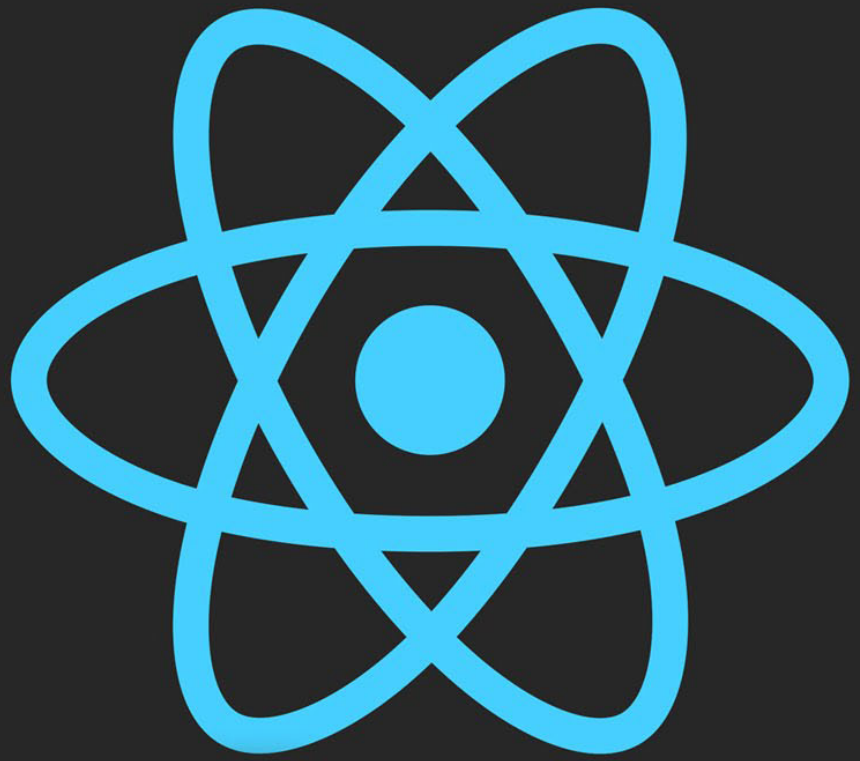
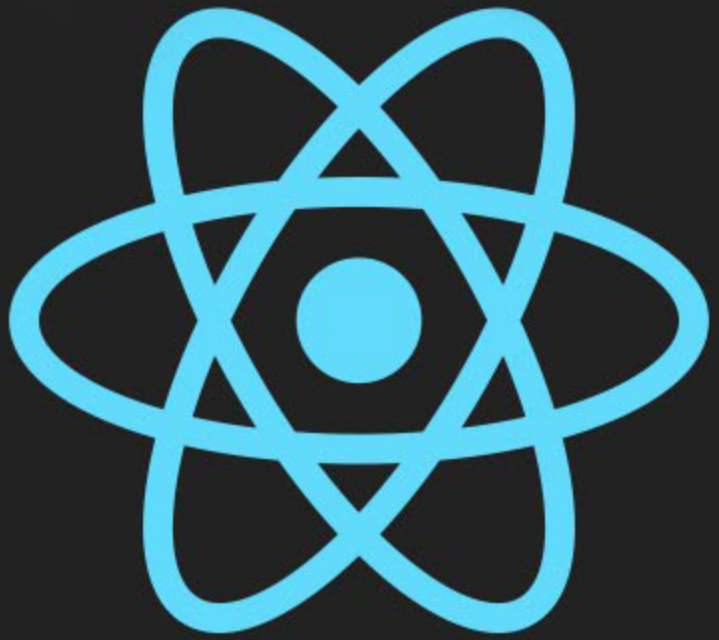


React

Introduction

@Chalach.mo @Vattanan.bu

Proptypes And DefaultProps



Proptypes

prop-types is Runtime type checking for React props and similar objects.

