React Components

November 24, 2016

Agenda

1. React components overview
2. Stateful/stateless components
3. Synthetic events and how to handle events…
4. Controlled/Uncontrolled components
5. Parent-child components relationships/communication
6. Refs, context

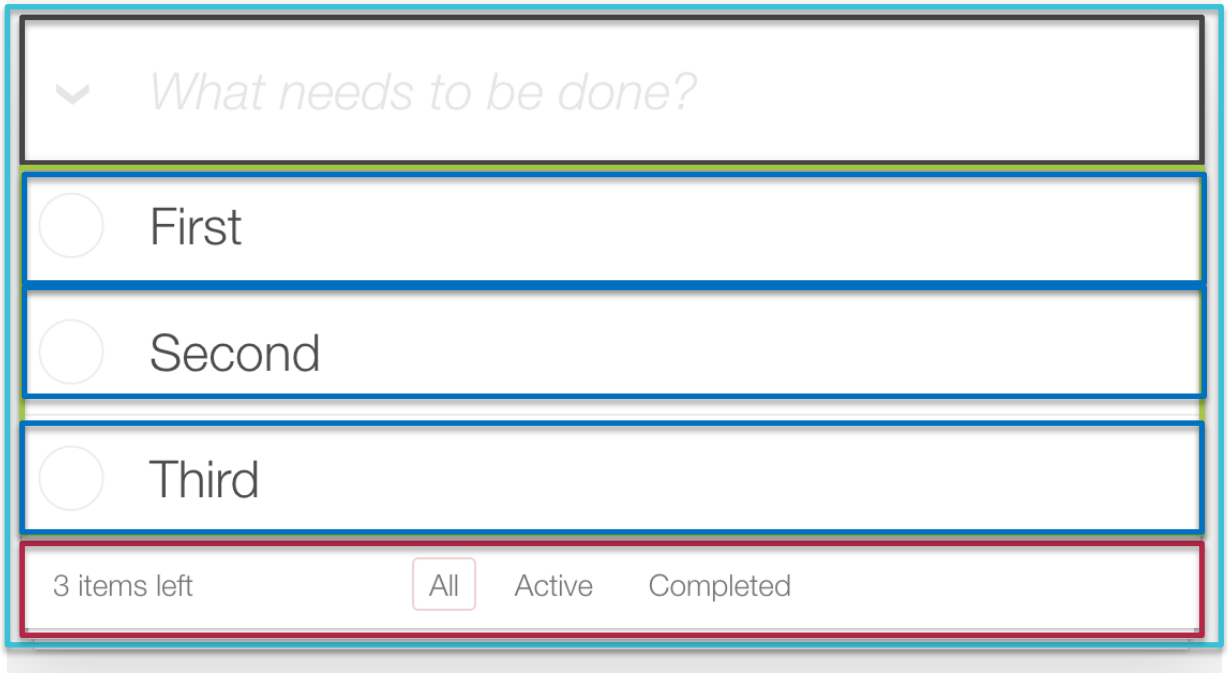
Components overview

What is Components?

React components are reusable chunks of JavaScript that output (via JSX) virtual HTML elements…

Components

React and components…



React application is a composition of components

Components

Building first component

Functional Component:

```
function MyFirstComponent(props) {
  return <div>My name is {props.name}</div>
}
```

Class Component:

```
class MyFirstComponent extends React.Component {
  render() {
    return <div>My name is {this.props.name}</div>
  }
}
```

Components are equal

State & Props

Props are immutable - some kind of component configuration, received from parent.

State - mutable. Can be changed using .setState() method inside component.

	props	state
Can get initial value from parent component?	Yes	Yes
Can be changed by parent component?	Yes	No
Can set default values inside component?	Yes	Yes
Can change inside component?	No	Yes
Can set initial value for child components?	Yes	Yes
Can be changed by child components?	Yes	No

Components

How it looks

```
class GreetingBox extends React.Component {
  constructor() {
    super();
    this.state = { name: getUniqueName() }
  }

  handleClick() {
    this.setState({ name: getUniqueName() })
  }

  render() {
    return (
      <div>
        <button className="btn" onClick={this.handleClick.bind(this)}>Change Name</button>
        <NameBox name={this.state.name}/>
      </div>
    )
  }
}

class NameBox extends React.Component {
  render() {
    return (
      <div className="name">
        My name is {this.props.name}
      </div>
    )
  }
}
```

