

## SECTION 4: STATE AND PROPS

# DIFFERENCE BETWEEN STATE AND PROPS

- 🔔 Properties come to the component from parent components.
- 🔔 State is initialized inside the component.
- 🔔 State is internal data-set
- 🔔 Properties are initialization data
- 🔔 Properties can be validated, state – not
- 🔔 Most of your components are not supposed to care about state. Only props!

# DIFFERENCE BETWEEN STATE AND PROPS

## What should go in State?

- ⑩ Data that a component's event handlers may change to trigger a UI update
- ⑩ Minimal possible representation of logic

## What shouldn't Go in State?

- ⑩ Duplicated data from props
- ⑩ React components
- ⑩ Computed data (`this.state.list.length`)

## SET INITIAL STATE

- 🔔 Before we can use state, we need to declare a default set of values for the initial state. This is done by defining a method called `getInitialState()` and returning an object or manually declare state object if you use ES2015
- 🔔 This method tells the component which values should be available from the first render cycle, until the state values are changed. You should never try to use state values without first declaring them in this manner.

# SET INITIAL STATE

```
var InterfaceComponent = React.createClass({
  getInitialState : function() {
    return {
      name : "pavlo",
      job : "developer"
    };
  },
  render : function() {
    return <div>
      My name is {this.state.name} and I am a
      {this.state.job}.
    </div>;
  }
});
ReactDOM.render(
  <InterfaceComponent />,
  document.body
);
```

```
class InterfaceComponent extends Component {
  constructor(){
    super();
    this.state = {
      name:"pavlo",
      job:"developer"
    }
  }
  render() {
    return <div>
      My name is {this.state.name} and I am a {this.state.job}.
    </div>;
  }
}
ReactDOM.render(
  <InterfaceComponent />,
  document.body
);
```

## SET STATE


- 🔔 Setting state should only be done from inside the component. As mentioned, state should be treated as private data, but there are times when you may need to update it.
- 🔔 You cannot set new state values until the component has been mounted. This happens when it's passed (whether directly or by nesting it) to the `React.renderComponent()` method. Then again, why would you need to?

# SET STATE

```
var InterfaceComponent = React.createClass({
  getInitialState : function() {
    return {
      name : "chris"
    };
  },
  handleClick : function() {
    this.setState({
      name : "bob"
    });
  },
  render : function() {
    return <div onClick={this.handleClick}>
      hello {this.state.name}
    </div>;
  }
});
```

```
var InterfaceComponent = React.createClass({
  getInitialState : function() {
    return {
      name : "chris"
    };
  },
  handleClick : function() {
    this.setState({
      name : "bob"
    });
  },
  render : function() {
    return <div onClick={this.handleClick}>
      hello {this.state.name}
    </div>;
  }
});
```

# REACT PROPS

 Props are immutable parameters passed by a parent component to a child sub-component. A component can not alter its #props object (and should not alter the props themselves), the only way for props to change is for a new render to be triggered, where the parent component passes new props to the child.

```
// Parent Component
var LikeList = React.createClass({
  render: function() {
    return (
      <ul><LikeListItem text='turtles.' /></ul>
    );
  }
});
```

```
// Child Component
var LikeListItem = React.createClass({
  render: function() {
    return (
      <li>
        {this.props.text}
      </li>
    );
  }
});
```



# GETDEFAULTPROPS







getDefaultProps will be called to get a default set of props, which will be overridden by the props eventually provided by a parent component. This is useful for optional props which have a sensible default value.

```
// Parent Component
var LikeList = React.createClass({
  render: function() {
    return (
      <ul><LikeListItem /></ul>
    );
  }
});
```

```
// Child Component
var LikeListItem = React.createClass({
  getDefaultProps: function() {
    return {
      text: 'N/A'
    };
  },
  render: function() {
    return (
      <li>{this.props.text}</li>
    );
  }
});
```

# PROPTYPES

-  As your app grows it's helpful to ensure that your components are used correctly. To do that in React you specify propTypes.
-  React.PropTypes exports a range of validators that can be used to make sure the data you receive is valid. When an invalid value is provided for a prop, a warning will be shown in the JavaScript console.
-  Note that for performance reasons propTypes is only checked in development mode.
-  propTypes documents the props expected by a component and defines validators for generating warnings in the dev console.

