Refactoring Into React Hooks

Matteo Antony Mistretta

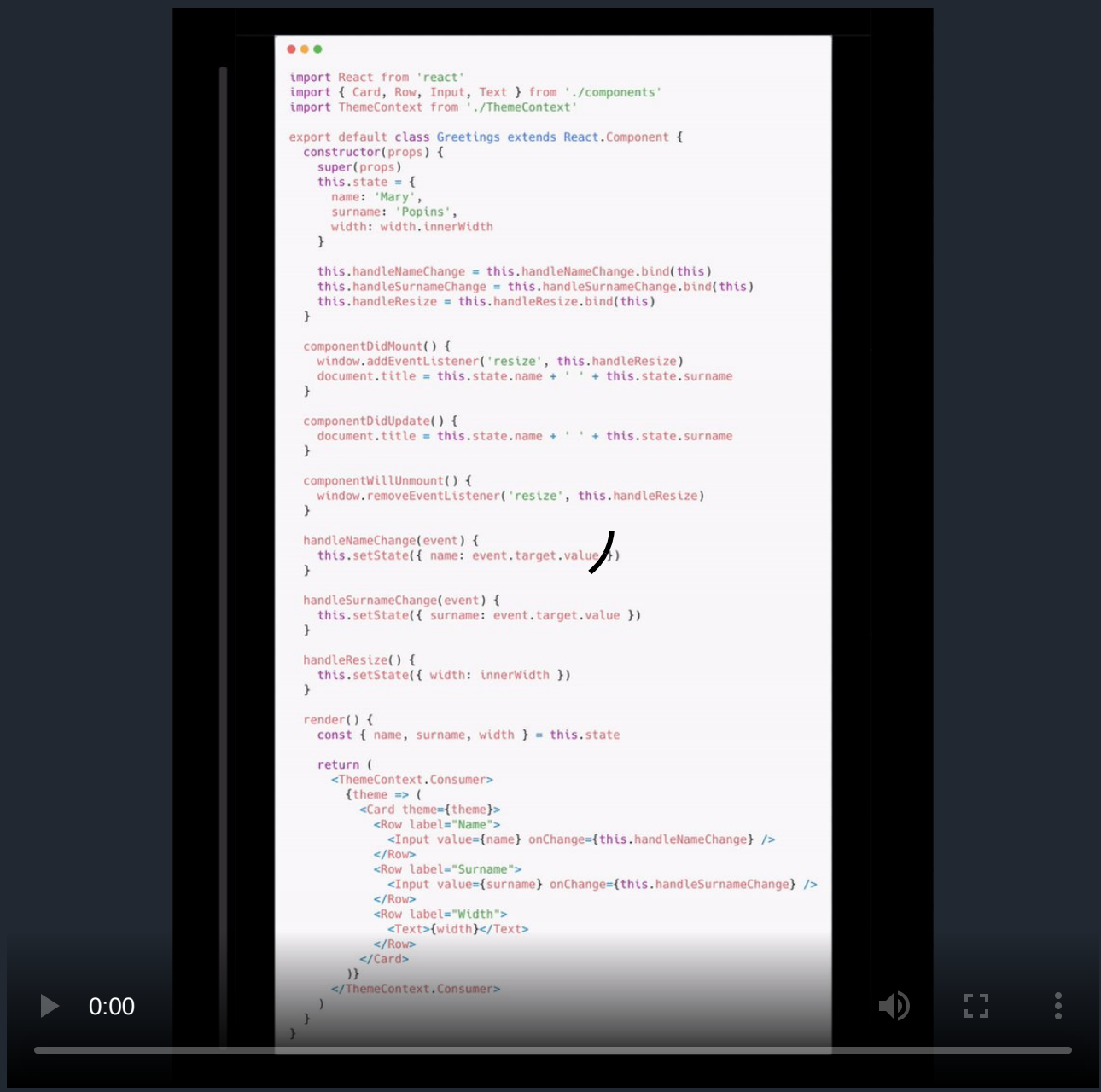
Inglorious Coderz

@antonymistretta

Why

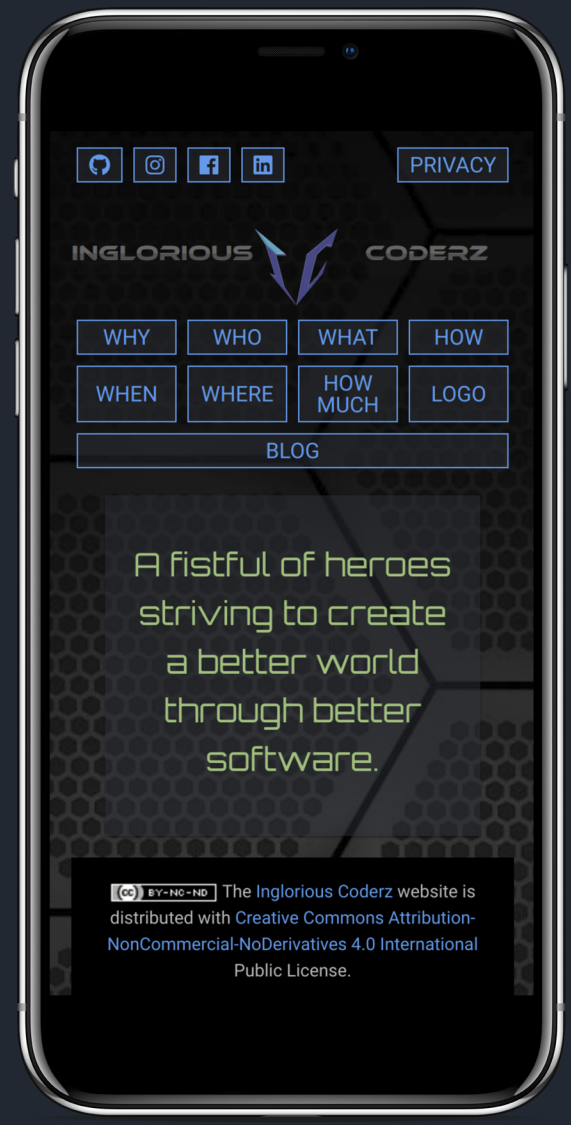
- They are stable
- They separate stateful logic
- They flatten our component hierarchy
- They allow us to go fully functional

Separation Of Concerns



Pavel Prichodko's tweet

```
antony@ingloriouscoderz ~> whoami
```




REFACTOR ALL THE THINGS!

Let's refactor...

1. *State*
2. Refs and Instance Attributes
3. Lifecycle Methods
4. Higher-Order Components
5. Render Props
6. Context API
7. Reducers
8. Redux

Hello world!

Hello world!

```
class MyComponent extends Component {
  state = { text: 'Hello world!' }