You can define initial state inside component

You can mutate state inside component

You can pass props to the child component

You can read props

Components

How it looks

Components

If you try mutate props

Typechecking with PropTypes

React has built-in typechecking abilities for props

```
        <Hello name={234234}/>
      </div>
    )
  }
}

class Hello extends React.Component {
  render() {
    return (
      <div>
        Hello, {this.props.name}!
      </div>
    )
  }
}

Hello.propTypes = {
  name: React.PropTypes.string
};
```

If you try to pass property with invalid type then React inform you

propTypes have to be static property of component

```
▶ Warning: Failed prop type: Invalid prop `name` of type `number` supplied to `Hello`, expected `string`.
  in Hello (created by App)
  in App
  in AppContainer
```

Components

Gotchas

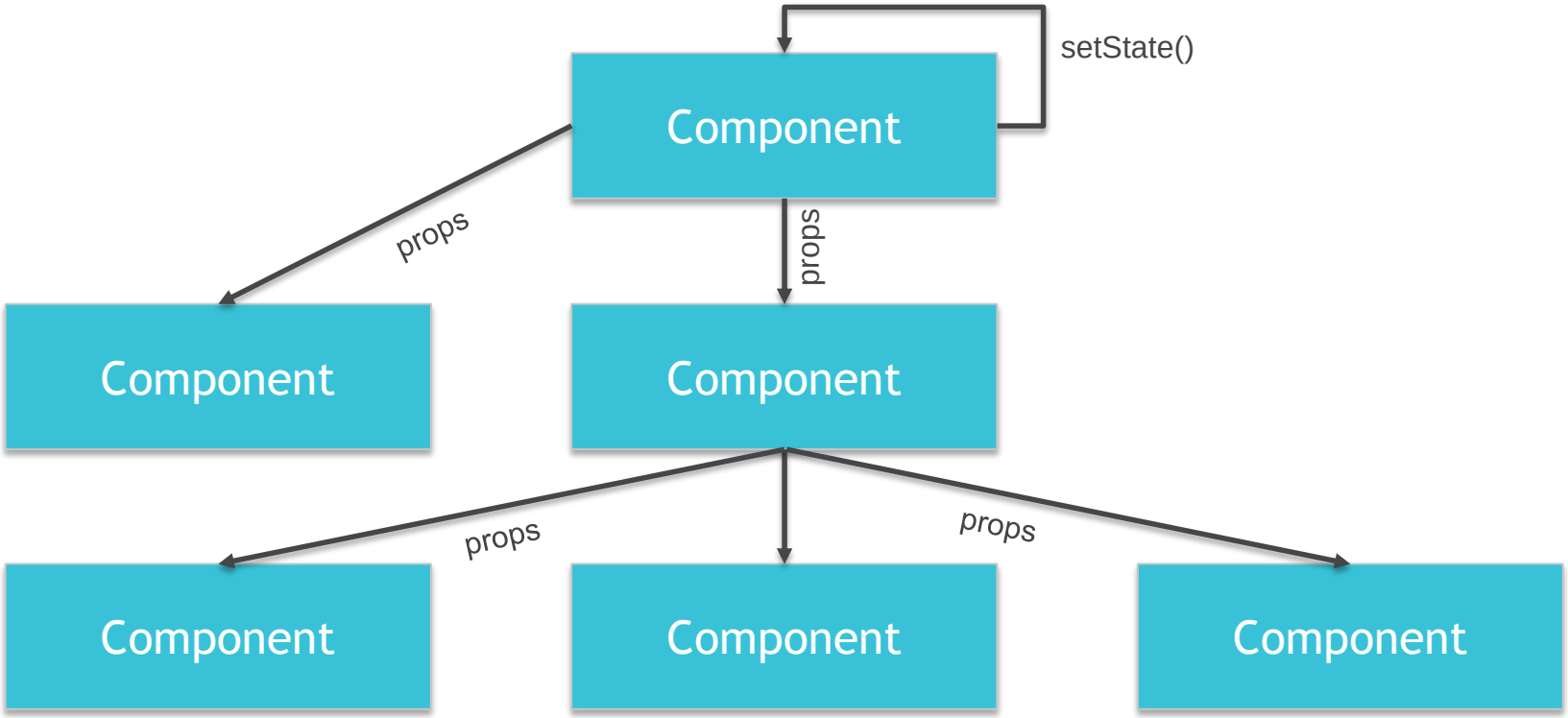
- Never call .setState() inside render()
- Don’t try to interact with browser inside render()
- Avoid side effects inside render inside render()
- Don’t modify state directly – this will not re-render a component

