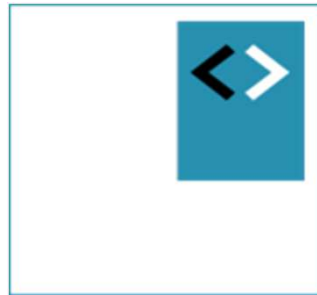


React Fundamentals

Module – function based components



Peter Kassenaar –
info@kassenaar.com



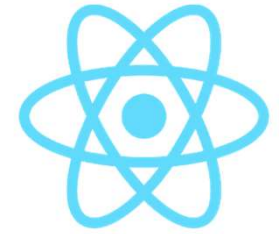
Components

Adding and using your own components

Types of components

Function Components

Class Components



First: import Bootstrap

- Import Bootstrap **just** to make it look nice
- Import as **generic** style library
- There *are* React-specific libraries
 - But we're not using them here (yet)
- <https://getbootstrap.com/>
- `npm install bootstrap`


Bootstrap

Build responsive, mobile-first projects on the web with the world's most popular front-end component library.

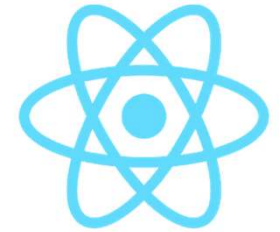
Bootstrap is an open source toolkit for developing with HTML, CSS, and JS. Quickly prototype your ideas or build your entire app with our Sass variables and mixins, responsive grid system, extensive prebuilt components, and powerful plugins built on jQuery.

[Get started](#)[Download](#)

Currently v4.3.1

The Bootstrap logo, a purple diamond shape with a white 'B' inside, and three more purple diamonds stacked behind it to create a 3D effect.

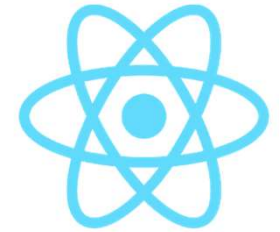
Use Bootstrap in application



Update `index.js`

```
// bootstrap stuff
import 'bootstrap/dist/css/bootstrap.min.css'

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

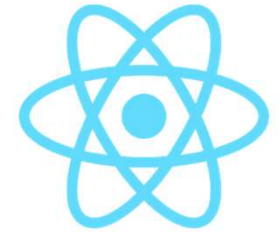


Function-based components

- Components are just functions
- They are rendered to the screen by the `ReactDOM.render()` function
 - 1st parameter – the component to render
 - 2nd parameter – the element to render it to

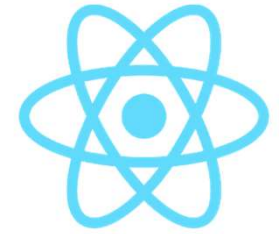
```
function HelloReact() {  
  return <div>Hello React!</div>  
}  
  
ReactDOM.render(  
  <HelloReact/>,  
  document.getElementById('root')  
) ;
```

Other syntax, same result

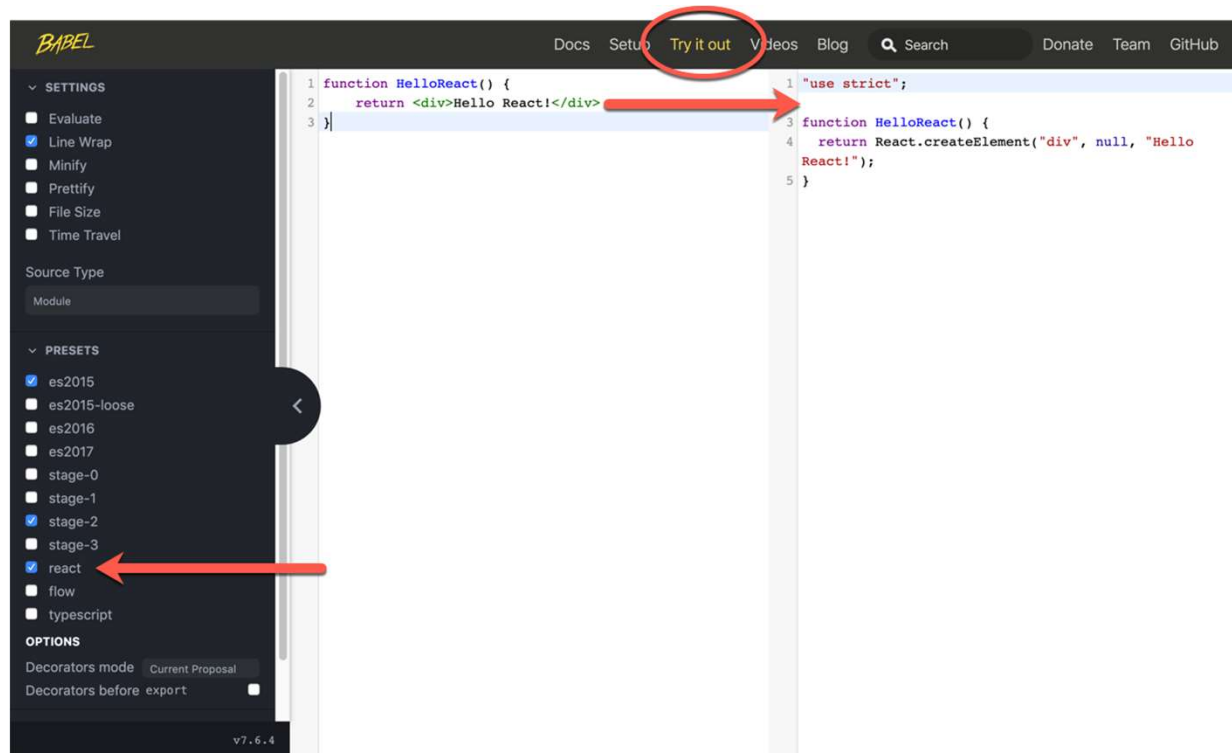