  handleChange = event => {
    this.setState({ text: event.target.value })
  }

  render() {
    const { text } = this.state
    return (
      <>
        <h1>{text}</h1>
        <input value={text} onChange={this.handleChange} />
      </>
    )
  }
}

render(<MyComponent />)
```

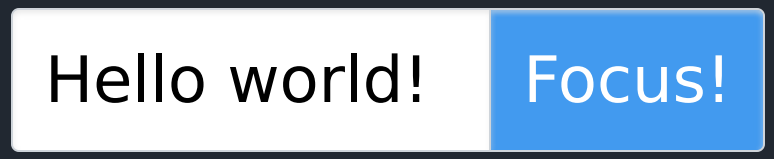
Hello world!

Hello world!

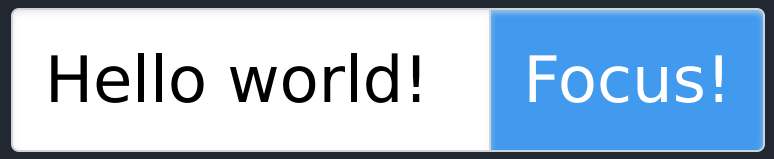
```
function MyComponent() {
  const [text, setText] = useState('Hello world!')

  function handleChange(event) {
    setText(event.target.value)
  }

  return (
    <>
      <h1>{text}</h1>
      <input value={text} onChange={handleChange} />
    </>
  )
}

render(MyComponent)
```

Let's refactor...