Use for this purposes component lifecycle methods or other methods…

Components data flow

A few words about data flow…

React provides unidirectional data flow



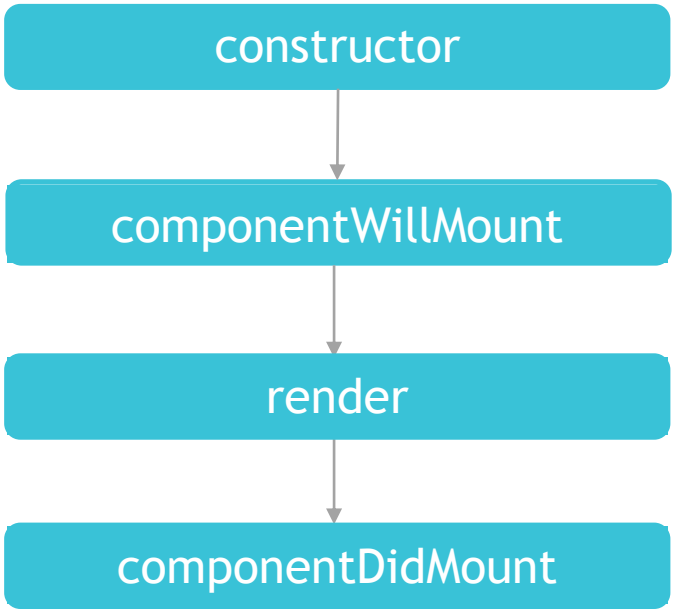
Lifecycle

Components have 3 main phases of their lifecycle and hooks for them

Lifecycle

Mounting

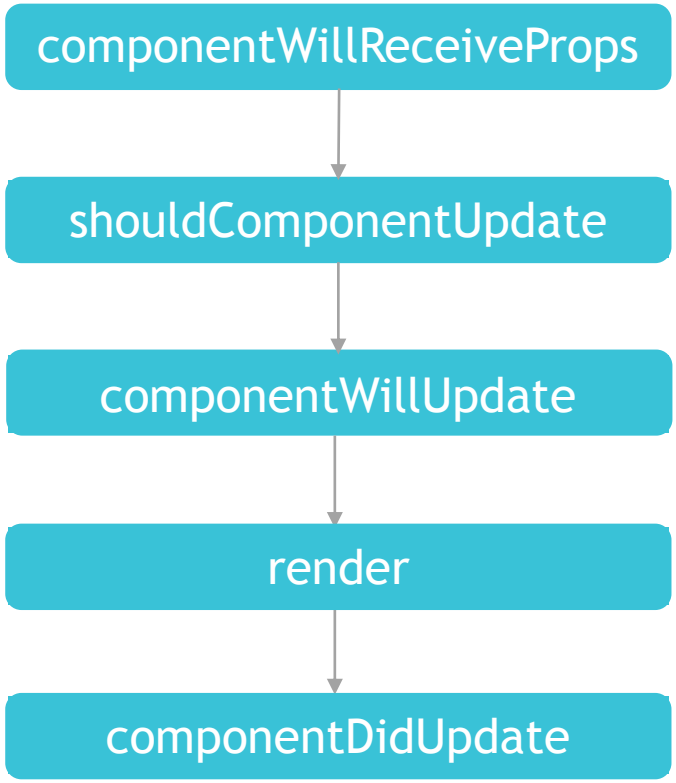
When your component is about to start its life and make its way to the DOM, the following lifecycle methods get called.



Lifecycle

Updating

After your components get added to the DOM, they can potentially update and re-render when a prop or state change occurs. During this time, a different collection of lifecycle methods will get called.



