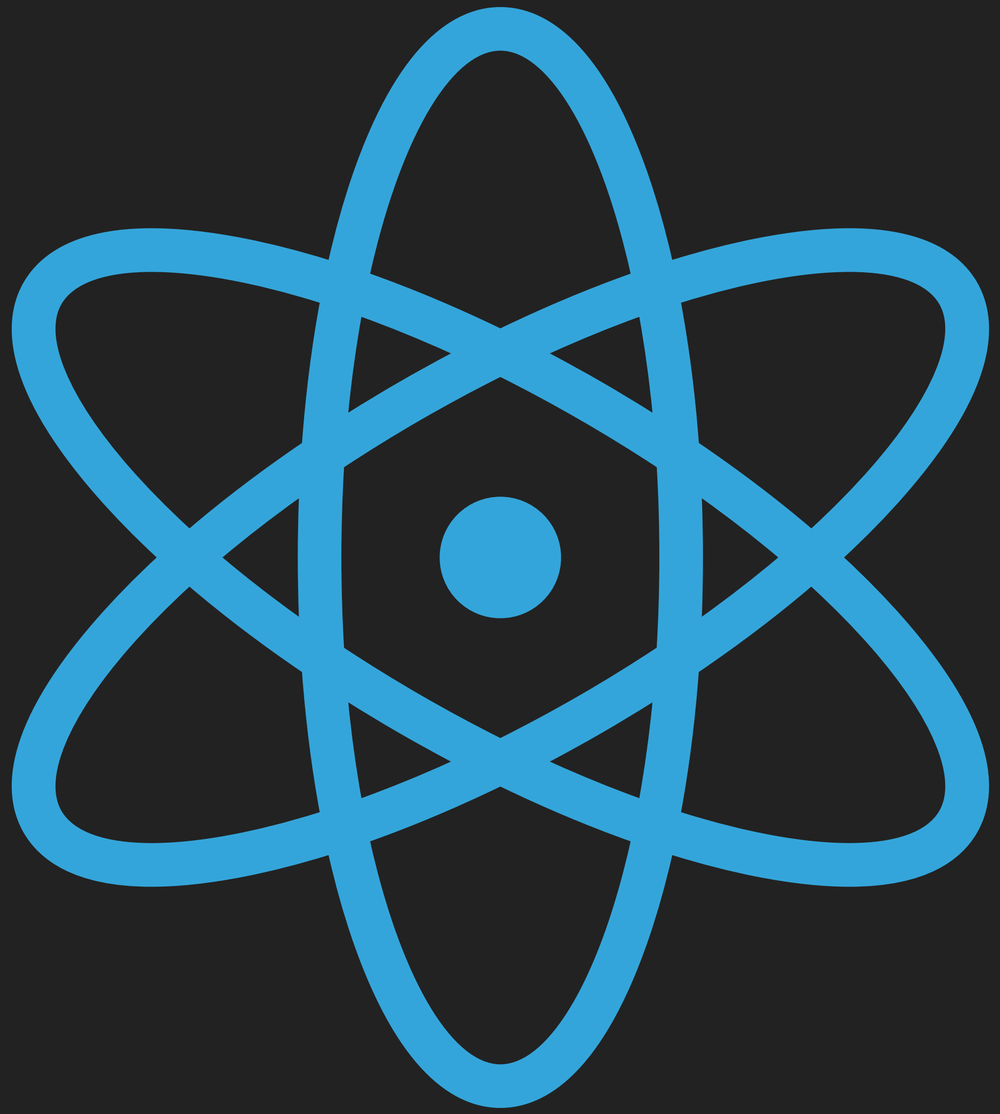
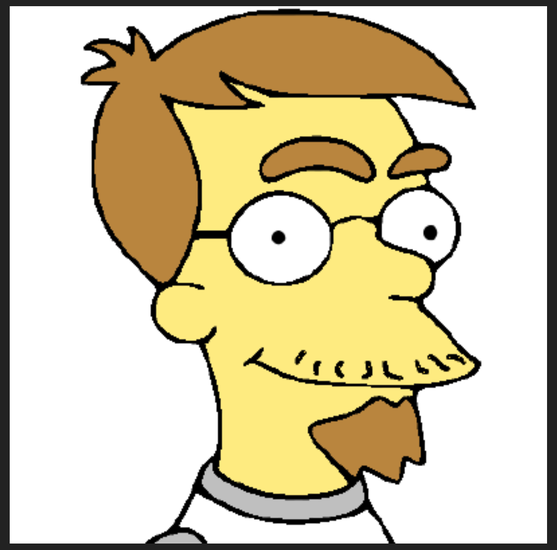


HOW REACT RENDERS

SOURCE OF MATERIALS

SOURCE OF MATERIALS

- Awesome blog of Mark Erikson



MOTIVATION

> Writing is a nature's way of letting you know how sloppy your thinking is
>
> Guindon

WHAT IS RENDERING?

