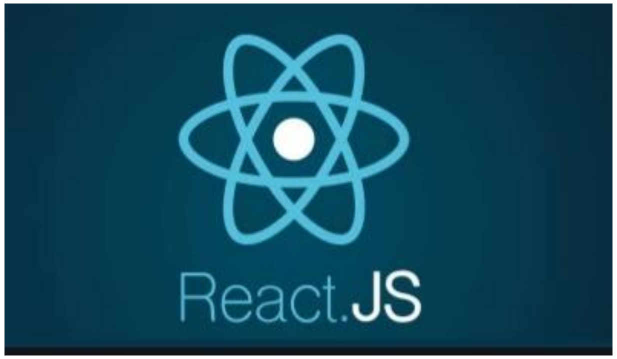

Presentation by Uplatz

Contact Us: https://training.uplatz.com/

Email: info@uplatz.com

Phone:+44 7836 212635

Uplatz

Table Of Contents:

React.createClass vs extends React.Component

Create React Component

- ➢ Let's explore the syntax differences by comparing two code examples.

React.createClass (deprecated)

- ➢ Here we have a const with a React class assigned, with the render function following on to complete a typical base component definition.

```
import React from 'react';
const MyComponent = React.createClass({
 render() {
 return (
 <div></div>
 );
}
});
export default MyComponent;
```

React.Component

- Let's take the above React.createClass definition and convert it to use an ES6 class.

```
import React from 'react';
class MyComponent extends React.Component {
 render() {
 return (
 <div></div>
 );
 }
}
export default MyComponent;
```

- In this example we're now using ES6 classes.
- For the React changes, we now create a class called **MyComponent** and extend from React.Component instead of accessing React.createClass directly.
- This way, we use less React boilerplate and more JavaScript.
- PS: Typically this would be used with something like Babel to compile the ES6 to ES5 to work in other browsers.

"this" Context

- Using React.createClass will automatically bind this context (values) correctly, but that is not the case when using ES6 classes.

React.createClass

- Note the onClick declaration with the this.handleClick method bound.
- When this method gets called React will apply the right execution context to the handleClick

```
import React from 'react';
const MyComponent = React.createClass({
 handleClick() {
 console.log(this); // the React Component instance
 },
 render() {
 return (
 <div onClick={this.handleClick}></div>
 );
 }
});
```

export default MyComponent;

React.Component

With ES6 classes this is null by default, properties of the class do not automatically bind to the React class (component) instance.

```
import React from 'react';
class MyComponent extends React.Component {
 constructor(props) {
 super(props);
 }
 handleClick() {
 console.log(this); // null
 }
 render() {
```

```
 return (
 <div onClick={this.handleClick}></div>
 );
 }
}
export default MyComponent;
```

There are a few ways we could bind the right this context.

Case 1: Bind inline:

```
import React from 'react';
class MyComponent extends React.Component {
 constructor(props) {
 super(props);
 }
 handleClick() {
```

```
console.log(this); // the React Component instance
 }
 render() {
 return (
 <div onClick={this.handleClick.bind(this)}></div>
 );
 }
}
export default MyComponent;
```

- There are a few ways we could bind the right this context.

Case 1: Bind inline:

```
import React from 'react';
class MyComponent extends React.Component {
```

```
constructor(props) {
 super(props);
 }
 handleClick() {
 console.log(this); // the React Component instance
 }
 render() {
 return (
 <div onClick={this.handleClick.bind(this)}></div>
 );
 }
}
export default MyComponent;
```

Case 2: Bind in the class constructor

- Another approach is changing the context of this.handleClick inside the constructor. This way we avoid inline repetition.
- Considered by many as a better approach that avoids touching JSX at all:

```
import React from 'react';
class MyComponent extends React.Component {
 constructor(props) {
 super(props);
 this.handleClick = this.handleClick.bind(this);
 }
 handleClick() {
 console.log(this); // the React Component instance
 }
render() {
 return (
 <div onClick={this.handleClick}></div>
 );
 }
}
export default MyComponent;
```

Case 3: Use ES6 anonymous function

- You can also use ES6 anonymous function without having to bind explicitly:

```
import React from 'react';
class MyComponent extends React.Component {
 constructor(props) {
 super(props);
 }
handleClick = () => {
 console.log(this); // the React Component instance
 }
 render() {
 return (
 <div onClick={this.handleClick}></div>
 );
 }
}
export default MyComponent;
```

Declare Default Props and PropTypes

- There are important changes in how we use and declare default props and their types.