1. State
2. *Refs and Instance Attributes*
3. Lifecycle Methods
4. Higher-Order Components
5. Render Props
6. Context API
7. Reducers
8. Redux



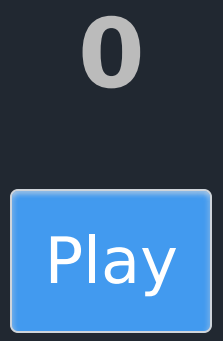

```jsx
class MyComponent extends Component {
  myRef = React.createRef()

  handleClick = () => this.myRef.current.focus()

  render() {
    return (
      <div className="input-group">
        <input defaultValue="Hello world!" ref={this.myRef} />
        <button onClick={this.handleClick}>Focus!</button>
      </div>
    )
  }
}

render(MyComponent)
```



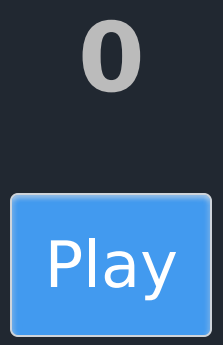

```
function MyComponent() {
  const myRef = useRef()
  const handleClick = () => myRef.current.focus()
  return (
    <div className="input-group">
      <input defaultValue="Hello world!" ref={myRef} />
      <button onClick={handleClick}>Focus!</button>
    </div>
  )
}

render(MyComponent)
```

Let's refactor...

1. State
2. Refs and Instance Attributes
3. *Lifecycle Methods*
4. Higher-Order Components
5. Render Props
6. Context API
7. Reducers
8. Redux



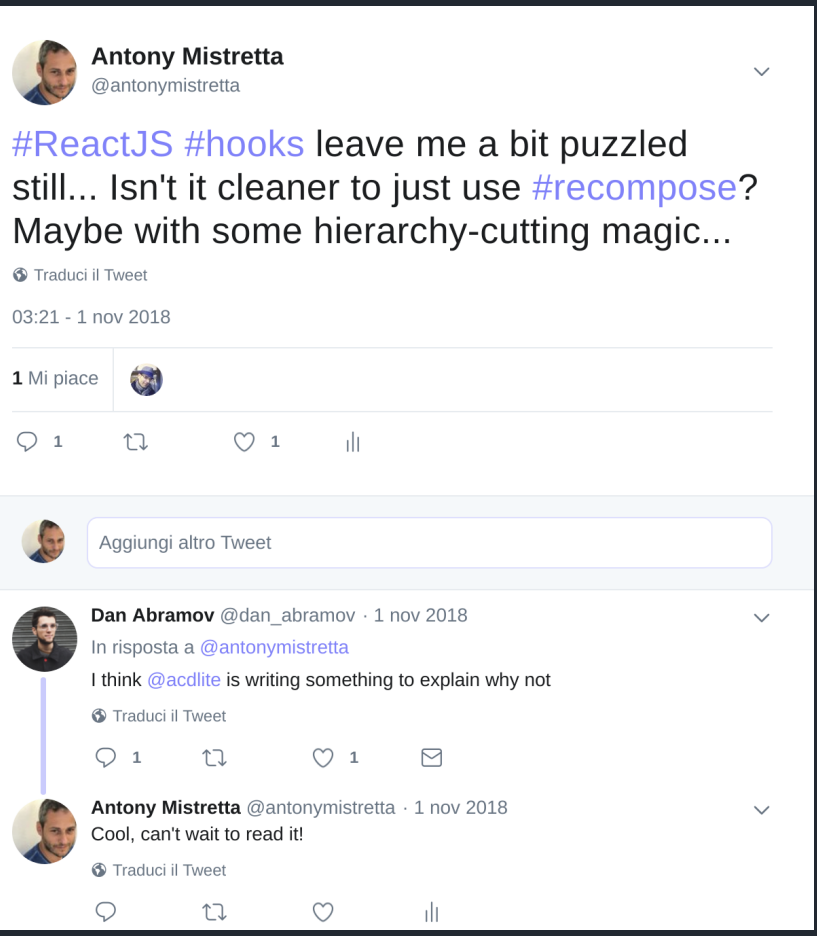

```
class MyComponent extends Component {
  state = { count: 0, play: false }

  start = () => {
    this.interval = setInterval(
      () => this.setState(({ count }) => ({ count: count + 1 })),
      1000,
    )
  }

  stop = () => clearInterval(this.interval)

  toggle = () => this.setState(({ play }) => ({ play: !play }))

  componentDidMount() {
    const { play } = this.state
    if (play) {
      this.start()
    }
  }

  componentDidUpdate(prevProps, prevState) {
    const { play } = this.state
    if (play !== prevState.play) {
      if (play) {
        this.start()
      } else {
        this.stop()
      }
    }
  }

  componentWillUnmount() {
    this.stop()
  }

  render() {
    const { count, play } = this.state
    return (
      <>
        <h1>{count}</h1>
        <button onClick={this.toggle}>{play ? 'Pause' : 'Play'}</button>
      </>
    )
  }
}

render(MyComponent)
```




```jsx
function MyComponent() {
  const [count, setCount] = useState(0)
  const [play, setPlay] = useState(false)

  function toggle() {
    setPlay(play => !play)
  }

  useEffect(() => {
    let interval = null

    function start() {
      interval = setInterval(() => setCount(count => count + 1), 1000)
    }

    function stop() {
      clearInterval(interval)
    }

    if (play) {
      start()
    } else {
      stop()
    }

    return () => stop()
  }, [play])

  return (
    <>
      <h1>{count}</h1>
      <button onClick={toggle}>{play ? 'Pause' : 'Play'}</button>
    </>
  )
}

render(MyComponent)
```