WHAT IS RENDERING?

- It is a process when React asks your components to describe what they want to look like. *
- It is a process when react traverse your components tree, starting from the root, and calls render API to get «description» of UI they want to represent.

WHAT COMPONENTS WILL BE RENDERED?

- Initial render: all components.
- Subsequent renders: components, marked as needing to be updated.

HOW TO DESCRIBE UI

- React.createElement(…)
- JSX

- JSX will be transformed in React.createElement(…) calls
- React.createElement(…) returns plain object, representing an element

RENDERING EXAMPLE

```
const SomeWrapper = (props) => <div>{props.children}</div>;

const jsxOutput = <SomeWrapper><p>JSX output</p></SomeWrapper>;

const createElementOutput = React.createElement(SomeWrapper, null, [
  React.createElement('p', null, 'createElement(...) output')
]);

console.log('JSX output', jsxOutput);
console.log('React.createElement output', createElementOutput);

ReactDOM.render(
  <>
    {jsxOutput}
    {createElementOutput}
  </>,
  document.getElementById('renderOutput'),
);
```

IMPORTANT POINTS

- Rendering result is an elements tree.
- Elements tree is referred to as the «Virtual DOM».
- After render has finished React makes diffing between current and previous element trees to find differences.
- If there are differences, React calculates updates to apply to the «Real DOM».
- Diffing + Calculation = Reconciliation

RENDERING WORK PHASES

- Render Phase itself – get elements tree (calling render API), find differences and calculate UI updates.
- Commit Phase – apply calculated changes to the «Real DOM».

WHAT ABOUT LIFECYCLE CALLS?

- After React has updated DOM, it fires «componentDidMount», «componentDidUpdate» and «useLayoutEffect». It is a sync process.
- After short timeout «useEffect» hooks are going to be run. This step is also known as «Passive Events Phase».

CONCURRENT MODE

- In this mode React might pause Render Phase to allow a browser to process events.
- Then React will either proceed, throw away or recalculate this stopped rendering work.
- After finishing async render work, React will commit changes in one step, in a sync way.

KEY POINTS

- Rendering ≠ DOM Updating
- Render call may result in the same output, so no UI changes are needed
- In Concurrent Mode rendering step is divided into several shreds. So, React might reject already finished work if any updates invalidate this work.

RENDERING DETAILS

HOW TO RENDER AFTER INITIAL RENDER HAS FINISHED

CLASS COMPONENTS

this.setState(…)

this.forceUpdate(…)

FUNC COMPONENTS

React.useState(…)

React.useReducer(…)