React.createClass

- In this version, the propTypes property is an Object in which we can declare the type for each prop.
- The getDefaultProps property is a function that returns an Object to create the initial props.

```
import React from 'react';
const MyComponent = React.createClass({
propTypes: {
 name: React.PropTypes.string,
 position: React.PropTypes.number
 },
 getDefaultProps() {
 return {
```

```
name: 'Home',
 position: 1
 };
 },
 render() {
 return (
 <div></div>
 );
 }
});
export default MyComponent;
```

React.Component

- This version uses propTypes as a property on the actual MyComponent class instead of a property as part of the createClass definition Object.

- The getDefaultProps has now changed to just an Object property on the class called defaultProps, as it's no longer a "get" function, it's just an Object.
- It avoids more React boilerplate, this is just plain JavaScript.

```
import React from 'react';
class MyComponent extends React.Component {
 constructor(props) {
 super(props);
 }
 render() {
 return (
 <div></div>
 );
 }
}
MyComponent.propTypes = {
 name: React.PropTypes.string,
 position: React.PropTypes.number
};
MyComponent.defaultProps = {
 name: 'Home',
 position: 1
};
export default MyComponent;
```

- Additionally, there is another syntax for propTypes and defaultProps.
- This is a shortcut if your build has ES7 property initializers turned on:

```
import React from 'react';
class MyComponent extends React.Component {
 static propTypes = {
 name: React.PropTypes.string,
 position: React.PropTypes.number
 };
 static defaultProps = {
 name: 'Home',
 position: 1
 };
 constructor(props) {
 super(props);
 }
render() {
 return (
 <div></div>
 );
 }
}
export default MyComponent;
```

Mixins

- We can use mixins only with the React.createClass way.

React.createClass

- In this version we can add mixins to components using the mixins property which takes an Array of available mixins.

- These then extend the component class.

```
import React from 'react';
var MyMixin = {
 doSomething() {
 }
};
const MyComponent = React.createClass({
 mixins: [MyMixin],
 handleClick() {
 this.doSomething(); // invoke mixin's method
 },
 render() {
 return (
 <button onClick={this.handleClick}>Do
Something</button>
 );
 }
});
export default MyComponent;
```

React.Component

- React mixins are not supported when using React components written in ES6.
- Moreover, they will not have support for ES6 classes in React.
- The reason is that they are considered harmful.

Set Initial State

- There are changes in how we are setting the initial states.

React.createClass

- We have a getInitialState function, which simply returns an Object of initial states

```
import React from 'react';
const MyComponent = React.createClass({
 getInitialState () {
 return {
 activePage: 1
 };
 },
 render() {
 return (
 <div></div>
 );
 } });
```

export default MyComponent;

React.Component

- In this version we declare all state as a simple initialisation property in the constructor, instead of using the getInitialState function.
- It feels less "React API" driven since this is just plain JavaScript.

```
import React from 'react';
class MyComponent extends React.Component {
 constructor(props) {
 super(props);
 this.state = {
 activePage: 1
 };
 }
render() {
 return (
 <div></div>
 );
 }
}
export default MyComponent;
```

ES6/React “this” keyword with ajax to get data from server

```
import React from 'react';
class SearchEs6 extends React.Component{
 constructor(props) {
 super(props);
 this.state = {
```

```
searchResults: []
 };
 }
 showResults(response){
 this.setState({
 searchResults: response.results
 })
 }
search(url){
 $.ajax({
 type: "GET",
 dataType: 'jsonp',
 url: url,
success: (data) => {
 this.showResults(data);
 },
 error: (xhr, status, err) => {
 console.error(url, status, err.toString());
 }
 });
 }
 render() {
 return (
 <div>
 <SearchBox search={this.search.bind(this)} />
 <Results searchResults={this.state.searchResults} />
 </div>
 );} }
```

React AJAX call

HTTP GET request

- Sometimes a component needs to render some data from a remote endpoint (e.g. a REST API).
- A standard practice is to make such calls in componentDidMount method.

Here is an example, using superagent as AJAX helper:

```
import React from 'react'
import request from 'superagent'
class App extends React.Component {
 constructor () {
 super()
 this.state = {}
 }
 componentDidMount () {
```

```
request
.get('/search')
.query({ query: 'Manny' })
.query({ range: '1..5' })
.query({ order: 'desc' })
.set('API-Key', 'foobar')
.set('Accept', 'application/json')
.end((err, resp) => {
if (!err) {
this.setState({someData: resp.text})
}
})
},
render() {
```

```
return (
 <div>{this.state.someData || 'waiting for
response...'}</div>
 )
 }
}
React.render(<App />,
document.getElementById('root'))
```

- A request can be initiated by invoking the appropriate method on the request object, then calling .end() to send the request.
- Setting header fields is simple, invoke .set() with a field name and value.
- The .query() method accepts objects, which when used with the GET method will form a query-string. The following will produce the path

- **/search?query=Manny&range=1..5&order=desc.**

POST requests

```
request.post('/user')
 .set('Content-Type', 'application/json')
 .send('{"name":"tj","pet":"tobi"}')
 .end(callback)
```

HTTP GET request and looping through data

- The following example shows how a set of data obtained from a remote source can be rendered into a component.
- We make an AJAX request using fetch, which is build into most browsers.
- **Use a fetch polyfill in production to support older browsers.**
-

- You can also use any other library for making requests (e.g. axios, SuperAgent, or even

plain Javascript).

