**Stateless Components Inherit From Stateful Components**

Let’s learn our first *programming pattern!*

In this lesson, we’ll take a look at a simple version of a programming pattern. The following lessons will expand upon that lesson, and by the end, we’ll have a programming pattern in its full complexity.

Our programming pattern uses two React components: a *stateful* component, and a *stateless* component. “Stateful” describes any component that has a state property; “stateless” describes any component that does not.

In our pattern, a *stateful* component passes its state down to a *stateless* component.

**Build a Stateful Component Class**

Let’s make a *stateful* component pass its state to a *stateless* component.

To make that happen, you need two component classes: a *stateful* class, and a *stateless* class.

**Instructions**

**1.**

We’ll build the stateful class first.

On line 1 of **Parent.js**, import the React.js library. Store the library in a variable named React.

On line 2, import the ReactDOM library. Store it in a variable named ReactDOM.

Make lines 3 and 4 empty. You’ll add code to line 3 later.

On line 5, declare a new Parent component. Parent will represent your *stateful* component class.

Ensure Parent extends React.Component. Add this method to your Parent component class:

render() {

return <div></div>;

}

**2.**

Since Parent is supposed to be *stateful*, it will need to set its initial state. That means that it will need a constructor method.

Before the render method, give Parent a method named constructor. Give constructor one parameter named props.

Inside of constructor()‘s body, call super(props). On the next line, still inside of constructor()‘s body, declare a property named this.state set equal to { name: 'Frarthur' }.

import React from 'react'

import ReactDOM from 'react-dom'

class Parent extends React.Component{

  constructor(props){

    super(props);

    this.state = {name: 'Frarthur'}

  }

  render() {

  return <div></div>;

}

}

**Build a Stateless Component Class**

Great! You just made a *stateful* component class named Parent.

Now, let’s make our *stateless* component class.

**Instructions**

**1.**

Select **Child.js**.

On line 1, import the React.js library. Store the library in a variable named React.

Leave line 2 blank. On line 3, declare a new component named Child. Child will represent your *stateless* component class.

Add the following method to your Child component class:

render() {

return <h1></h1>;

}

**2.**

Child is going to receive a prop called name, and display that prop on the screen.

How can you make a component *display* a prop called name?

* To access a prop, use the expression this.props.name-of-prop.
* To make a component *display* something, include that thing in the render function’s return statement.

You need to include this.props.name inside of Child‘s render function’s return statement.

Add this expression in between the <h1></h1> tags:

Hey, my name is {this.props.name}!

**3.**

A <Parent /> is going to pass a prop to a <Child />.

That means that a <Parent /> is going to *render* a <Child />. Rendering is the only way for a component to pass props to another component.

Any component rendered by a different component must be included in an export statement.

On line 3, put the word export before the word class, so that the line begins: export class Child.

**4.**

That’s it! Child is ready to inherit a prop and display it.

import React from 'react'

export class Child extends React.Component{

  render() {

  return <h1>Hey, my name is {this.props.name}!</h1>;

  }

}

# Pass a Component's State

A <Parent /> is supposed to pass its state to a <Child />.

Before a <Parent /> can pass anything to a <Child />, you need to import Child into **Parent.js**.

**Instructions**

**1.**

To import a local component, we will need to modify our import syntax to use local files and named exports. For example, if we wanted to import a component called ComponentName from a local file called **Component.js** we would write

import { ComponentName } from './Component'

On line 3, import the Child component from **Child.js**.

**Parent.js** and **Child.js** share the same parent directory.

**2.**

Great! Now Parent is ready to pass its state to a <Child />.

Inside of Parent‘s .render() method’s return statement, get rid of the <div></div>.

Replace it with a <Child /> instance.

Give <Child /> an attribute with a name of name. The attribute’s value should be the name property stored in this.state.

**3.**

All that’s left is to render your components!

At the bottom of **Parent.js**, call ReactDOM.render();.

For ReactDOM.render()‘s first argument, pass in <Parent />.

For ReactDOM.render‘s second argument, pass in document.getElementById('app').

Rendering <Parent /> will render both components, because Parent‘s render function returns a <Child />. Click Run, and see the rendered information that you passed down from Parent.

import React from 'react'

import ReactDOM from 'react-dom'

import { Child } from './Child'

class Parent extends React.Component{

  constructor(props){

    super(props);

    this.state = {name: 'Frarthur'}

  }

  render() {

  return <Child name={this.state.name}/>;

}

}

ReactDOM.render(<Parent />, document.getElementById('app'));

# Don't Update props

Great work! You just passed information from a stateful component to a stateless component. You will be doing a lot of that.

You learned earlier that a component can change its state by calling this.setState(). You may have been wondering: how does a component change its props?

The answer: it doesn’t!

A component should never update this.props. Look at **Bad.js** to see an example of what not to do.

This is potentially confusing. props and state store dynamic information. Dynamic information can change, by definition. If a component can’t change its props, then what are props for?