- As your app grows, you can catch a lot of bugs with typechecking. For some applications, you can use JavaScript extensions like Flow or TypeScript to typecheck your whole application. But even if you don't use those, React has some built-in typechecking abilities. To run typechecking on the props for a component, you can assign the special propTypes property:

Proptypes Example

```
import React from 'react';
import PropTypes from 'prop-types';


class Greeting extends Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

Proptypes Example(2)

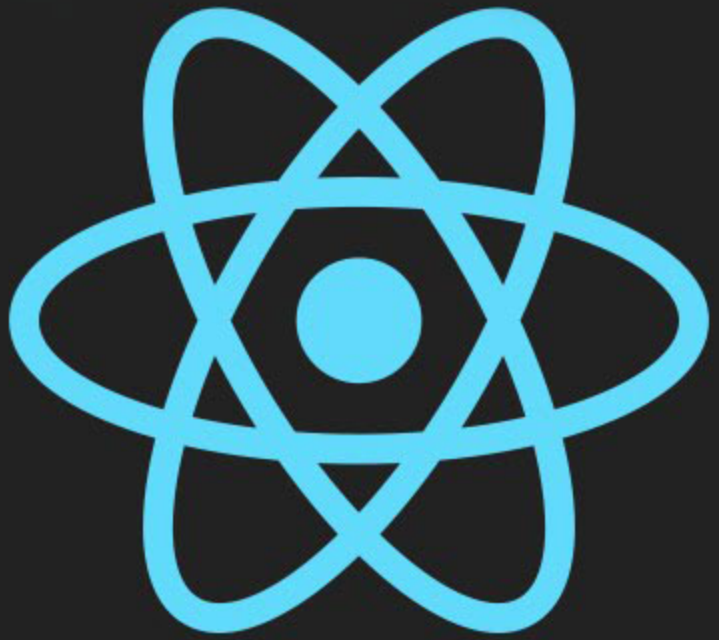
```
import PropTypes from 'prop-types';

MyComponent.propTypes = {

optionalArray      : PropTypes.array,
optionalBool       : PropTypes.bool,
optionalFunc       : PropTypes.func,
optionalNumber     : PropTypes.number,
optionalObject     : PropTypes.object,
optionalString     : PropTypes.string,
optionalSymbol     : PropTypes.symbol,
optionalNode       : PropTypes.node,
optionalElement    : PropTypes.element,
optionalMessage    : PropTypes.instanceOf(Message),
optionalEnum       : PropTypes.oneOf(['News', 'Photos']),
optionalUnion      : PropTypes.oneOfType([
                       PropTypes.string,
                       PropTypes.number,
                       PropTypes.instanceOf(Message)
                     ]),
optionalArrayOf    : PropTypes.arrayOf(PropTypes.number),
optionalObjectOf   : PropTypes.objectOf(PropTypes.number),
optionalObjectWithShape: PropTypes.shape({
          color: PropTypes.string,
          fontSize: PropTypes.number
}),
```

```
requiredFunc: PropTypes.func.isRequired,
requiredAny: PropTypes.any.isRequired,

customProp: function(props, propName, componentName) {
 if (!/matchme/.test(props[propName])) {
  return new Error('Invalid prop `' + propName + '`
            supplied to' +' `' + componentName + '`.
            Validation failed.'
   );
 }
},