Let's refactor...

1. State
2. Refs and Instance Attributes
3. Lifecycle Methods
4. *Higher-Order Components*
5. Render Props
6. Context API
7. Reducers
8. Redux


Antony Mistretta
@antonymistretta
#ReactJS #hooks leave me a bit puzzled still... Isn't it cleaner to just use #recompose? Maybe with some hierarchy-cutting magic...
Traduci il Tweet
03:21 - 1 nov 2018
1 Mi piace
1
1
Aggiungi altro Tweet
Dan Abramov @dan_abramov · 1 nov 2018
In risposta a @antonymistretta
I think @acdlite is writing something to explain why not
Traduci il Tweet
1
1
Antony Mistretta @antonymistretta · 1 nov 2018
Cool, can't wait to read it!
Traduci il Tweet

A Note from the Author (acdlite, Oct 25 2018):

Hi! I created Recompose about three years ago. About a year after that, I joined the React team. Today, we announced a proposal for *Hooks*. Hooks solves all the problems I attempted to address with Recompose three years ago, and more on top of that. I will be discontinuing active maintenance of this package (excluding perhaps bugfixes or patches for compatibility with future React releases), and recommending that people use Hooks instead. **Your existing code with Recompose will still work**, just don't expect any new features. Thank you so, so much to @wuct and @istarkov for their heroic work maintaining Recompose over the last few years.

gaearon commented on 14 Nov 2018 • edited

To clarify:

Andrew hasn't been working on new features in Recompose for two years. So declaring that he doesn't plan to add new features to it doesn't change anything in practice for existing users.

Andrew feels uneasy about recommending Recompose for new projects when Hooks solve a large subset of the same problems without introducing excessive tree nesting and similar issues. I agree the wording was perhaps too strong but he's the maintainer and he has the right to point out its flaws. He plans to write a longer article to explain what he meant because it seems like there's a lot of FUD going on.