**A React component should use props to store information that can be changed, but can only be changed by a different component.**

**A React component should use state to store information that the component itself can change.**

If that’s a bit confusing, don’t worry! The next two lessons will be examples.

# Child Components Update Their Parents' state

In the last lesson, you passed information from a stateful, parent component to a stateless, child component.

In this lesson, you’ll be expanding on that pattern. The stateless, child component will update the state of the parent component.

Here’s how that works:

1

The parent component class defines a method that calls this.setState().

For an example, look in **Step1.js** at the .handleClick() method.

2

The parent component binds the newly-defined method to the current instance of the component in its constructor. This ensures that when we pass the method to the child component, it will still update the parent component.

For an example, look in **Step2.js** at the end of the constructor() method.

3

Once the parent has defined a method that updates its state and bound to it, the parent then passes that method down to a child.

Look in **Step2.js**, at the prop on line 28.

4

The child receives the passed-down function, and uses it as an event handler.

Look in **Step3.js**. When a user clicks on the <button></button>, a click event will fire. This will make the passed-down function get called, which will update the parent’s state.

Click through the three files in order, and try to follow their chronology. Whenever you’re ready, click Next and we’ll build an example!

# Define an Event Handler

To make a child component update its parent’s state, the first step is something that you’ve seen before: you must define a state-changing method on the parent.

**Instructions**

**1.**

Select **Child.js**.

Look at Child‘s render function. It’s similar to the last lesson, but you can see a <select> dropdown menu that wasn’t there before.

Click Run. Try selecting different names from the dropdown menu in the browser.

It doesn’t work! When you select a name, that name is supposed to replace “Frarthur” on the screen.

Look at line 8. Notice that the name inside of the <h1></h1> is equal to this.props.name. In order to make the dropdown menu change the <h1></h1>, you will need the dropdown menu to change the value of this.props.name!

**2.**

How can you change Child‘s this.props.name?

Open **Parent.js** and look at line 13.

Parent renders a <Child />, passing in a name prop. This name prop is the same value that the <Child /> displays in its <h1></h1>.

You need Child‘s dropdown menu to change Parent‘s this.state.name! That will cause <Child /> to get passed a new name prop, which will change the name displayed on the screen.

In **Parent.js**, define a new function that can change this.state.name:

changeName(newName) {

this.setState({

name: newName

});

}

render() { ...

# Pass the Event Handler

In the last exercise, you defined a function in Parent that can change Parent‘s state.

Parent must pass this function down to Child, so that Child can use it in an event listener on the dropdown menu.

**Instructions**

**1.**

We now need to make sure that the .changeName() method will always refer to the instance of Parent, even when we pass it down to Child to use.

In the constructor method of Parent, bind this.changeName to the current value of this and store it in this.changeName.

Hint

The generic syntax for binding a method in the constructor is:

this.methodName = this.methodName.bind(this);

**2.**

Pass .changeName() down to Child!

In **Parent.js**, inside of Parent‘s render function, add a second attribute to <Child />. Give this attribute a name of onChange, and a value of the changeName method.

class Parent extends React.Component {

  constructor(props) {

    super(props);

    this.state = { name: 'Frarthur' };

    this.changeName = this.changeName.bind(this);

  }

  changeName(newName) {

    this.setState({

      name: newName

    });

  }

  render() {

    return <Child name={this.state.name} onChange={this.changeName} />

  }

}

‘s render function, give <select> a new attribute.

Make the attribute’s *name* onChange. This will create an *event listener*.

Make the attribute’s *value* equal to the function that you passed in from **Parent.js**. If you can’t remember what prop name you used, then check Parent‘s render function.

**2.**

Try selecting a new name from the dropdown menu.

It doesn’t work! Why not? Take another look at this.props.onChange:

changeName(newName) {

this.setState({

name: newName

});

}

This function needs to be passed a new name as an argument, in order to work properly.

When a user selects a new dropdown item, it will invoke changeName, but it *won’t* pass the correct argument! Instead of passing a new name, it will pass an *event object*, as all event listeners do.

This is a common problem when passing down an *event handler* in React! The solution is to define another function.

This new function should take an *event object* as an argument, extract the name that you want from that event object, and then call the *event handler*, passing in the extracted name! It sounds like a lot, but you will see this happen so often that it will soon feel intuitive.

In *Child.js*, before the render function, define a new function that can be passed an *event object:*

handleChange(e) {

const name = e.target.value;

this.props.onChange(name);

}

render() { ...

**3.**

You’ve defined a new function that can take an *event object*, and use it to correctly update the parent’s state. Now let’s put that new function to use!

As with all methods that we pass in React, we must first bind this to our new method to the current instance of Child.

Add the following boilerplate constructor to your Child class:

constructor(props) {

super(props);

}

Then at the end of your constructor bind .handleChange() to the current value of this.

**4.**

In Child‘s render function, change the *event handler* from {this.props.onChange}, to {this.handleChange}.