```
• const HelloReact = () => {  
  return (  
    <div>Hello React!</div>  
  )  
}  
  
ReactDOM.render(  
  <HelloReact/>,  
  document.getElementById('root')  
) ;
```

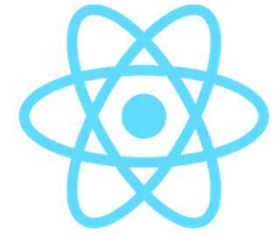
HelloReact Component



- The HelloReact Component is NOT valid JavaScript
- It is JSX, compiled to React API calls by Babel



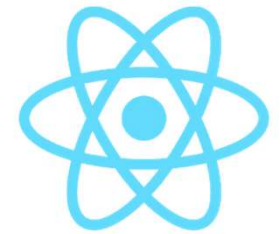
<https://babeljs.io/>,
menu option *Try it out*



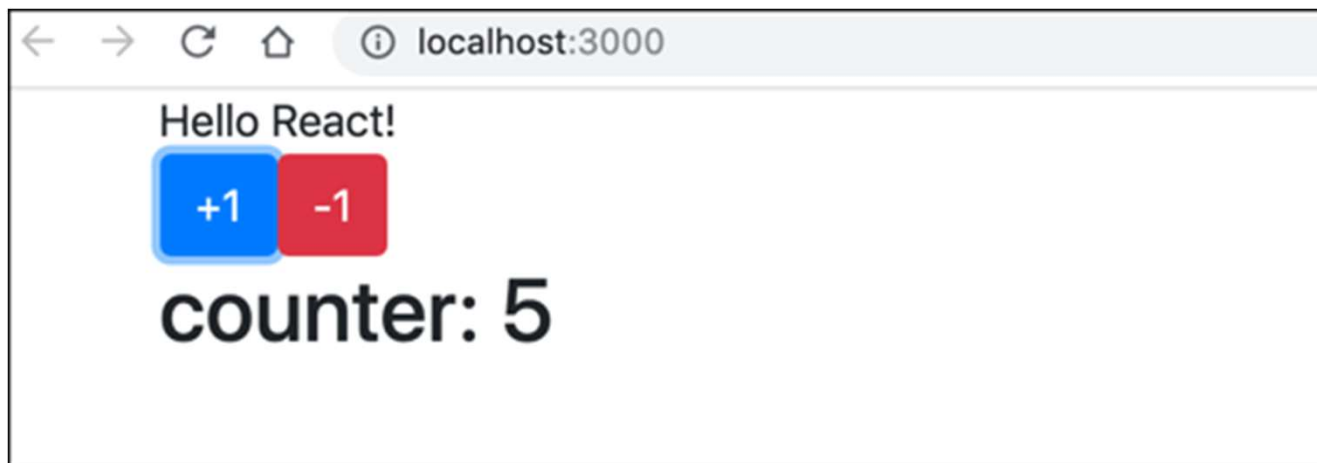
Convention: Component Names

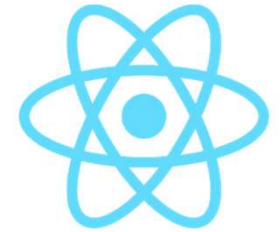
- Start components with Uppercase first letter
 - `function Button() {...}` NOT
 - `function button() {...}`
- Preferred: use two words to distinguish from standard HTML elements
 - `function ReactButton() {...}`
- Opinion: React Style Guide(s)
 - AirBnB JSX Style Guide:
<https://github.com/airbnb/javascript/tree/master/react>
 - More options: <https://css-tricks.com/react-code-style-guide/>

Creating a simple Counter component



Simple result:

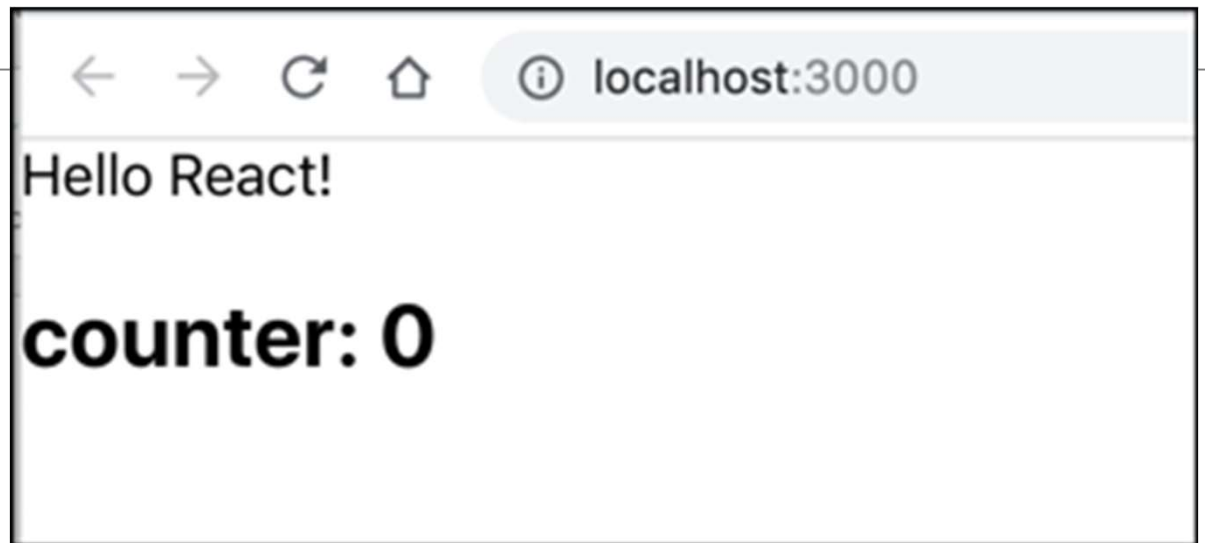




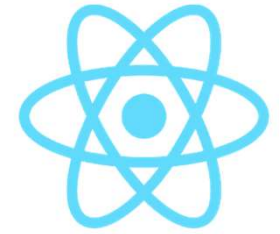
Component functionality - Hooks

- We need *state* to hold the value of counter
 - Using a built in method `React.useState()`
 - This returns two objects – you can call them any way you like
 - First object: *state object (getter)*
 - Second object: *updater function (setter)*
- We use ES6 *destructuring* to assign them to variables
- We initialize the first variable (in our case: `counter`) with a value

