**REACTJS**

**PART 2**

**State & Lifecycle**

Function Clock(props) {

Return(

<div>

<h1>Hello, World</h1>

<h2>It is {props.date.toLocaleTimeString()}.</h2>

</div>

);

};

Function tick(){

ReactDOM.render(

<Clock date={new Date()} />,

Document.getElementById(‘root’)

);

}

setInterval(tick, 1000);

In this section we will learn how to make the clock component truly reusable and encapsulated. It will set up its own timer and update itself every second. The example above misses a crucial requirement: the fact that the clock sets up a timer and updates the UI every second should be an implementation detail of the clock. Ideally we want to write this once and the clock update itself:

ReactDOM.render(<Clock />, document.getElementById(‘root’));

To implement this, we need to add “state” to the Clock component. State is similar to props, but it is private and fully controlled by the component. Local state is a feature only available to classes (not functions).

Converting a function to a class

1. Create an ES6 class with the same name, that extends React.Component
2. Add a single empty method to it called render()
3. Move the body of the function in to the render() method
4. Replace props with *this.props* in the render() body
5. Delete the remaining empty function decleration

Example:

Class Clock extends React.Component{

Render(){

Return(

<div>

<h1>Hello, World!</h1>

<h2>It is {this.props.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

This lets us use additional features such as local state and lifecycle hooks.

Adding local state to a class

//we will move the date from *props* to *state* in 3 steps:

1. Replace *this.props.date* with *this.state.date* in the render method
2. Add a class constructor that assigns the initial *this.state*:

Class Clock extends React.Component{

Constructor(props){

Super(props);

This.state = {date: new Date()};

}

Render(){

Return(

<div>

<h1>Hello, World!</h1>

<h2>it is, {this.state.date.toLocalTimeString()}.</h2>

</div>

);

}

}

//note how we pass *props* to the base constructor. Class components should always call the base constructor with *props*.

1. Remove the date from the <Clock /> element.

//next we will make the clock set up its own timer and update itself every second

Adding lifecycle methods to a class

In applications with many components, its very important to free up resources taken by the components when they are destroyed. We want to setup a timer whenever the Clock is rendered to the DOM for the first time. This is called *mounting* in React.

We also want to clear that timer whenever the DOM produced by the Clock is removed. This is called *unmounting.*

We can declare special methods in the component class to run some code when a component mounts and unmounts:

componentDidMount()

componentWillUnmount()

//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 timerID right on *this*.

\*note – 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);

}

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

Tick() {

This.setState({date: new Date()});

}

**Handling events**

React events are named using camelCase. With JSX you pass a function as the event handler, rather than a string:

<button onClick = {activateLasers}>Activate</button>

When you define a component using an ES6 class, a common pattern is for an event handler to be a method on the class.

\*Make sure to bind methods in the constructor to reference *this*, otherwise *this* will be undefined – as the class is rendered in the DOM, *this* will not point to the class in which it was created\*

Constructor(props){

Super(props);

This.state = {isToggleOn: true};

This.handleClick = this.handlieClick.bind(this);;

}

handleClick(){

this.setState(prevState => ({

isToggleOn: !prevState.isToggleOn

}));

}

Render() {

Return (

<button onClick={this.handleClick}> {this.state.isToggleOn ? “On” : “Off” } </button>

);

}

**Conditional rendering**

In React, you can create distinct components that encapsulate behaviour you need. Then, you can render only some of them, depending on the state of your application. Conditional rendering in React works the same way conditions do in Javascript.

Example 1

Class UserGreeting extends React.Component{

Render(){

Return <h1>Welcome Back!</h1>

}

}

Class GuestGreeting extends React.Component {

Render() {

Return <h1>Please sign up</h1>

}

}

Class Greeting extends React.Component {

Render() {

Const isLoggedIn = this.props.isLoggedIn;

If(isLoggedIn) {

Return <UserGreeting />;

}else{

Return <GuestGreeting />;

}

}

}

ReactDOM.render(<Greeting isLoggedIn = {true} />, document.getElementById(‘root’));

Example 2:

Render() {

Const isLoggedIn = this.state.isLoggedIn;

Return (

<div>

The user is {isLoggedIn ? “Currently” : “not” } Logged in.

</div>

);

}

**Lists & Keys**

In JS

Const numbers = [1,2,3,4,5];

Const doubled – numbers.map((number) => number \* 2);

Console.log(doubled);

In React

Const numbers = [1,2,3,4,5];

Const listItems = numbers.map((number) => <li>{number}</li>);

//we include the entire listItems array inside a <ul> element and render it to the DOM:

ReactDOM.render(<ul>{listItems}</ul>, document.getElementById(‘root’));

//usually you would render lists inside a component:

Class NumberList extends React.Component{

Render() {

Const numbers = this.props.numbers;

Const listItems = numbers.map((n) =>

<li>{n}</li>;

Return (

<ul>{listItems}</ul>

);

}

};

Const numbers = [1,2,3,4,5];

ReactDOM.render(<NumberList numbers = {numbers} />, document.getElementById(‘root’));

When you run this code, you’ll be given a warning that a key should be provided for list items. A “key” is a special string attribute you need to include when creating lists of elements. To assign a key:

Const listItems = numbers.map((n) =>

<li key={n.toString()}> {n} </li>

);

Keys help React identify which items have changed, are added, or are removed. A good rule of thumb is that elements inside the map() call need keys.

Keys used within arrays should be unique among their siblings. However, they don’t need to be globally unique.

Class Blog extends React.Component{

Render() {

Const sidebar = (

<ul>

{this.props.posts.map((post) =>

<li key={post.id}> {post.title} </li>

)}

</ul>

);

Const content = this.props.posts.map((post) =>

<div key = {post.id}>

<h3>{post.title}</h3>

<p>{post.content}</p>

</div>

);

Return (

<div>

{sidebar}

<hr />

{content}

</div>

);

}

}

Const.posts = [

{id: 1, title: “Hello World”, content: “Welcome”},

{id: 2, title: “Install”, content: “You can install…”}

];

ReactDOM.render(<Blog posts={posts} />, document.getElementById(‘root’));