånger så kan man dela upp state i flera useState deklARATIONER....



# Komplex state

En app med två state (left, right)

```
const App = () => {  
  const [left, setLeft] = useState(0)  
  const [right, setRight] = useState(0)  
  
  return (  
    <div>  
      {left}  
      <button onClick={() => setLeft(left + 1)}>  
        left  
      </button>  
      <button onClick={() => setRight(right + 1)}>  
        right  
      </button>  
      {right}  
    </div>  
  )  
}
```





# Komplex state

Men om vi skulle baka in 2 state i 1!

```
{  
  left: 0,  
  right: 0  
}
```



# Komplex state

```
const App = () => {  
  const [clicks, setClicks] = useState({  
    left: 0, right: 0  
  })  
  
  const handleLeftClick = () => {  
    const newClicks = {  
      left: clicks.left + 1,  
      right: clicks.right  
    }  
    setClicks(newClicks)  
  }  
  
  const handleRightClick = () => {  
    const newClicks = {  
      left: clicks.left,  
      right: clicks.right + 1  
    }  
    setClicks(newClicks)  
  }  
  
  return (  
    <div>  
      {clicks.left}  
      <button onClick={handleLeftClick}>left</button>  
      <button onClick={handleRightClick}>right</button>  
      {clicks.right}  
    </div>  
  )  
}
```



# Komplex state

Nu behöver våra event handlers uppdatera 2 värden

```
const handleLeftClick = () => {  
  const newClicks = {  
    left: clicks.left + 1,  
    right: clicks.right  
  }  
  setClicks(newClicks)  
}
```



# Komplex state

Det är bara 1 värde varje event handler ska uppdatera  
så kan vi använda "object spread"

```
const handleLeftClick = () => {  
  const newClicks = {  
    ...clicks,  
    left: clicks.left + 1  
  }  
  setClicks(newClicks)  
}
```

```
const handleRightClick = () => {  
  const newClicks = {  
    ...clicks,  
    right: clicks.right + 1  
  }  
  setClicks(newClicks)  
}
```



# Komplex state

Vi kan därmed refaktorera lite...

```
const handleLeftClick = () =>
  setClicks({ ...clicks, left: clicks.left + 1 })

const handleRightClick = () =>
  setClicks({ ...clicks, right: clicks.right + 1 })
```



# Komplex state

Någon kanske undrar varför vi inte bara uppdaterar på följande sätt

```
const handleClick = () => {  
  clicks.left++  
  setClicks(clicks)  
}
```

Men vi får inte uppdatera state på det sättet, utan måste alltid ge nytt state (nytt objekt)

- OBS: Vi måste kopiera värden....



# Övning

Refaktorera ert formulär

- Lägg till count i state så att state  
    `useState({ list: [], count: 0 })`
- Låt `onSubmit` uppdatera både list och count
- Visa i List komponenten både list och count



# Hantera arrays

Vi lägger till ett state till vår app

```
const App = () => {  
  const [left, setLeft] = useState(0)  
  const [right, setRight] = useState(0)  
  const [allClicks, setAll] = useState([])  
  
  const handleLeftClick = () => {  
    setAll(allClicks.concat('L'))  
    setLeft(left + 1)  
  }  
  
  const handleRightClick = () => {  
    setAll(allClicks.concat('R'))  
    setRight(right + 1)  
  }  
  
  return (  
    <div>  
      {left}  
      <button onClick={handleLeftClick}>left</button>  
      <button onClick={handleRightClick}>right</button>  
      {right}  
      <p>{allClicks.join(' ')}</p>  
    </div>  
  )  
}
```





# Hantera arrays

Varför gör vi inte på följande sätt?

```
const handleLeftClick = () => {  
  allClicks.push('L')  
  setAll(allClicks)  
  setLeft(left + 1)  
}
```

Diskutera....