```
const [counter, setCounter] = React.useState(0)
return (
  <div>
    <h2>counter: {counter}</h2>
  </div>
)
```

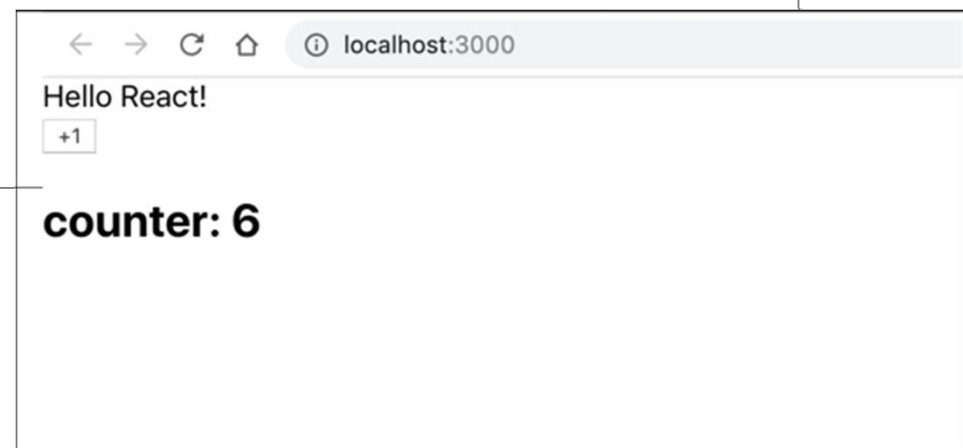



On JSX Syntax



- You can use variables anywhere in JSX, using the *single curly brace* syntax
 - `<h2>{counter}</h2>` to display the current value of `counter`
 - Other frameworks often use double curly braces `{{ ... }}`
- To update the counter we use an *event handler*
 - This is the second argument that `useState()` returns
 - It looks like the DOM API, but it is case sensitive (`onClick`, `onBlur`, `onSubmit`, etc).
 - The event handler receives a function reference

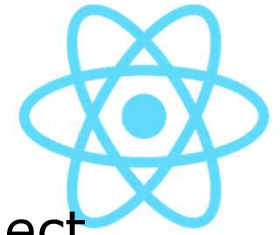
```
function Counter() {  
  