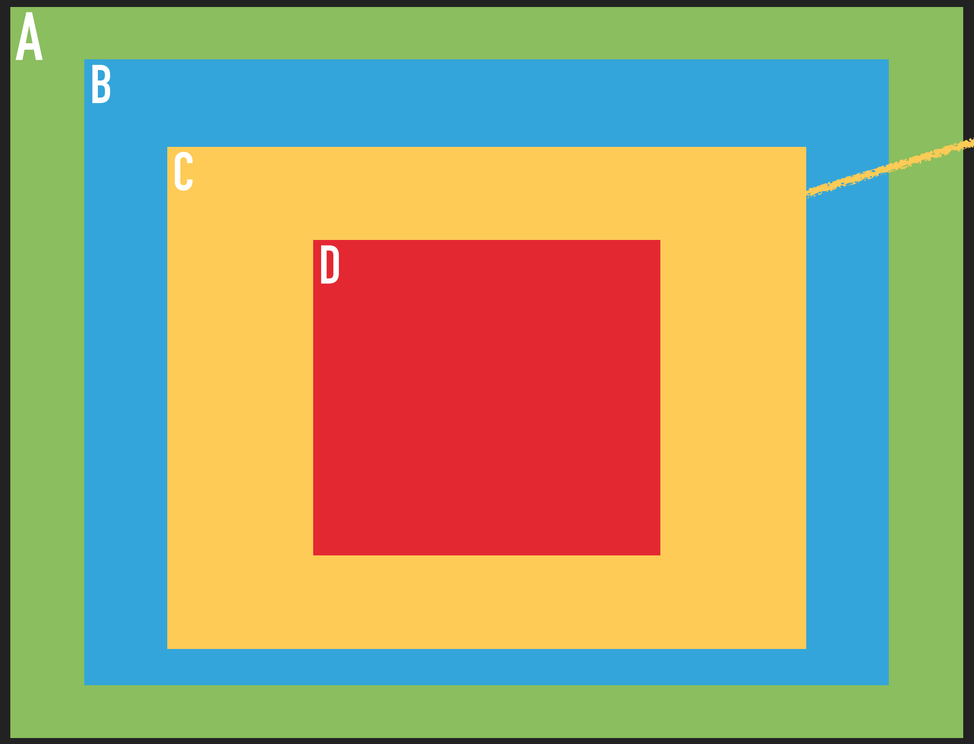
OTHERS

ReactDOM.render(…)

STANDARD RENDER BEHAVIOR

- When parent component renders, React will recursively render all child components inside of it.
- Rect doesn’t care whether «props» changed or not. It renders child components just because the parent rendered!

STANDARD RENDER BEHAVIOR EXAMPLE



const [counter, setCounter] = React.useState(0);

EXPLORE CODE EXAMPLE HERE!

Act like we're redrawing the entire app on every update

React Team *

. . . rendering is not a bad thing — it's how React knows whether it needs to actually make any changes to the DOM!

Mark Erikson *

HOW TO IMPROVE PERFORMANCE?

APPROACHES

- Do less work.
- Do the same work faster.

To improve React’s performance we should do less work!

HOW TO DO LESS WORK?

- Rendering should depend on «props» and «state». So…
- …we can check whether they changed. And…
- …if there are no changes, rendering should be skipped.

RENDER BATCHING

- This is an internal optimization used by React.
- Multiple «this.setState(…)» calls will be
  - pushed to queue, and
  - executed by groups

This technique allows to reduce «this.setState(…)» calls, resulting in a single render pass.

State Updates May Be Asynchronous

React Docs *

RENDER BATCHING | IMPLEMENTATION

- React wraps event handlers with internal function «unstable_batchedUpdates».
- Then React is watching state updates inside this wrapper function.
- All state updates pushed to the queue will be applied in a single pass.

RENDER BATCHING | PSEUDOCODE

```
const internalEventHandler = (event) => {
  const batchedUpdatesQueue = [];
  const userProvidedEventHandler = findUserProvidedEventHandler(event);

  unstable_batchedUpdates(() => {
    // Any updates, queued inside "userProvidedEventHandler",
    // will be pushed into "batchedUpdatesQueue".
    userProvidedEventHandler(event);
  });

  renderWithQueuedStateUpdates(batchedUpdatesQueue);
};
```

RENDER BATCHING | KEY POINTS

- Any state updates performed outside «unstable_batchedUpdates» internal function will not be batched together.
- React batches just updates performed within one event handler.

RENDER BATCHING | REAL WORLD EXAMPLE

```
const App = () => {
  const [counter, setCounter] = React.useState(0);
  const fetchData = async () => ({ data: 'Data from BE.' });

  console.log('Render! Counter: ', counter);

  const handleButtonClick = React.useCallback(async () => {
    setCounter(1);
    setCounter(2);

    const data = await fetchData();

    setCounter(3);
    setCounter(4);
  }, [counter]);

  return <button onClick={handleButtonClick}>Click me!</button>;
};
```

RENDER BATCHING | REAL WORLD EXAMPLE EXPLANATIONS

- The first pass batches «setCounter(1)» and «setCounter(2)». This is because they both happen inside «unstable_batchedUpdates(…)» call.
- «setCounter(3)» & «setCounter(4)» happens after an «await». And it is beyond the «unstable_batchedUpdates(…)» call. For this reason these two calls are not batched.

FACTS ABOUT «UNSTABLE_BATCHED_UPDATES» FUNCTION

- It is unstable one and is NOT officially supported part of the React API.
- React team says that «it's the most stable of the «unstable» APIs, and half of the code at Facebook relies on that function».
- This function is exported from ReactDOM. But, please, don't utilize it for your own purposes!
- In Concurrent Mode React will always batch updates. Hurra!

RENDER OPTIMIZATION TECHNIQUES

API TO SKIP RENDERING

- React.Component.shouldComponentUpdate(…)
- React.PureComponent
- React.memo(…)

HOW DO THESE «SKIPPERS» WORK?