- We set the data we receive as component state, so we can access it inside the render method.
- There, we loop through the data using map. Don't forget to always add a unique key attribute (or prop) to the looped element, which is important for React's rendering performance.

```
import React from 'react';
class Users extends React.Component {
 constructor() {
 super();
 this.state = { users: [] };
 }
```

```
componentDidMount() {
 fetch('/api/users')
 .then(response => response.json())
 .then(json => this.setState({ users: json.data }));
 }
 render() {
 return (
 <div>
 <h1>Users</h1>
 {
 this.state.users.length == 0
 ? 'Loading users...'
 : this.state.users.map(user => (
 <figure key={user.id}>
<img src={user.avatar} />
 <figcaption>
 {user.name}
 </figcaption>
 </figure>
 ))
 }
 </div>
 );
 }
}
ReactDOM.render(<Users />,
document.getElementById('root'));
```

➢ Ajax in React without a third party library - a.k.a with VanillaJS

The following would work in IE9+

```
import React from 'react‘
class App extends React.Component {
 constructor () {
 super()
 this.state = {someData: null}
 }
 componentDidMount () {
 var request = new XMLHttpRequest();
 request.open('GET', '/my/url', true);
request.onload = () => {
 if (request.status >= 200 && request.status < 400)
// Success!
 this.setState({someData: request.responseText})
 } else {
 // We reached our target server, but it returned an
error
 // Possibly handle the error by changing your state.
 }
 };
request.onerror = () => {
 // There was a connection error of some sort.
 // Possibly handle the error by changing your state.
 };
request.send();
 },
 render() {
 return (
 <div>{this.state.someData || 'waiting for
response...'}</div>
 )
 }
}
React.render(<App />,
document.getElementById('root'))
```

Communication Between Components

Child to Parent Components

Sending data back to the parent, to do this we simply pass a function as a prop from the parent component

- to the child component, and the child component calls that function.
- In this example, we will change the Parent state by passing a function to the Child component and invoking that function inside the Child component.

```
import React from 'react';
class Parent extends React.Component {
 constructor(props) {
 super(props);
 this.state = { count: 0 };
 this.outputEvent = this.outputEvent.bind(this);
 }
 outputEvent(event) {
 // the event context comes from the Child
 this.setState({ count: this.state.count++ });
```

```
}
 render() {
 const variable = 5;
 return (
 <div>
 Count: { this.state.count }
 <Child clickHandler={this.outputEvent} />
 </div>
 );
 }
}
class Child extends React.Component {
 render() {
 return (
<button onClick={this.props.clickHandler}>
 Add One More
 </button>
 );
 }
}
export default Parent;
```

- Note that the Parent's outputEvent method (that changes the Parent state) is invoked by the Child's button onClick event.

Not-related Components

- The only way if your components does not have a parent-child relationship (or are related but too further such as a grand grand grand son) is to have some kind of a signal that one component subscribes to, and the other writes into.

- Those are the 2 basic operations of any event system: subscribe/listen to an event to be notify, and send/trigger/publish/dispatch a event to notify the ones who wants.
- There are at least 3 patterns to do that. You can find a comparison here.

Here is a brief summary:

- **Pattern 1:** Event Emitter/Target/Dispatcher: the listeners need to reference the source to subscribe.
- **to subscribe: otherObject.addEventListener('click', () => { alert('click!'); });**
- **to dispatch: this.dispatchEvent('click');**
- **Pattern 2:** Publish/Subscribe: you don't need a specific reference to the source that triggers the event, there is a global object accessible everywhere that handles all the events.