  const [counter, setCounter] = React.useState(0);  
  
  function updateCounter() {  
    return setCounter(counter + 1);  
  }  
  
  return (  
    <div>  
      <button onClick={updateCounter}>+1</button>  
      <h2>counter: {counter}</h2>  
    </div>  
  )  
}
```



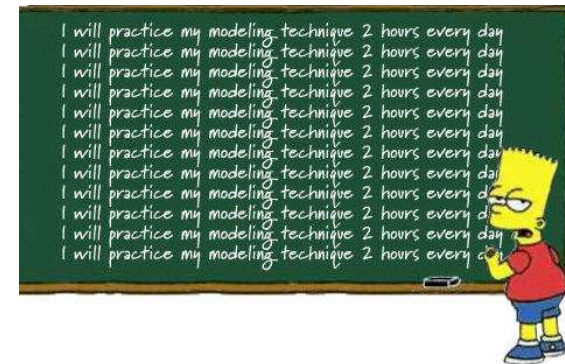
Or, use ES6-notation:

```
const updateCounter = () => setCounter(counter + 1);
```

Workshop



- Continue with your own project, or use the sample project.
- Implement the **subtract** function (-1) for the counter component.
- Create a new button, that **doubles** ($*2$) the current value of the counter.
- Create a **Reset** button, that sets the counter back to 0.
- Example `../110-counter-component`

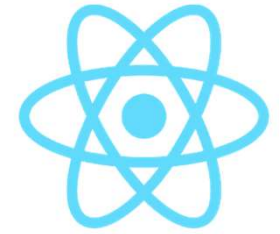




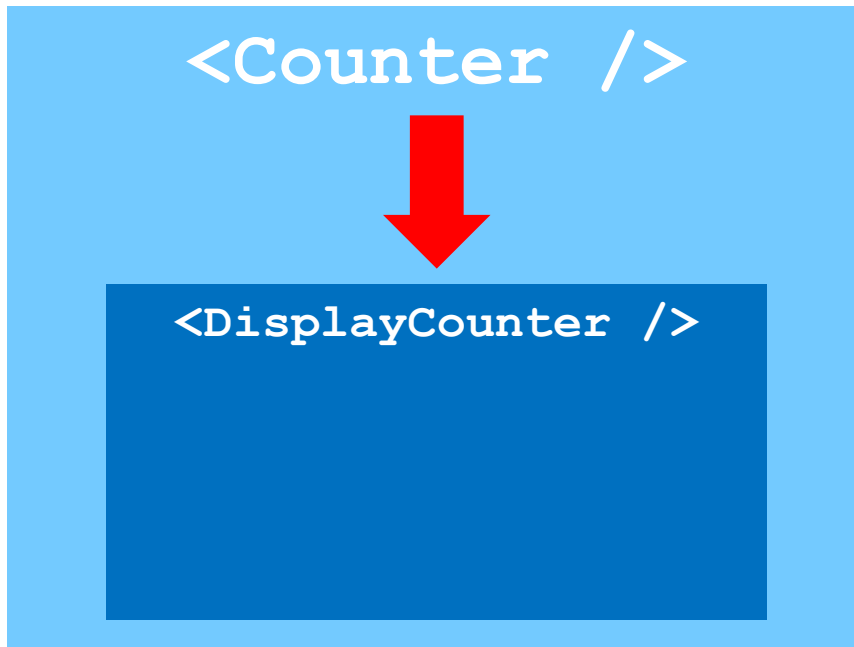
Using Props

Passing data down to other components

Passing state – One-way-dataflow



- **SoC**: We want to create a new component for showing the value of the counter:
 - `<Counter />` **functional component** (it updates the state)
 - `<DisplayCounter />` **display component** (it just displays the counter)
- But: **state** is unique to each component
- We need to pass the counter state to new component
- Introducing the **props** object

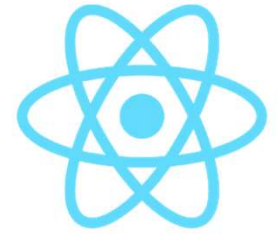


Pass props down. In this case the value of counter

We can pass as many props as we need.


Every prop becomes an *attribute* on the receiving component

Props



- Every component receives a `props` object
 - Again, you can name it anything you want
 - But it is commonly named `props`
- It holds key/value-pairs for every property passed down
- In `Counter.js` we create a prop `counter` and pass it the current value of `counter`:
 - `<DisplayCounter counter={counter}/>`

Receiving props