- Any of the described approaches allow to skip rendering of an entire subtree.
- This means it changes «Standard Render Behavior».

KEY OPTIMIZATION API FOR FUNC COMPONENTS

- React.useMemo(...)
- React.useCallback(...)

HOW PROPS REFERENCES AFFECT RENDER OPTIMIZATION

- React re-renders nested components when parent changes. So…
- Passing new «props» doesn’t matter!

```
const Parent = () => {
  const [counter, setCounter] = React.useState(0);
  const handleClick = () => { setCounter(counter + 1); };
  const data = { text: 'Hello World!' };

  console.log('Render <Parent />');

  return (
    <>
      <Child data={data} />
      <button onClick={handleClick}>Counter: {counter}</button>
    </>
  );
};
```

WHEN DO REFERENCES MATTER?

```
const ChildMemoized = React.memo(Child);

const Parent = () => {
  const [counter, setCounter] = React.useState(0);
  const handleClick = () => { setCounter(counter + 1); };
  const data = React.useMemo(() => ({ text: 'Hello World!' }), []);

  console.log('Render <Parent />');

  return (
    <>
      <ChildMemoized data={data} />
      <button onClick={handleClick}>Counter: {counter}</button>
    </>
  );
};
```

DON'T OPTIMIZE HOST COMPONENTS

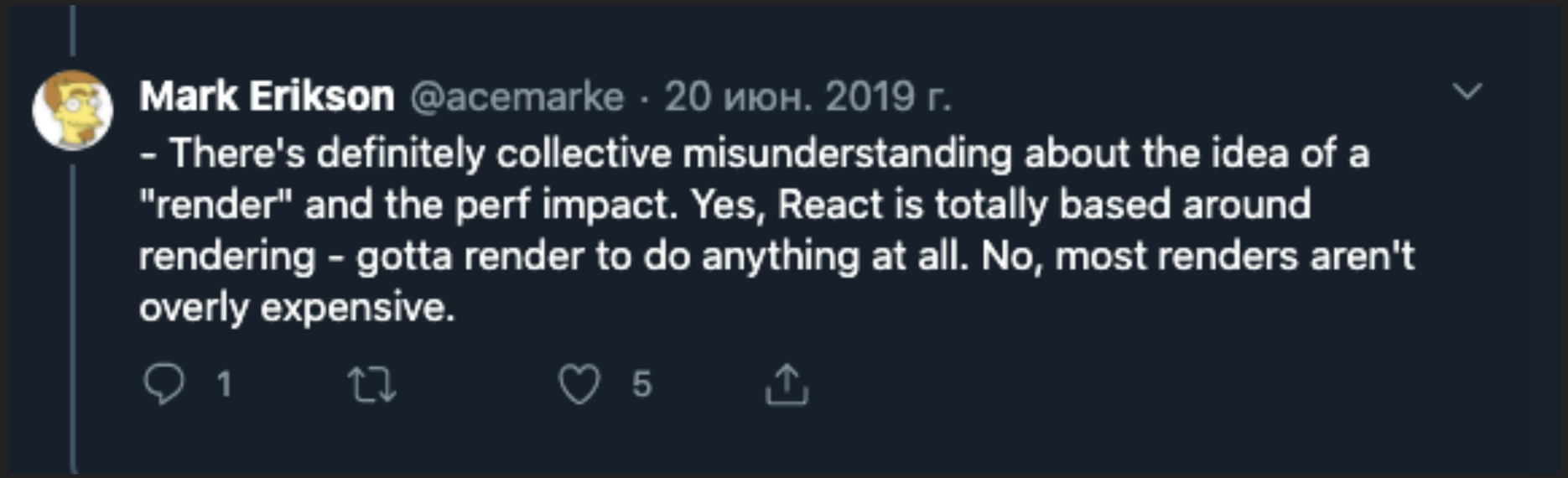
```
const App = () => {
  const [counter, setCounter] = React.useState(0);
  const handleClick = () => { setCounter(counter + 1) };

  return <button onClick={handleClick}>Counter: {counter}</button>;
};
```