 customArrayProp: PropTypes.arrayOf(function(propValue, key,
  componentName, location, propFullName) {
  if (!/matchme/.test(propValue[key])) {
   return new Error('Invalid prop `' + propFullName + '`
            supplied to' +' `' + componentName + '`.
            Validation failed.'
    );
  }
 })
};
```

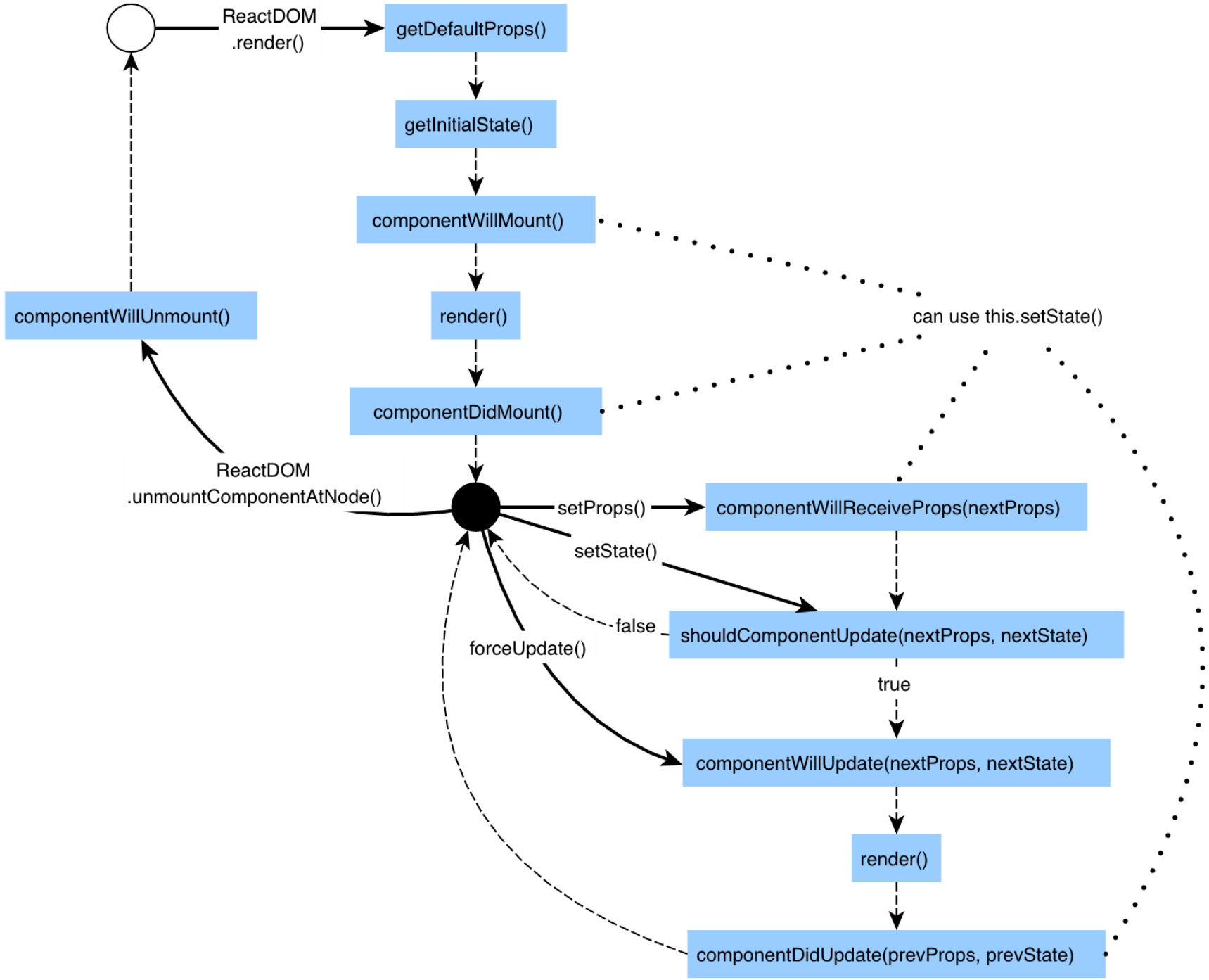
Default

You can define default values for your props by assigning to the special defaultProps property:

```
import React from 'react';

class Greeting extends React.Component {
  static defaultProps = {
    name: 'stranger'
  }
  render() {
    return (
      <div>Hello, {this.props.name}</div>
    )
  }
}
```



React Lifecycle


ReactDOM
.render()
getDefaultProps()
getInitialState()
componentWillMount()
render()
componentDidMount()
componentWillUnmount()
ReactDOM
.unmountComponentAtNode()
can use this.setState()
setProps()
componentWillReceiveProps(nextProps)
setState()
false
shouldComponentUpdate(nextProps, nextState)
forceUpdate()
true
componentWillUpdate(nextProps, nextState)
render()
componentDidUpdate(prevProps, prevState)

React Life Cycle Example

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
  componentDidMount() {

  }
  componentWillUnmount() {

  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

React Life Cycle Example(2)

These methods are called “lifecycle hooks”.

The componentDidMount() hook runs after the component output has been rendered to the DOM.
This is a good place to set up a timer:

```
componentDidMount() {
  this.timerID = setInterval(
    () => this.tick(),
    1000
  );
}
```

Note how we save the timer ID right on this.

React Life Cycle Example(3)

While this.props is set up by React itself and this.state has a special meaning, you are free to add additional fields to the class manually if you need to store something that is not used for the visual output.

If you don't use something in render(), it shouldn't be in the state.

We will tear down the timer in the componentWillUnmount() lifecycle hook:

```
componentWillUnmount() {
  clearInterval(this.timerID);
}
```

React Life Cycle Example(4)

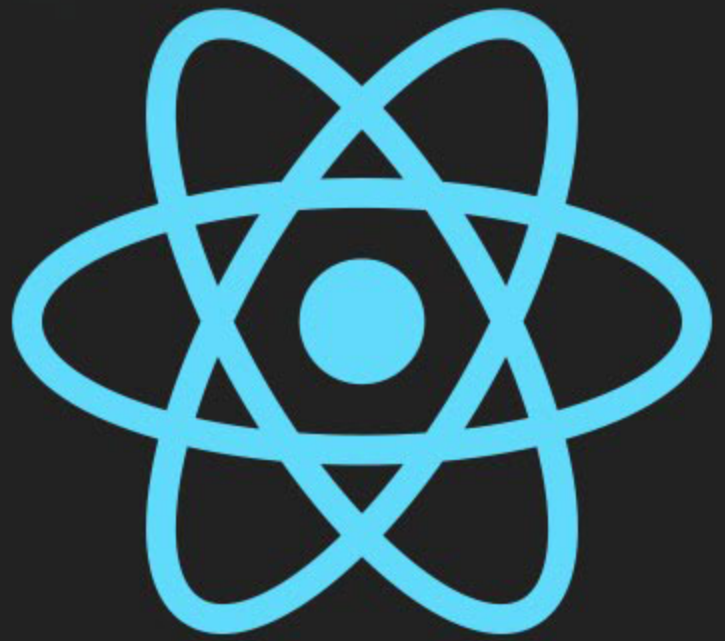
Finally, we will implement a method called tick() that the Clock component will run every second.
It will use this.setState() to schedule updates to the component local state:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }
  componentWillUnmount() {
    clearInterval(this.timerID);
  }
  tick() {
    this.setState({
      date: new Date()
    });
  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It
    is{this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

Stateless

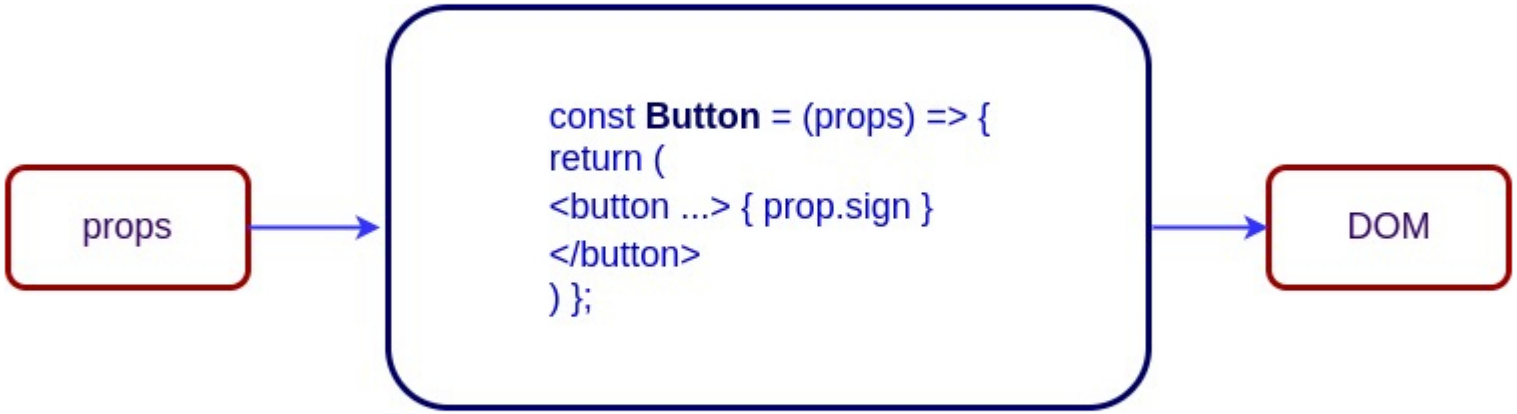
Vs Stateful

Components



Functional Component

Functional components are just JavaScript functions. They take in an optional input which, as I've mentioned earlier, is what we call props.



using an arrow function, we can skip the use of two keywords, function and return, and a pair of curly brackets. With the new syntax, you can define a component in a single line like this.