```
// import React from 'react'

function DisplayCounter(props) {
  return (
    <div>
      {props.counter}
    </div>
  )
}
export default DisplayCounter
```

But what if the components are siblings?

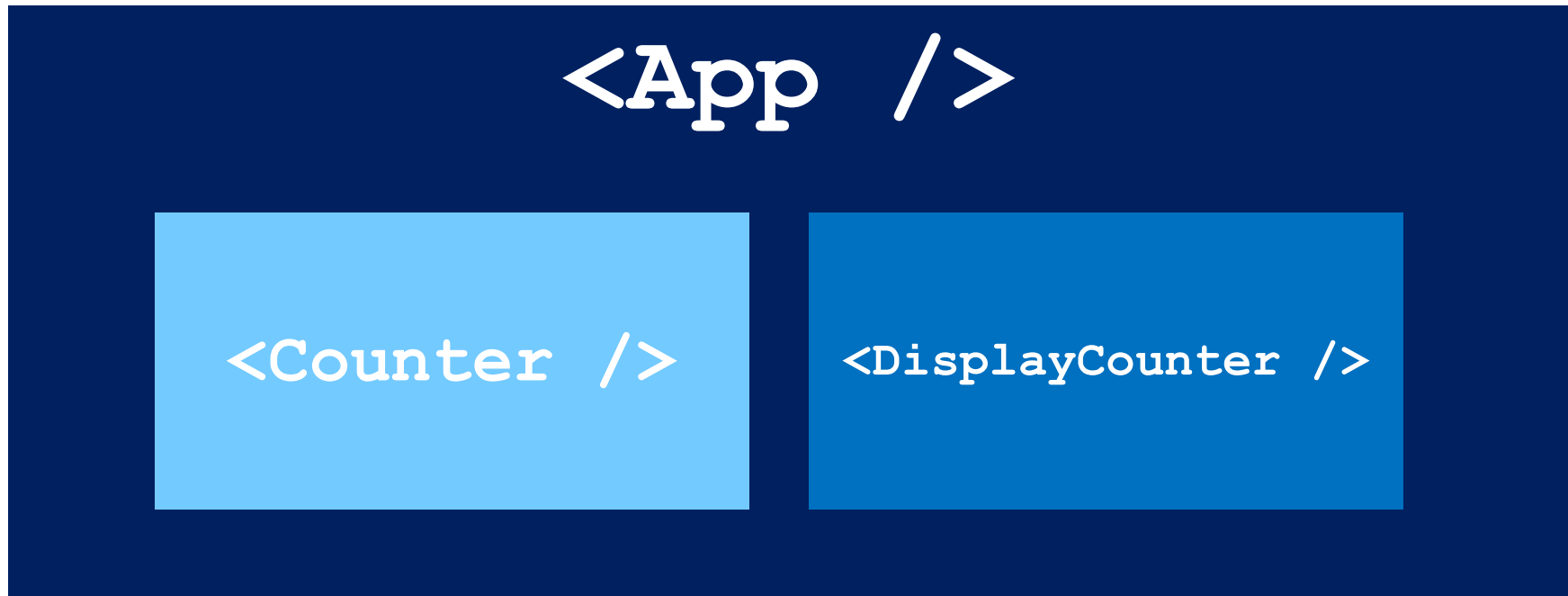
```
<Counter />
```

```
<DisplayCounter />
```

We need an enclosing *parent* component that holds the state.

React: “*We are lifting state up*”

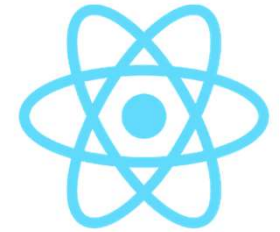
But what if the components are siblings?



Now `<App />` holds the state.

It is passing *functionality* down to `<Counter />`

and *data* down to `<DisplayCounter />`



Next – passing functionality down

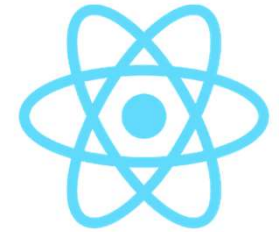
- We can also pass *functions* as props
 - After all: functions are **just JavaScript objects**
- Lift state up from `<Counter />` to `<App />`