If you're happy with Recompose and don't experience its downsides, you can keep on using it without any issues. New versions of Recompose will continue to be released together with React updates etc.

👍 55 😄 3 ❤️ 14

Hello world!

Hello world!

```
const enhance = compose(
  withState('text', 'setText', 'Hello world!'),
  withHandlers({
    onChange: ({ setText }) => event => setText(event.target.value),
  }),
  pure,
)

const MyComponent = enhance(({ text, onChange }) => (
  <>
    <h1>{text}</h1>
    <input value={text} onChange={onChange} />
  </>
))

render(MyComponent)
```

Hello world!

Hello world!

```
const MyComponent = memo(function MyComponent() {
  const text = useText()
  return (
    <>
      <h1>{text.value}</h1>
      <input {...text} />
    </>
  )
})

function useText() {
  const [text, setText] = useState('Hello world!')
  const handleChange = event => setText(event.target.value)
  return { value: text, onChange: handleChange }
}

render(MyComponent)
```

Let's refactor...

1. State
2. Refs and Instance Attributes
3. Lifecycle Methods
4. Higher-Order Components
5. *Render Props*
6. Context API
7. Reducers
8. Redux


```jsx
class Toggler extends Component {
  state = { on: this.props.defaultOn }

  toggle = () => this.setState(({ on }) => ({ on: !on }))

  render() {
    const { children } = this.props
    const { on } = this.state
    return children({ on, toggle: this.toggle })
  }
}

function Child({ on, toggle }) {
  return <button onClick={toggle}>{on ? 'Turn off' : 'Turn on'}</button>
}

function Parent() {
  return (
    <Toggler defaultOn={false}>
      {({ on, toggle }) => <Child on={on} toggle={toggle} />}
    </Toggler>
  )
}

render(Parent)
```

Turn on

```
function useToggle(defaultOn) {
  const [on, setOn] = useState(defaultOn)
  const toggle = useCallback(() => setOn(!on), [on])
  return { on, toggle }
}

function Child() {
  const { on, toggle } = useToggle(false)
  return <button onClick={toggle}>{on ? 'Turn off' : 'Turn on'}</button>
}

function Parent() {
  return <Child />
}

render(Parent)
```

Let's refactor...

1. State
2. Refs and Instance Attributes
3. Lifecycle Methods
4. Higher-Order Components
5. Render Props
6. *Context API*
7. Reducers
8. Redux

Hello Antony!

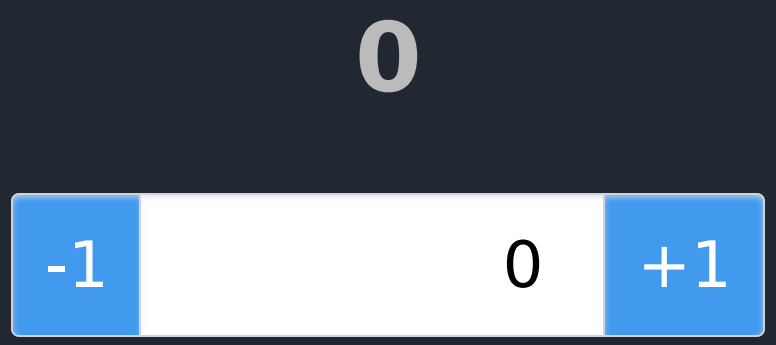
```jsx
const UserContext = createContext()
const ThemeContext = createContext()

function Parent() {
  return (
    <UserContext.Provider value="Antony">
      <ThemeContext.Provider value={{ color: '#e06c75' }}>
        <Child />
      </ThemeContext.Provider>
    </UserContext.Provider>
  )
}

function Child() {
  return (
    <UserContext.Consumer>
      {user => (
        <ThemeContext.Consumer>
          {theme => <h1 style={theme}>Hello {user}!</h1>}
        </ThemeContext.Consumer>
      )}
    </UserContext.Consumer>
  )
}

render(Parent)
```

Hello Antony!