Lifecycle

Unmounting

The last phase when your component is about to be destroyed and removed from the DOM:

componentWillUnmount

Components

Additional way to define components

```
class App extends React.PureComponent {
  render() {
    return (
      <div>
        ... // some content here
      </div>
    )
  }
}
```

Equal to **React.Component** but implements
shouldComponentUpdate
with a shallow prop and state comparison.

Stateful & Stateless

Stateful & Stateless

Main Characteristics:

Stateful Component

- The same as stateless
- + can have and manage own state

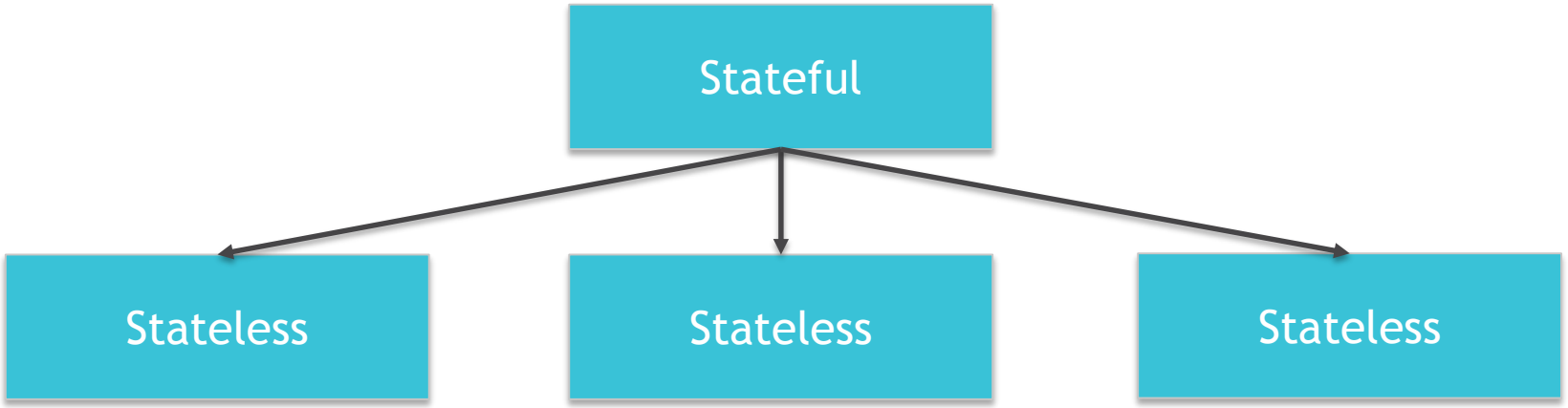
State-only

- Know how everything works
- Just a container for other components
- Provide application data
- Perform data fetching

Stateless Component

- Know how to display received data
- Receive and operate only with props

Stateful & Stateless



Stateful & Stateless

```
class TodoContainer extends React.Component {
  componentDidMount() {
    fetchTodos('/some-url').then((response)=> { this.setState({ list: response }); })
  }

  setFilters(filters="active") {
    this.setState({
      filters: filters
    })
  }

  render() {
    return (
      <div>
        <TodoList list={this.state.list}/>
        <FiltersPane
          filters={this.state.filters}
          setFilters={this.setFilters.bind(this)}/>
      </div>
    )
  }
}

const TodoList = ({list}) => (
  <ul>{list.map((todo)=> <li>{todo.name}</li>)}</ul>
);

const FiltersPane = ({filters, setFilters}) => (
  <button onClick={setFilters}>Set filters</button>
);
```

In general stateless components can be defined as functional components

Stateful & Stateless

Benefits:

- Application become more understandable
- Reduce complexity
- Components (stateless) become reusable
- A lot of simple components
- Easy to test

Synthetic Events

Synthetic Events

What is Synthetic Events?