```
// Our parent component - it holds the state for the child components
function App() {
  const [counter, setCounter] = React.useState(0);

  const incrementCounter = () => setCounter(counter + 1);

  return (
    ...
  );
}
```

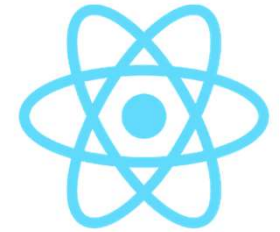
Send props down to child components



```
function App() {  
  const [counter, setCounter] = React.useState(0);  
  
  const incrementCounter = () => setCounter(counter + 1);  
  
  return (  
    <div className="container">  
      <h2>Hello React</h2>  
  
      <Counter increment={incrementCounter}/>  
      <DisplayCounter counter={counter}/>  
    </div>  
  );  
}
```



Update `<Counter />` to receive props

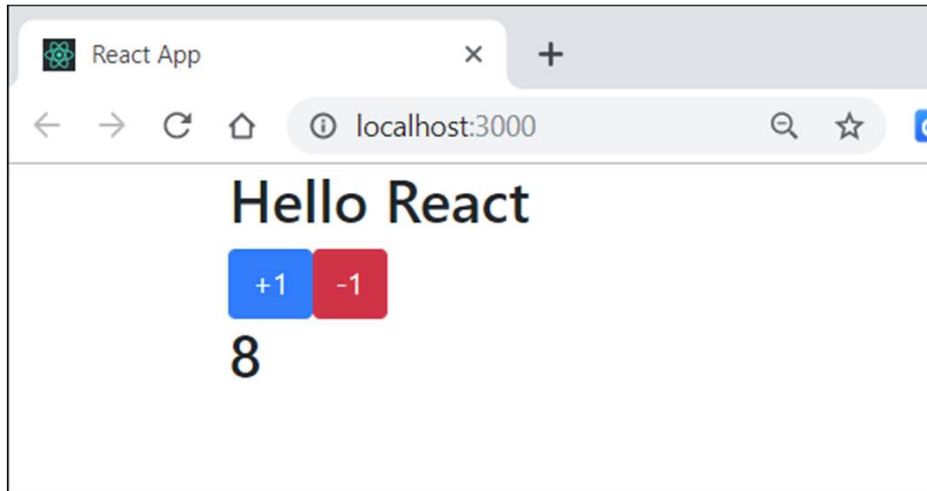


```
function Counter(props) {  
  return (  
    <div>  
      <button className="btn btn-primary"  
        onClick={props.increment}>+1</button>  
    </div>  
  )  
}
```

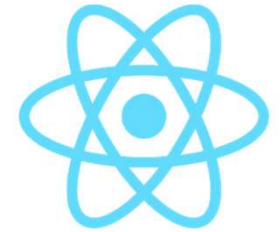
`props.increment` is just a function pointer here.

It points to the function in the parent component

Result




Result is the same visually,
but with a different, more flexible architecture




Alternate notation

- We can also **destructure** the props into their own object
- This is just a **different notation** for the same functionality



```
// counter.js(x)
function Counter({increment}) {
  return (
    <div>
      <button className="btn btn-primary"
        onClick={increment}>+1</button>
      ...
    </div>
  )
}

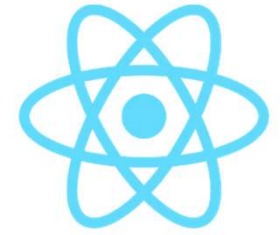
export default Counter
```



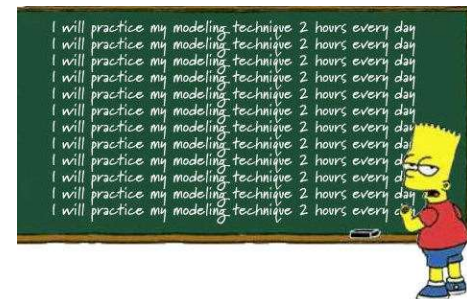
```
// DisplayCounter.js(x)
function DisplayCounter({counter}) {
  return(
    <h2>
      {counter}
    </h2>
  )
}