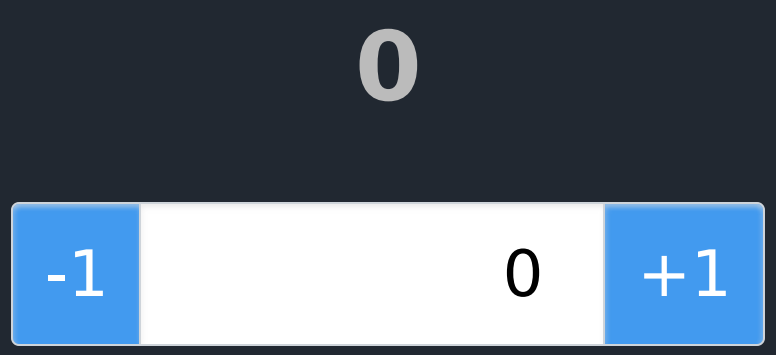
```jsx
const UserContext = createContext()
const ThemeContext = createContext()

function Parent() {
  return (
    <UserContext.Provider value="Antony">
      <ThemeContext.Provider value={{ color: '#e06c75' }}>
        <Child />
      </ThemeContext.Provider>
    </UserContext.Provider>
  )
}

function Child() {
  const user = useContext(UserContext)
  const theme = useContext(ThemeContext)
  return <h1 style={theme}>Hello {user}!</h1>
}

render(Parent)
```

Let's refactor...

1. State
2. Refs and Instance Attributes
3. Lifecycle Methods
4. Higher-Order Components
5. Render Props
6. Context API
7. *Reducers*
8. Redux




```
function counter(state = 0, action) {
  const { type, payload } = action
  switch (type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    case 'SET_VALUE':
      return payload
    default:
      return state
  }
}

const enhance = compose(
  withReducer('count', 'dispatch', counter, 0),
  withHandlers({
    increment: ({ dispatch }) => () => dispatch({ type: 'INCREMENT' }),
    decrement: ({ dispatch }) => () => dispatch({ type: 'DECREMENT' }),
    setValue: ({ dispatch }) => value =>
      dispatch({ type: 'SET_VALUE', payload: value }),
  }),
  withHandlers({
    handleChange: ({ setValue }) => event =>
      setValue(parseInt(event.target.value)),
  }),
)

const Counter = enhance(({ count, increment, decrement, handleChange }) => (
  <>
    <h1>{count}</h1>
    <div className="input-group">
      <button onClick={decrement}>-1</button>
      <input type="number" value={count} onChange={handleChange} />
      <button onClick={increment}>+1</button>
    </div>
  </>
))

render(Counter)
```

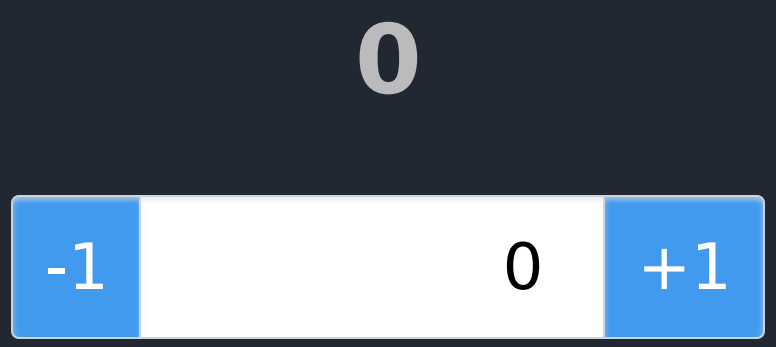



```jsx
function Counter() {
  const { count, increment, decrement, handleChange } = useCounter()
  return (
    <>
      <h1>{count}</h1>
      <div className="input-group">
        <button onClick={decrement}>-1</button>
        <input type="number" value={count} onChange={handleChange} />
        <button onClick={increment}>+1</button>
      </div>
    </>
  )
}

function useCounter() {
  const [count, dispatch] = useReducer(counter, 0)
  const increment = () => dispatch({ type: 'INCREMENT' })
  const decrement = () => dispatch({ type: 'DECREMENT' })
  const setValue = value => dispatch({ type: 'SET_VALUE', payload: value })
  const handleChange = event => setValue(parseInt(event.target.value))
  return { count, increment, decrement, handleChange }
}

render(Counter)
```