Click Run, and try selecting some different names from the dropdown menu. It should work!

import React from 'react';

export class Child extends React.Component {

  constructor(props) {

    super(props);

    this.handleChange = this.handleChange.bind(this);

  }

  handleChange(e) {

    const name = e.target.value;

    this.props.onChange(name);

  }

  render() {

    return (

      <div>

        <h1>

          Hey my name is {this.props.name}!

        </h1>

        <select id="great-names" onChange={this.handleChange}>

          <option value="Frarthur">

            Frarthur

          </option>

          <option value="Gromulus">

            Gromulus

          </option>

          <option value="Thinkpiece">

            Thinkpiece

          </option>

        </select>

      </div>

    );

  }

}

import React from 'react';

import ReactDOM from 'react-dom';

import { Child } from './Child';

class Parent extends React.Component {

  constructor(props) {

    super(props);

    this.state = { name: 'Frarthur' };

    this.changeName = this.changeName.bind(this);

  }

  changeName(newName) {

    this.setState({

      name: newName

    });

  }

  render() {

    return <Child name={this.state.name} onChange={this.changeName} />

  }

}

ReactDOM.render(

  <Parent />,

  document.getElementById('app')

);

# Automatic Binding

Great work! Stateless components updating their parents’ state is a React pattern that you’ll see more and more. Learning to recognize it will help you understand how React apps are organized.

# Child Components Update Sibling Components

Patterns within patterns within patterns!

In lesson 1, you learned your first React programming pattern: a stateful, parent component passes down a prop to a stateless, child component.

In lesson 2, you learned that lesson 1’s pattern is actually part of a larger pattern: a stateful, parent component passes down an event handler to a stateless, child component. The child component then uses that event handler to update its parent’s state.

In this lesson, we will expand the pattern one last time. A child component updates its parent’s state, and the parent passes that state to a sibling component.

An understanding of this final pattern will be very helpful in the wild, not to mention in the next React course. Click Next and we’ll build an example!

# One Sibling to Display, Another to Change

One of the very first things that you learned about components is that they should only have one job.

In the last lesson, Child had two jobs:

1 - Child displayed a name.

2 - Child offered a way to change that name.

You should make like Solomon and divide Child in two: one component for displaying the name, and a different component for allowing a user to change the name.

In the code editor, select **Child.js**. Notice that these lines have vanished:

<h1>

Hey, my name is {this.props.name}!

</h1>

The new version of Child renders a dropdown menu for changing the name, and that’s it.

Select **Sibling.js** in the code editor.

Read through the render function’s return statement.

Here, the name is displayed! Or at least it will be displayed, once you’ve done a little editing.

That brings us to the essential new concept for this lesson: you will have one stateless component display information, and a different stateless component offer the ability to change that information.

**Instructions**

**1.**

Click Run.

Select **Parent.js**, and look at the rendered <Parent /> in the browser. Try selecting a name from the dropdown menu. Does it work?

**Pass the Right props to the Right Siblings**

Look at **Parent.js** in the code editor.

Three things have changed in this file since the last Lesson:

1. Sibling has been required on line 4.
2. A <Sibling /> instance has been added to the render function on line 27.
3. <Sibling /> and <Child /> have been wrapped in a <div></div>, since JSX expressions must have only one outer element.

**Instructions**

**1.**

Sibling‘s job is to display the selected name.

That name is *stored* in Parent‘s state. Parent will have to pass the name to Sibling, so that Sibling can display it.

In Parent‘s render function, pass the name to <Sibling />:

name={this.state.name}

**2.**

Child‘s job is to offer a way to *change* the chosen name. *Not* to display it!

In **Parent.js**, look at lines 25 and 26.

You are passing Child two pieces of dynamic information:

1. The chosen name (this.state.name).
2. A way to *change* the chosen name (this.changeName).

One of those is now unnecessary!

Remove an attribute from <Child />, so that it receives a way to *change* the chosen name and nothing more.

# Stateless Components Inherit From Stateful Components Recap

You just executed your first complete React programming pattern!

Let’s review. Follow each step in the code editor:

* A stateful component class defines a function that calls this.setState. (**Parent.js**, lines 15-19)
* The stateful component passes that function down to a stateless component. (**Parent.js**, line 24)
* That stateless component class defines a function that calls the passed-down function, and that can take an event object as an argument. (**Child.js**, lines 10-13)
* The stateless component class uses this new function as an event handler. (**Child.js**, line 20)
* When an event is detected, the parent’s state updates. (A user selects a new dropdown menu item)
* The stateful component class passes down its state, distinct from the ability to change its state, to a different stateless component. (**Parent.js**, line 25)
* That stateless component class receives the state and displays it. (**Sibling.js**, lines 5-10)
* An instance of the stateful component class is rendered. One stateless child component displays the state, and a different stateless child component displays a way to change the state. (**Parent.js**, lines 23-26)

This pattern occurs in React all the time! The more that you see it, the more that its elegance will become clear.

Being introduced to this pattern is your first step towards understanding how React apps fit together! You’ll get more practice using it throughout this course, as well as in the course after this one.