export default DisplayCounter
```

Workshop



- Lift state up in your own application, passing `props` and `functions` to child components
- OR: start with the sample application, and implement the `decrement` function
- New: pass the `className` for the button as a prop (for instance: `btn-primary`, `btn-success`, `btn-info`,...)
- Update the `<App />` component
- Use both notations for props
- Example `../120-props`

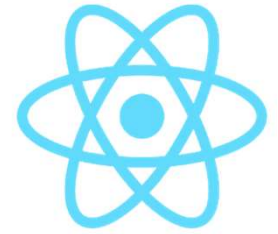




Passing arguments for props

What if your functions (passed down as props) needs parameters?

Component Reusability



- Let's say we want to make the `<Counter />` more generic and reusable
- We want to pass in a `value` to add or subtract from the `counter`
- We need to pass parameters to the function!

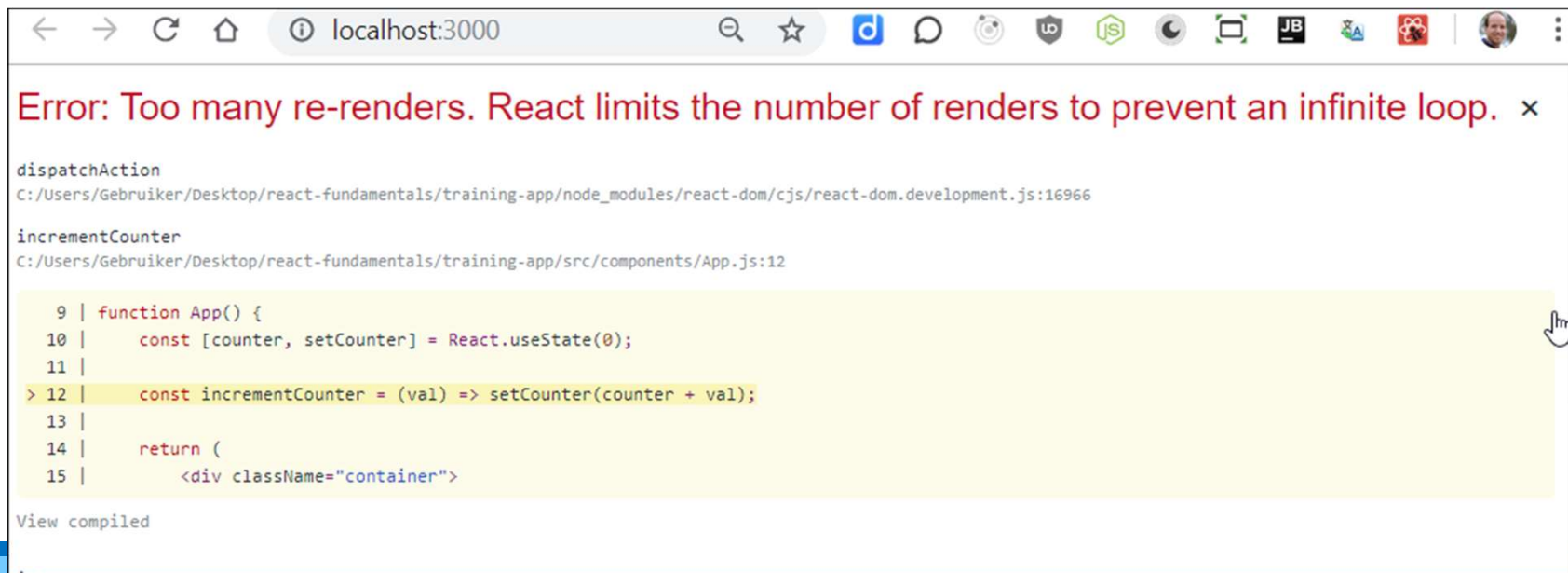
Update the parent component

1. Update the `incrementCounter` function in the parent component (`<App />`)


```
const incrementCounter = (val) => setCounter(counter + val);
```

2. Update the `prop` that is passed. However, this is **invalid!**:

```
<Counter increment={incrementCounter(30)} />
```



This is also invalid

```
function Counter(props) {  
  return (  
    <div>  
      <button className="btn ..."   
        onClick={props.increment(30)}>+30</button>  
    </div>  
  )  
}
```

Error: Maximum update depth exceeded. This can happen when a component repeatedly calls `setState` inside `componentWillUpdate` or `componentDidUpdate`. React limits the number of nested updates to prevent infinite loops. ✕

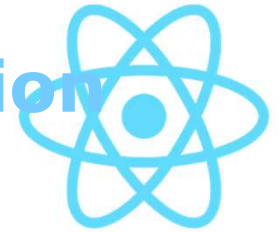
► 3 stack frames were collapsed.

incrementCounter [as increment]
C:/Users/Gebruiker/Desktop/react-fundamentals/training-app/src/components/App.js:12