```
const Hello = ({ name }) => (<div>Hello, {name}!</div>);
```

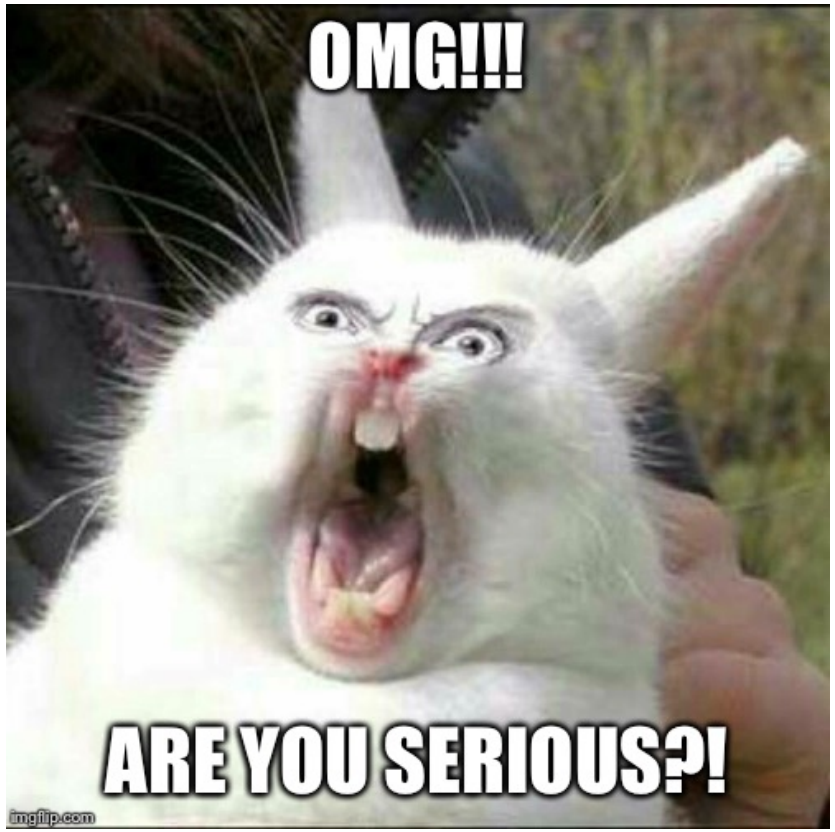
Class Component

Class components offer more features, and with more features comes more baggage. The primary reason to choose class components over functional components is that they can have state.

The state = {count: 1} syntax is part of the public class fields feature. More on this below.

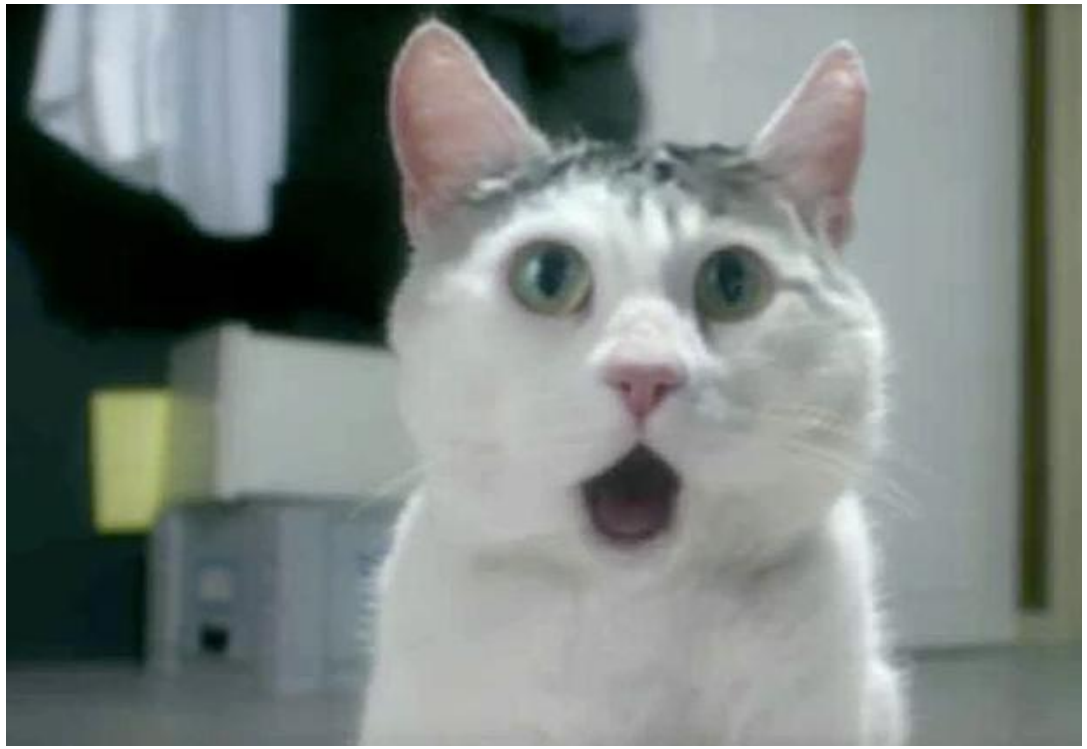
Stateful Component

Stateful Components === Class Components