- Just a normalized/cross-browser wrapper around plain event object
- Have the same interface as the browsers event
- Provide access to the original browser event through **event.nativeEvent**

Code

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
DOMEventTarget target
number timeStamp
string type
```

Synthetic Events

How it looks:

Events are passed into elements as props

```
class Component extends React.Component {
  keyDownHandler(event) {
    console.log('button pressed');
  }

  clickHandler(event) {
    this.setState({ clicked: true });
  }

  render() {
    return (
      <div>
        <input type="text" onKeyDown={this.keyDownHandler}/>

        <button onClick={this.clickHandler.bind(this)}>Click me</button>
      </div>
    );
  }
}
```

React events are named using camelCase.

Handlers are passed as a function rather than a string

Synthetic Events

Can’t Directly Listen to Events on Components

```
function Button() {
  return <button>Click me</button>
}

class Component extends React.Component {
  clickHandler(event) {
    this.setState({ clicked: true });
  }

  render() {
    return (
      <div>
        <Button onClick={this.clickHandler.bind(this)}/>
      </div>
    );
  }
}
```

The reason is because components are wrappers for DOM elements.

Synthetic Events

What if we need to listening regular DOM events?

Don’t do it if you have no strong reason…

Synthetic Events

What if we need to listening regular DOM events?

```
class Component extends React.Component {
  someEventHandler(event) {
    this.setState({ something: true });
  }

  componentDidMount() {
    window.addEventListener("someDOMEvent", this.someEventHandler);
  }

  componentWillUnmount() {
    window.removeEventListener("someDOMEvent", this.someEventHandler);
  }

  render() {
    return (
      <div>
        <button>Click Me</button>
      </div>
    );
  }
}
```

Synthetic Events

Why React and Synthetic Events?

- Browser Compatibility
- Improved Performance

Controlled & Uncontrolled

Controlled & Uncontrolled

With React, you basically get two different ways to deal with forms:

- Standard HTML form elements that can be modified by the user.
- “Controlled” HTML form elements that can only be modified programmatically.

Uncontrolled component

```
class UncontrolledComponent extends React.Component {
  handleSubmitClick() {
    const name = this.input.value;
  }

  render() {
    return (
      <div>
        <input type="text" ref={(input)=> this.input = input}/>
        <button onClick={this.handleSubmitClick}>Sign up</button>
      </div>
    )
  }
}
```

1. **Uncontrolled inputs** are similar to traditional HTML form inputs
2. You have to ‘pull’ the value from the field when you need it
3. Simplest way to implement HTML element

Controlled component

```
class ControlledComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  render() {
    return (
      <div>
        <input type="text" value={this.state.value} onChange={this.handleChange.bind(this)} />
      </div>
    )
  }
}
```

Controlled component

Controlled components benefits

1. Input value will always be this.state.value
2. We can easily control element value and react on state mutation
3. Data (state) and UI (inputs) are always in sync
4. Component can respond to input changes immediately: validations, disable/enable buttons, format inputs etc

Conclusion

For forms which is simple in terms of UI feedback (validation, formating etc) best choice is **Uncontrolled** components with **refs**

A lot of UI feedbacks – your choice is **Controlled** components

Parent-Child relationships

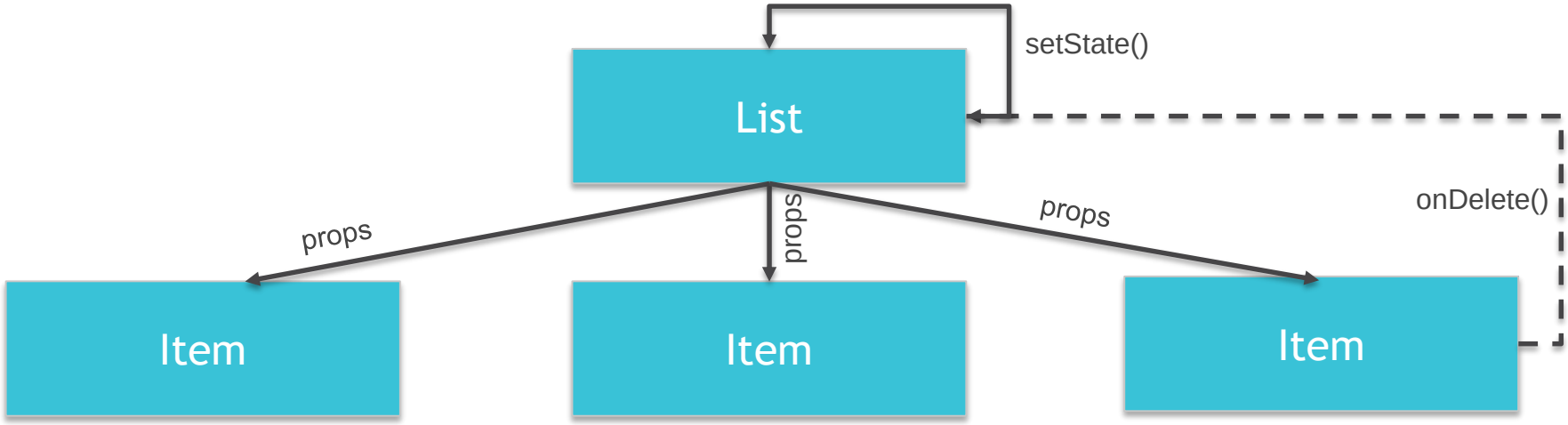
Communication

Best ways how to communicate between parents and children

- State and Props
- State management libraries like Redux, MobX etc

Communication

Communication using props and callbacks from stateful components which help mutate state



Communication