# Async

Vi lägger till ytterligare state

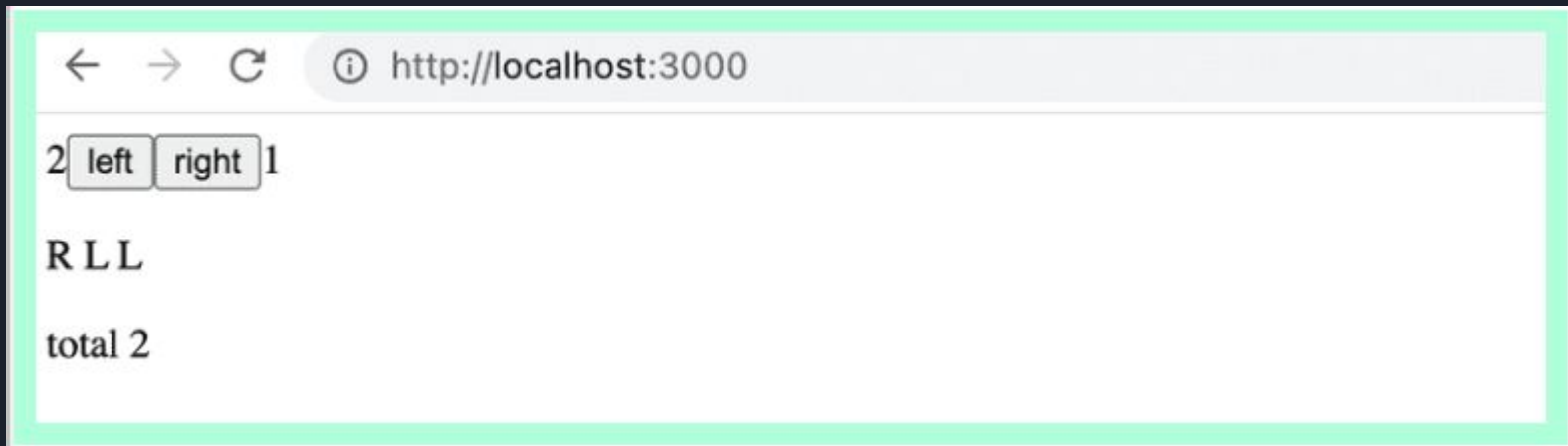
Men vi får problem...

```
const App = () => {  
  const [left, setLeft] = useState(0)  
  const [right, setRight] = useState(0)  
  const [allClicks, setAll] = useState([])  
  const [total, setTotal] = useState(0)  
  
  const handleLeftClick = () => {  
    setAll(allClicks.concat('L'))  
    setLeft(left + 1)  
    setTotal(left + right)  
  }  
  
  const handleRightClick = () => {  
    setAll(allClicks.concat('R'))  
    setRight(right + 1)  
    setTotal(left + right)  
  }  
  
  return (  
    <div>  
      {left}  
      <button onClick={handleLeftClick}>left</button>  
      <button onClick={handleRightClick}>right</button>  
      {right}  
      <p>{allClicks.join(' ')}</p>  
      <p>total {total}</p>  
    </div>  
  )  
}
```



# Async

Vi ser att vi hela tiden ligger efter i count





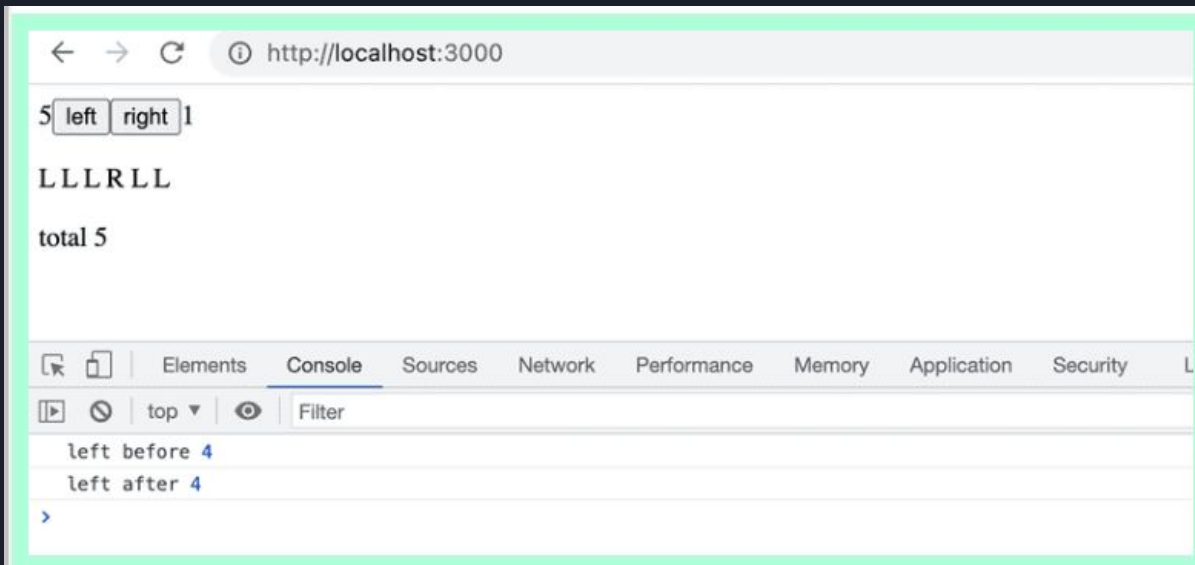
# Async

Vi lägger till några loggar för att kolla vad problemet är

```
const App = () => {  
  // ...  
  const handleClick = () => {  
    setAll(allClicks.concat('L'))  
    console.log('left before', left)  
    setLeft(left + 1)  
    console.log('left after', left)  
    setTotal(left + right)  
  }  
  
  // ...  
}
```

# Async

Vi ser i loggen nu att left är både 4 och 4 efter att setLeft har anropats!





# Async

Vi fixar det genom att lägga till en variabel som både setLeft och setTotal använder

```
const App = () => {  
  // ...  
  const handleClick = () => {  
    setAll(allClicks.concat('L'))  
    const updatedLeft = left + 1  
    setLeft(updatedLeft)  
    setTotal(updatedLeft + right)  
  }  
  
  // ...  
}
```



# Övning

Refaktorera ert formulär

- Ändra ert state från 1 till 2

`useState({ list: [], count: 0 }) => useState([]) och useState(0)`