Stateless Component

Stateless Components === Functional Components

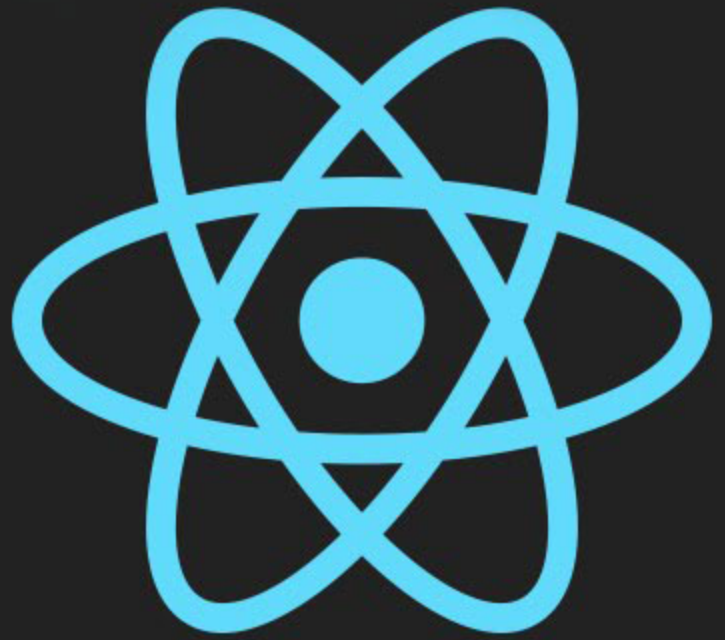


What Differrent . . .

Stateful Components === Class Components

Stateless Components === Functional Components

Controller Vs Uncontrolled Components



Controlled Component

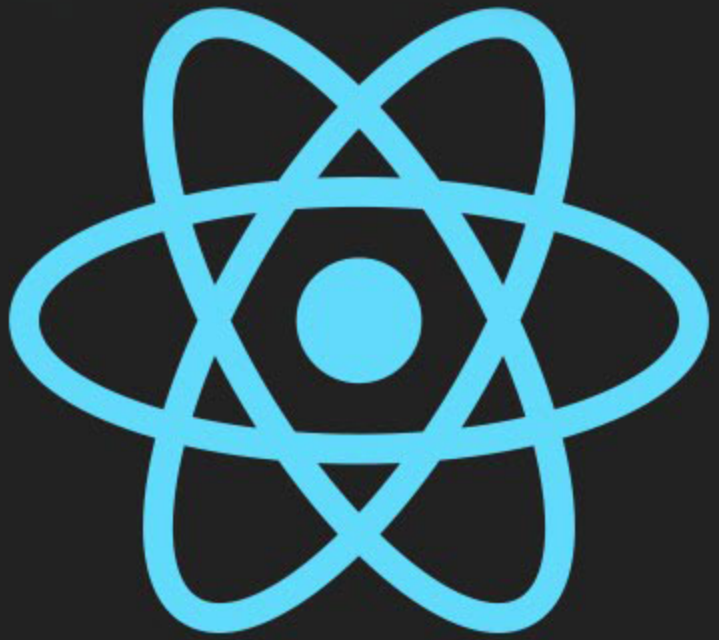
We can combine the two by making the React state be the “single source of truth”. Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a “controlled component”.