```
class List extends Component {
  constructor() {
    super();
    this.state = { items: [{ name: 'first' }, { name: 'second' }, { name: 'third' }] }
  }

  onDelete(item) {
    this.setState({ items: this.state.items.filter((i)=> i !== item) })
  }

  render() {
    return (
      <ul>
        {this.state.items.map((item)=> <Item key={item.name} item={item} onDelete={this.onDelete.bind(this)}/>)}
      </ul>
    )
  }
}

class Item extends Component {
  render() {
    return (
      <li>
        {this.props.item.name}
        <button onClick={()=> this.props.onDelete(this.props.item)}>Remove</button>
      </li>
    )
  }
}
```

Refs & Context

Refs

This is a special property which allows to refer to the corresponding instance

```
class RefExample extends Component {
  render() {
    return (
      <div>
        <input type="text" ref={(input)=> this.textInput = input}/>
        <ChildComponent ref={(component)=> this.childComponent = component}/>
      </div>
    )
  }
}
```

Refs

This is a special property which allows to refer to the corresponding instance

- Give ability to get access to the DOM node. Could be useful for set focus on input element etc.
- React automatically destroy refs (or set null if you use callbacks) if element was destroyed. So don’t worry about memory leaks.
- Don’t make sense to use ref for functional components because they have no instances
- Allow get access to the child component instance using refs. Method also can be called but it’s not recommended.

Context

Warning:

If you want your application to be stable, don't use context. It is an experimental API and it is likely to break in future releases of React.

Context

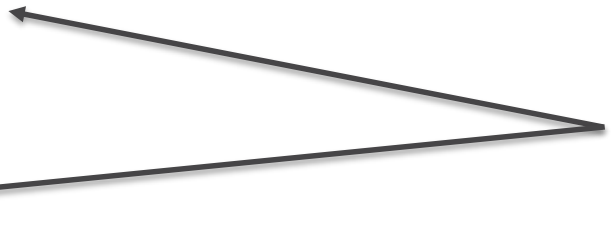
Allow to pass data through the component tree without having to pass the props down manually at every level

```
class Button extends React.Component {
  render() {
    return (<div> {this.context.text} </div>)
  }
}

class Item extends React.Components {
  render() {
    return (
      <div> Item <Button>Click</Button> </div>
    )
  }
}

class List extends React.Component {
  getChildContext() {
    return {
      text: "text"
    };
  }

  render() {
    return (
      <div>
        { this.props.list.map((item)=> <Item />) }
      </div>
    )
  }
}
```



We can ignore step of passing property deeper

Components/Web Components

Components + Web Components

React.Component

Provides a declarative way that keeps the DOM in sync with data.

Web Component

Encapsulated and interoperable custom element that extend HTML itself.

Can be used together?

Components + Web Components

```
var someElement = document.registerElement('some-element', {
  prototype: Object.create(HTMLElement.prototype)
});

class Component extends React.Component {
  componentDidMount() {
    ReactDOM.findDOMNode(this.refs.customElement)
      .addEventListener('some-event', this.doSomething);
  }

  componentWillUnmount() {
    ReactDOM.findDOMNode(this.refs.customElement)
      .removeEventListener('some-event', this.doSomething);
  }

  render() {
    return (
      <some-element ref="customElement"></some-element>
    )
  }
}
```

Events emitted by a Web Component not properly propagate so event have to be attached manually

Need to use refs to access Web Component API

THANK YOU

QUESTIONS...