- to subscribe: globalBroadcaster.subscribe('click', () => { alert('click!'); });
- to dispatch: globalBroadcaster.publish('click');
- **Pattern 3:** Signals: similar to Event Emitter/Target/Dispatcher but you don't use any random strings here.
- Each object that could emit events needs to have a specific property with that name.
- This way, you know exactly what events can an object emit.

to subscribe: otherObject.clicked.add(() => { alert('click'); });

to dispatch: this.clicked.dispatch();

Parent to Child Components

- That the easiest case actually, very natural in the React world and the chances are - you are already using it.
- You can pass props down to child components.
- In this example message is the prop that we pass down to the child component, the name message is chosen arbitrarily, you can name it anything you want.

```
import React from 'react';
class Parent extends React.Component {
 render() {
 const variable = 5;
 return (
 <div>
 <Child message="message for child" />
<Child message={variable} />
 </div>
 );
 }
}
class Child extends React.Component {
 render() {
 return <h1>{this.props.message}</h1>
 }
}
export default Parent;
```

Here, the <Parent /> component renders two <Child /> components, passing message for child inside the first component and 5 inside the second one.

Stateless Functional Components

Stateless Functional Component

- Components let you split the UI into independent, reusable pieces.
- This is the beauty of React; we can separate a

page into many small reusable components.

- Prior to React v14 we could create a stateful React component using React.Component (in ES6), or
- React.createClass (in ES5), irrespective of whether it requires any state to manage data or not.
- React v14 introduced a simpler way to define components, usually referred to as stateless functional
- components. These components use plain JavaScript functions.

For example:

```
function Welcome(props) {
 return <h1>Hello, {props.name}</h1>;
}
```

- This function is a valid React component because it accepts a single props object argument with data and returns a React element.
- We call such components functional because they are literally JavaScript functions.
- Stateless functional components typically focus on UI; state should be managed by higher-level "container"
- components, or via Flux/Redux etc.
- Stateless functional components don't support state or lifecycle methods.

Benefits:

- 1. No class overhead
- 2. Don't have to worry about this keyword
- 3. Easy to write and easy to understand
- 4. Don't have to worry about managing state values
- 5. Performance improvement
- Summary: If you are writing a React component that doesn't require state and would like to create a reusable UI, instead of creating a standard React Component you can write it as a stateless functional component.

Let's take a simple example :

- Let's say we have a page that can register a user, search for registered users, or display a list of all the registered users.

This is entry point of the application, index.js:

```
import React from 'react';
import ReactDOM from 'react-dom';
import HomePage from './homepage'
ReactDOM.render(
 <HomePage/>,
 document.getElementById('app')
);
```

- The HomePage component provides the UI to register and search for users.
- Note that it is a typical React component
- including state, UI, and behavioral code.
- The data for the list of registered users is stored in the state variable, but our reusable List (shown below) encapsulates the UI code for the list.

homepage.js:

```
import React from 'react'
import {Component} from 'react';
import List from './list';
export default class Temp extends Component{
constructor(props) {
 super();
 this.state={users:[], showSearchResult: false,
searchResult: []};
 }
 registerClick(){
 let users = this.state.users.slice();
 if(users.indexOf(this.refs.mail_id.value) == -1){
 users.push(this.refs.mail_id.value);
this.refs.mail_id.value = '';
 this.setState({users});
 }else{
 alert('user already registered');
 }
 }
 searchClick(){
 let users = this.state.users;
 let index = users.indexOf(this.refs.search.value);
 if(index >= 0){
 this.setState({searchResult: users[index],
showSearchResult: true});
 }else{
 alert('no user found with this mail id');
 } }
```

```
hideSearchResult(){
 this.setState({showSearchResult: false});
 }
 render() {
 return (
 <div>
 <input placeholder='email-id' ref='mail_id'/>
 <input type='submit' value='Click here to register'
onClick={this.registerClick.bind(this)}/>
 <input style={{marginLeft: '100px'}}
placeholder='search' ref='search'/>
 <input type='submit' value='Click here to register'
onClick={this.searchClick.bind(this)}/>
```

```
{this.state.showSearchResult ?
 <div>
 Search Result:
 <List users={[this.state.searchResult]}/>
 <p onClick={this.hideSearchResult.bind(this)}>Close
this</p>
 </div>
:<div>
 Registered users:
<br/>
{this.state.users.length ?
 <List users={this.state.users}/>
 :"no user is registered"
 }
</div>
 }
 </div>
 );
 }
}
```

Finally, our stateless functional component List, which is used display both the list of registered users and the search results, but without maintaining any state itself.

list.js:

```
import React from 'react';
var colors = ['#6A1B9A', '#76FF03', '#4527A0'];
var List = (props) => {
 return(
<div>
 {
 props.users.map((user, i)=>{
 return(
 <div key={i} style={{color: colors[i%3]}}>
 {user}
 </div>
 );
 })
 }
 </div>
 );
}
export default List;
```

Summary:

- React.createClass vs extends React.Component
- React AJAX call
- Communication Between Components
- Stateless Functional Components

Thank You………

If you have any quries please write to info@uplatz.com".