```
class App extends React.Component {
  constructor() {
    super();
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      value: '',
    };
  }
  handleChange(event) {
    this.setState({
      value: event.target.value,
    });
  }
  render() {
    return (
      <input type="text" value={this.state.value}
                         onChange={this.handleChange} />
    );
  }
}
```

Uncontrolled Component

In most cases, we recommend using **controlled component** to implement forms. In a controlled component, form data is handled by a React component. The alternative is uncontrolled components, where form data is handled by the DOM itself. To write an uncontrolled component, instead of writing an event handler for every state update, you can use a ref to get form values from the DOM.

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.value);
    event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={(input) => this.input = input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```



Higher-Order Components

High-Order Component

A higher-order component (HOC) is an advanced technique in React for reusing component logic. HOCs are not part of the React API, per se. They are a pattern that emerges from React's compositional nature. Concretely, **a higher-order component is a function that takes a component and returns a new component.**

```
import React, { Component } from 'react'

class CommentList extends Component {
  render() {
    return (
      <ul>
        {
          this.props.comments.map(comment => <li key={comment.body}>{comment.body}</li>)
        }
      </ul>
    )
  }
}

export default CommentList
```

High-Order Component Example

```
import React, { Component } from 'react'
import withLoading from '../hocs/withLoading'

class CommentList extends Component {
  render() {
    return (
      <ul>
        {
          this.props.comments.map(comment => <li key={comment.body}>{comment.body}</li>)
        }
      </ul>
    )
  }
}

export default withLoading('comments')(CommentList)
```

High-Order Component Example(2)

```
import React, { Component } from 'react'
import Loading from '../components/Loading'

const withLoading = (propName) => (WrappedComponent) => {
  return class ComponentWithLoading extends Component {
    render() {
      return this.props[propName].length === 0 ? <Loading /> : <WrappedComponent {...this.props} />
    }
  }
}

export default withLoading
```

Homework

1. https://www.kirupa.com/react/simple_todo_app_react.htm

Reference

- https://reactjs.org/docs/state-and-lifecycle.html#adding-lifecycle-methods-to-a-class
- https://reactjs.org/docs/typechecking-with-proptypes.html
- https://code.tutsplus.com/tutorials/stateful-vs-stateless-functional-components-in-react--cms-29541
- https://reactjs.org/docs/higher-order-components.html
- https://medium.com/@aofleejay/รู้จักกับ-higher-order-component-2d74ba7e1428
- **React: Functional Web Development with React and Redux** 1st Edition, Kindle Edition