```
9 | function App() {  
10 |   const [counter, setCounter] = React.useState(0);  
11 |  
> 12 |   const incrementCounter = (val) => setCounter(counter + val);  
13 |  
14 |   return (  
15 |     <div className="container">
```


[View compiled](#)

Solution: create a prop and inline function

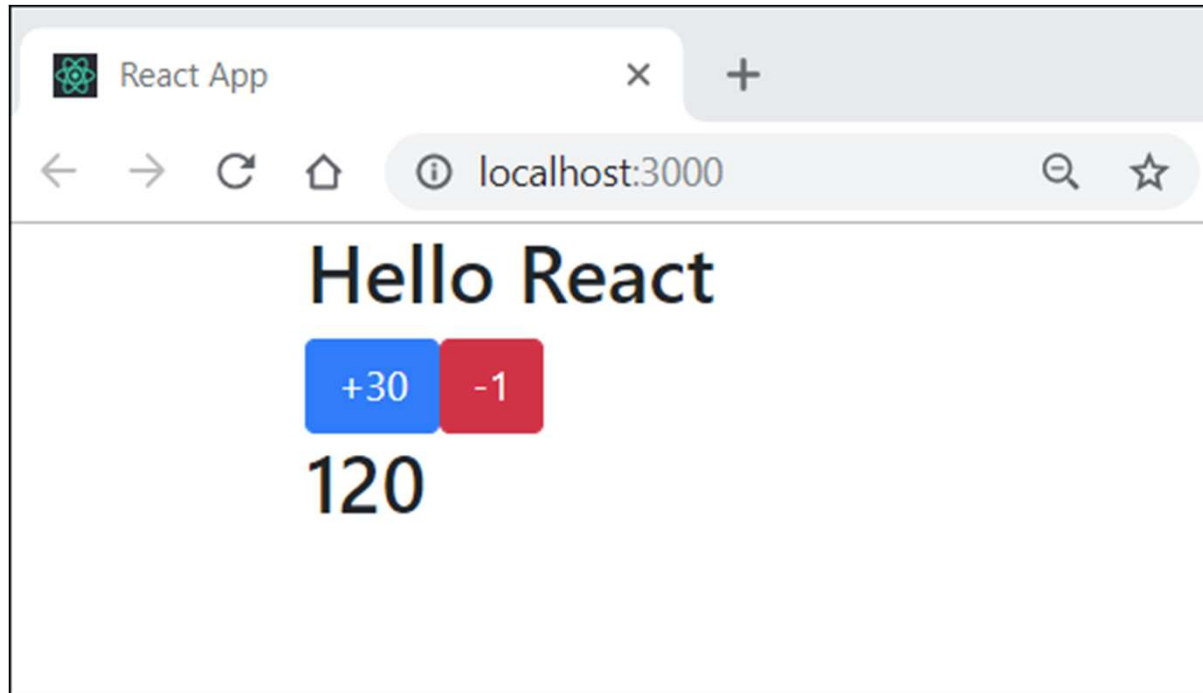


```
<Counter increment={incrementCounter} val={30} />
```

```
function Counter(props) {  
  return (  
    <div>  
      <button className="btn ..."   
        onClick={() => props.increment(props.val)}>  
        {props.val}  
      </button>  
    </div>  
  )  
}
```

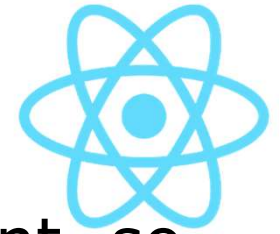
A red arrow pointing upwards from the bottom of the code block towards the `onClick` prop in the `<button>` element.

Result



Example `../130-props-parameters`

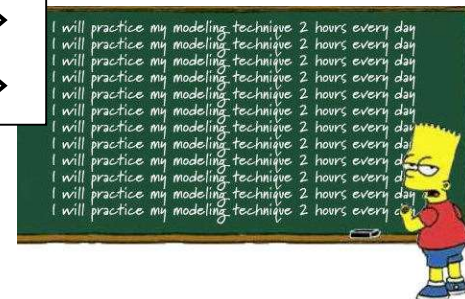
Workshop

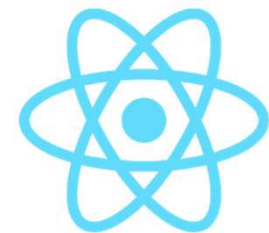


- Create a more generic `<Counter />` component, so you can add/subtract a random number from the counter.
- Start from `../130-props-parameters`
- You should be able to call it like this:

```
<Counter increment={incrementCounter} val={1}/>  
<Counter increment={incrementCounter} val={5}/>  
<Counter increment={incrementCounter} val={10}/>  
<Counter increment={incrementCounter} val={50}/>
```

(workshops: ../1-generic-counter)





Checkpoint

- There are two types of components. `function-based` and `class-based`
 - We're going to use class-based components next.
- You know how to load `3rd party libraries`.
- You can add and use as many components as you like.
- You know about component `state` and `React Hooks`
- You know how to pass state and functionality as `props` to child components.