# PROPTYPES

```
// Child Component (ES5)
var LikeListItem = React.createClass({
  propTypes: {
    text: React.PropTypes.string
  },
  getDefaultProps: function() {
    return {
      text: 'N/A'
    };
  },
  render: function() {
    return (
      <li>{this.props.text}</li>
    );
  }
});
```

```
// Child Component (ES6)
class LikeListItem extends React.Component{
  render() {
    return (
      <li>{this.props.text}</li>
    );
  }
});

LikeListItem propTypes = { text: React.PropTypes.string };
LikeListItem defaultProps = { text: 'N/A' };
```

# PROPTYPES

 Numerous different validators:

 Link to all of them:

<https://goo.gl/QwZ0GO>

```
// Anything that can be rendered: numbers, strings, elements or an array
// (or fragment) containing these types.
```

```
optionalNode: React.PropTypes.node,
```

```
// A React element.
```

```
optionalElement: React.PropTypes.element,
```

```
// You can also declare that a prop is an instance of a class. This uses
// JS's instanceof operator.
```

```
optionalMessage: React.PropTypes.instanceOf(Message),
```

```
// You can ensure that your prop is limited to specific values by treating
// it as an enum.
```

```
optionalEnum: React.PropTypes.oneOf(['News', 'Photos']),
```

```
// An object that could be one of many types
```

```
optionalUnion: React.PropTypes.oneOfType([
  React.PropTypes.string,
  React.PropTypes.number,
  React.PropTypes.instanceOf(Message)
]),
```

# PROPTYPES AND PASSING PROPS

🔔 How to make sure that component has at least one child?

🔔 A common type of React component is one that extends a basic HTML element in a simple way. Often you'll want to copy any HTML attributes passed to your component to the underlying HTML element. To save typing, you can use the JSX spread syntax to achieve this:

```
propTypes: {  
  children: React.PropTypes.element.isRequired  
},
```

```
var CheckLink = React.createClass({  
  render: function() {  
    // This takes any props passed to CheckLink and copies them to <a>  
    return <a {...this.props}>{'√'} {this.props.children}</a>;  
  }  
});  
  
ReactDOM.render(  
  <CheckLink href="/checked.html">  
    Click here!  
  </CheckLink>,  
  document.getElementById('example')  
)
```

## REFS

- 🔔 React supports a very special property that you can attach to any component that is output from `render()`.
- 🔔 This special property allows you to refer to the corresponding backing instance of anything returned from `render()`.
- 🔔 Useful when you want to find DOM Markup, use React-components in not-React application,
- 🔔 It is always guaranteed to be the proper instance, at any point in time.
- 🔔 2 Types: Callback Ref, String Ref

# REFS

## Callback Ref:

```
render: function() {  
  return <TextInput ref={c => this._input = c} />;  
},  
componentDidMount: function() {  
  this._input.focus();  
},
```

 Callback will be executed immediately after the component is mounted

## String Ref

```
render: function() {  
  return <input ref="myInput" />;  
},  
componentDidMount: function() {  
  this.refs.myInput.focus();  
},
```

## REFS

🔔 Never access refs inside of any component's render method - or while any component's render method is even running anywhere in the call stack.

🔔 Refs may not be attached to a stateless function

🔔 You can create public methods on components and call them like:

```
this.refs.myTypeahead.reset()
```

🔔 Refs are automatically managed for you! If that child is destroyed, its ref is also destroyed for you




# CHILDREN

- 🔔 There may be situations where you want components to wrap provided components rather than generate those from props:
- 🔔 Children are placed between a component's opening and ending tags, like regular HTML elements. The wrapper component can access those children via `this.props.children`.

```
var Likes = React.createClass({
  render: function() {
    return (
      <div>
        <LikesListWrapper>
          <LikeListItem text="turtles" />
        </LikesListWrapper>
      </div>
    );
  }
});

var LikesListWrapper = React.createClass({
  render: function() {
    return (
      <ul>{ this.props.children }</ul>
    );
  }
});
```

## CLASSNAME

 Because class is a reserved JavaScript keyword, to set the class of an element you will need to use the className property name instead.

```
render: function() {  
  return (  
    <a className="btn btn-default">I am a  
    button with a class!</a>  
  );  
}
```