PROPS WITH CHILDREN

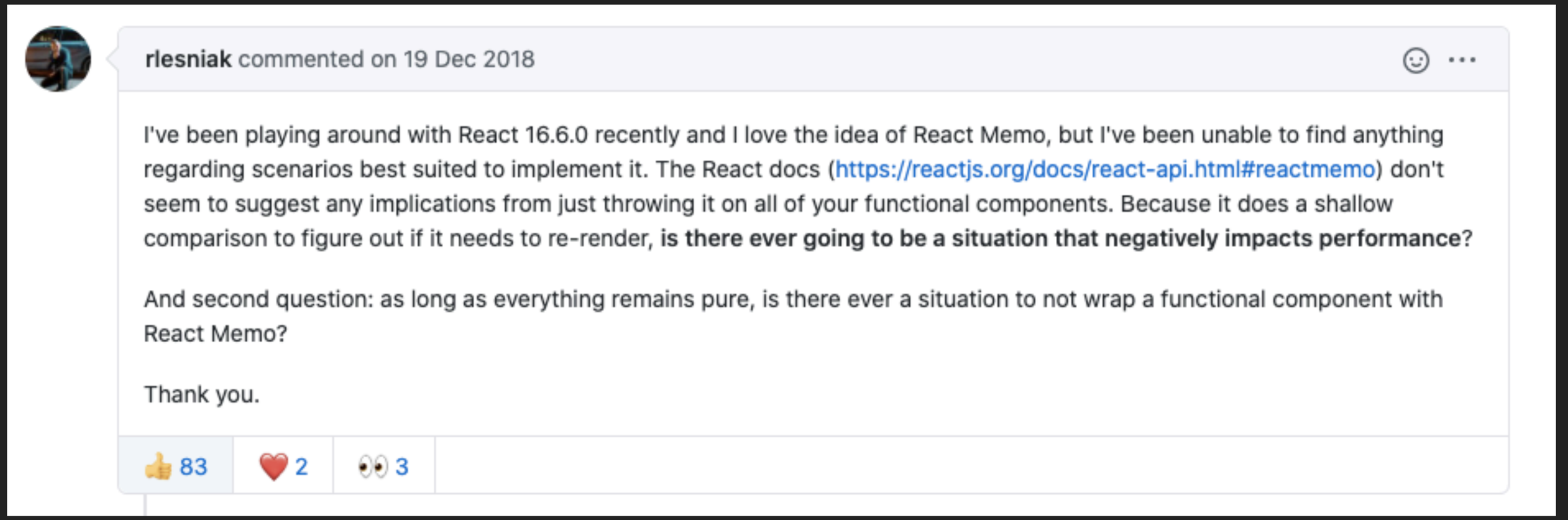
```
const Parent = () => {
  const [counter, setCounter] = React.useState(0);
  const handleClick = () => { setCounter(counter + 1); };
  const data = React.useMemo(() => ({ text: 'Hello World!' }), []);

  console.log('Render <Parent />');

  return (
    <>
      <ChildMemoized data={data}>
        <span>Hello World!</span>
      </ChildMemoized>
      <button onClick={handleClick}>Counter: {counter}</button>
    </>
  );
};
```

MEMOIZE EVERYTHING?

STILL WANT TO MEMOIZE EVERYTHING?



AND AGAIN ABOUT MEMOIZATION...



rlesniak commented on 19 Dec 2018

I've been playing around with React 16.6.0 recently and I love the idea of React Memo, but I've been unable to find anything regarding scenarios best suited to implement it. The React docs (https://reactjs.org/docs/react-api.html#reactmemo) don't seem to suggest any implications from just throwing it on all of your functional components. Because it does a shallow comparison to figure out if it needs to re-render, **is there ever going to be a situation that negatively impacts performance**?

And second question: as long as everything remains pure, is there ever a situation to not wrap a functional component with React Memo?

Thank you.

👍 83 ❤️ 2 👀 3

IMMUTABILITY MATTERS. FUNC COMPONENTS

```
const App = () => {
  const [data, setData] = React.useState({ counter: 0 });

  const handleMutableClick = React.useCallback(() => {
    data.counter = data.counter + 1;

    setData(data);
  }, [data]);

  const handleImmutableClick = React.useCallback(() => {
    setData({ ...data, counter: data.counter + 1 });
  }, [data]);
```

IMMUTABILITY MATTERS (OR NOT?). CLASS COMPONENTS

```
class App extends React.Component {
  state = {
    data: { counter: 0 },
  };

  handleMutableClick = () => {
    this.state.data.counter = this.state.data.counter + 1;

    this.setState({ data: this.state.data });
  };

  handleImmutableClick = () => {
    this.setState({
      data: {
        counter: this.state.data.counter + 1,
      },
    });
  };
```

THATS IT!

TO BE CONTINUED. . .

LINKS

- Examples GitHub