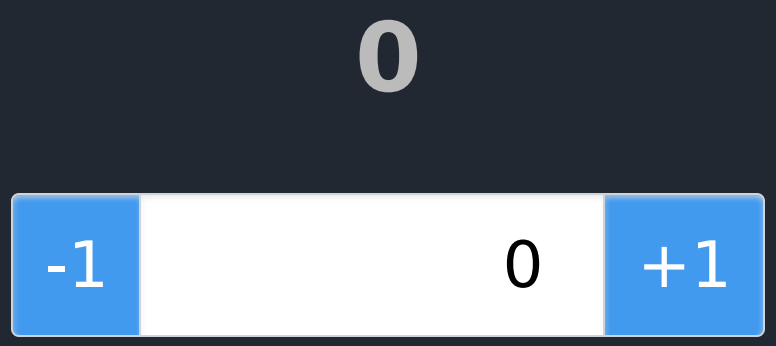
```jsx
function counter(state = 0, action) {
  const { type, payload } = action
  switch (type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    case 'SET_VALUE':
      return payload
    default:
      return state
  }
}
```

Let's refactor...

1. State
2. Refs and Instance Attributes
3. Lifecycle Methods
4. Higher-Order Components
5. Render Props
6. Context API
7. Reducers
8. *Redux*




```jsx
function counter(state = 0, action) {
  const { type, payload } = action
  switch (type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    case 'SET_VALUE':
      return payload
    default:
      return state
  }
}

const CounterContext = createContext()

class Parent extends Component {
  dispatch = action =>
    this.setState(({ count }) => ({ count: counter(count, action) }))
  increment = () => this.dispatch({ type: 'INCREMENT' })
  decrement = () => this.dispatch({ type: 'DECREMENT' })
  setValue = value => this.dispatch({ type: 'SET_VALUE', payload: value })
  handleChange = event => this.setValue(parseInt(event.target.value))
  state = {
    count: 0,
    increment: this.increment,
    decrement: this.decrement,
    handleChange: this.handleChange,
  }

  render() {
    return (
      <CounterContext.Provider value={this.state}>
        <Child />
      </CounterContext.Provider>
    )
  }
}

function Child() {
  const { count, increment, decrement, handleChange } = useContext(
    CounterContext,
  )
  return (
    <>
      <h1>{count}</h1>
      <div className="input-group">
```




```jsx
function counter(state = 0, action) {
  const { type, payload } = action
  switch (type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    case 'SET_VALUE':
      return payload
    default:
      return state
  }
}

function useCounter() {
  const [count, dispatch] = useReducer(counter, 0)
  const increment = () => dispatch({ type: 'INCREMENT' })
  const decrement = () => dispatch({ type: 'DECREMENT' })
  const setValue = value => dispatch({ type: 'SET_VALUE', payload: value })
  const handleChange = event => setValue(parseInt(event.target.value))
  return { count, increment, decrement, handleChange }
}

const CounterContext = createContext()
```

```jsx
function Parent() {
  const counter = useCounter()

  return (
    <CounterContext.Provider value={counter}>
      <Child />
    </CounterContext.Provider>
  )
}

function Child() {
  const { count, increment, decrement, handleChange } = useContext(
    CounterContext,
  )
  return (
    <>
      <h1>{count}</h1>
      <div className="input-group">
        <button onClick={decrement}>-1</button>
        <input type="number" value={count} onChange={handleChange} />
        <button onClick={increment}>+1</button>
      </div>
    </>
```

Hooks:

- Are still completely optional
- Simplify and organize code
- Are composable
- Will give performance gains
- Are subject to rules
- Will not replace everything else

Thank you.

Questions?

source code