**Reactjs Document (v1)**

**Created by**

**Vineet Semwal (vineetsemwal82@gmail.com)**

**Reactjs**

React is JS library for building user interfaces

There are 2 libraries for working on web in React

1. React 2) ReactDom

**React**

Library for creating views, this is one we use

**ReactDom**

Library to render view in browser, This is the one which gets used when we use React library, we don't directly use ReactDom api

For mobile apps we have ReactNative

**Declarative Library**

React is declarative, we tell react what UI to create, react will create UI for us

With React, we do not interact with the DOM API directly. Instead, we provide instructions for what we want React to build, and React will take care of building/updating the dom for us.

**Create new react project**

npx is node package executor

npx create-react-app appname

For typescript

npx create-react-app appname --template typescript

run the app

cd into the appname and run command npm start

npx is a binary, full form node package executor, it helps to run package like create-react-app without even having to install it so npx will temporary download a package from npm registry and execute it

create-react-app is a project generator

**React Elements**

Like browser dom is made up of dom elements, React dom is made up of react elements. A react element is just a description of what an actual element will look like.

React.createElement("div", { id: "name-0" }, "scooby");

First argument is name of the element we want to create, second element provides attributes of that element, third element provides body

During rendering react will convert to normal html

<div id=”name-0”> scooby</div>

**JSX**

1. Javascript and xml
2. Not compulsory for Reactjs but it makes the job easy
3. className should be used instead of mentioning class for css class

eventHooks are also mentioned in camel case foreg. onClick instead of onclick , onChange instead of onchange

1. Prevents injection attack by escaping any input
2. Transpiled/converted by Babbel into javascript which browser understands, actually it is converted into javascript objects , React.createElement(\*)

JSX example and what it gets converted to after transpiling by Babel

JSX

const element = (

<div className="greeting" >

scooby </div>

);

Equivalent Javascript

const element = React.createElement(

'h1',

{className: 'greeting'},

'Hello, world!');

**Compiler**

Babel, transpiles jsx into pure react

**Control flow**

1. we run npm start

2) index.html is served in browser, it contains root dom node

4) control enters index.js

5) React Dom renders App component

6) App component produces html which is rendered in the browser

**Component**

1. Helps in splitting view into reusable pieces
2. Represents part of view
3. Can be nested within other components in fact recommended approach is to try splitting a big component into smaller component for reusability
4. Convention is Components name are started with Capital letter

**Two types Of Components**

1. Functional Component
2. Class Component

**Functional Component**

Stateless by nature without hooks(introduced later)

Should be preferred over class components

function Greet(){

return ( <div>hello </div>)}

**Class Component**

Stateful by nature , can maintain internal state

Provides lifecycle hooks we will discuss later

class Greet extends React.Component{

render(){

return ( <div>hello </div>)

} }

**Props**

1. Props are Read-Only, a component can't modify its own properties
2. For User defined components Reactpasses JSX attributes and children to the component as a single object. We call this object “props”.

Example

In parent component will be declared like this

<Greet msg="hello friends" />

function Greet(props){

return ( <div>hello {props.msg} </div>) ;

}

Or in class component

class Greet extends React.Component{

render(){

return ( <div>hello {this.props.username} </div>)

} }

**State**

State is similar to props, but it is private and fully controlled by the component.

Every component has its own state, state is mutable/changeable

We can set state update in a class component by calling setState method

In below example , object containing counter is set as state in constructor

Import React, { Component } from 'react';

class ClickCounter extends Component {

constructor(props){

super(props);

this.state={counter:0};// initial state

}

increment() {

this.setState({counter:this.state.counter+1});// state update

}

render() {

return (

<div>

Counter is {this.state.counter}

<br />

<button onClick={() => this.increment()}> Increment </button>

</div>

);

}

}

export default ClickCounter;

**Fragment**

Fragments let you group a list of children without adding extra nodes to the DOM.

In the below example parent element in returning jsx can't be div or any other element as it will mean an extra tag in tr

function Table(){

return(

<table>

<tr> <Columns/> </tr>

</table>

);}

function Columns() {

return (

<React.Fragment>

<td>column 1</td>

<td>column 2</td>

</React.Fragment>

);}

**Note**

<> and </> can be used as opening and closing tags in place of <React.Fragment> </React.Fragment>

**List**

Array has map() method, It creates a new array with the results of calling a function for every array element,

In the below code we passed a function(arrow function) that will run on every element, new array will have the numbers with double the value of previous array

let numbers=[1,2,3,4];

let newNumbers=number.map(num=>num\*2);

We could have used index which is the second parameter , below code does the same job as above code, plus it logs the newValue at index

let numbers=[1,2,3,4];

let newNumbers=number.map((num,index)=>{

let newValue= num\*2;

console.log("new number at index ="+index+" ="+newValue);

return newValue;

});

**Render multiple components**

We can use map method of the array to render multiple dom elements or components, in below code we have array of users, we supply a function to map which return a new dom element or component, it is applied in the JSX expression {}

function ListExample(){

const users=[{id:1, firstName:'scooby', lastName:'doo'}, {id:2, firstName:'simba', lastName:'dog'}]

return (

<ul>

{users.map(user=><li> {user.id} {user.firstName} {user.lastName} </li>)}

</ul>

);

}

export default ListExample;

**Above code will work but if we see console there is a warning that a key should be provided for list items , Read about keys in next topic**

**Keys**

1. Helps React identify which items have changed, are added, or are removed.
2. Should be given to the elements inside the array to give the elements a stable identity.
3. Use string that uniquely identifies a list item among its siblings. Mostly ids from data is used as keys foreg. user's id, product's id etc

Below code, key is user's id

function ListExample(){

const users=[{id:1, firstName:'scooby', lastName:'doo'}, {id:2, firstName:'simba', lastName:'dog'}]

return (

<ul>

{users.map(user=><li key={user.id}> {user.id} {user.firstName} {user.lastName} </li>)}

</ul>

);

}

export default ListExample;

**If you don't have any unique value in data then index can be used like in below code**

**Please note:** indexes are not recommended as key is the only thing react uses to identify the dom element when you remove a node or push a new node at start, index value of the dom element will get changed so react will not identify the dom element correctly and so it will not display/update it as expected

function ListExample(){

const cities=["delhi", "mumbai", "bengaluru"]

return (

<ul>

{cities.map((city,index)=><li key={city}> {city} </li>)}

</ul>

);

}

export default ListExample;

**Refs**

Refs are used to refer/access dom nodes

function RefExample() {

const usernameRef = React.createRef();

const submitHandler = (event) => {

event.preventDefault();

const element = usernameRef.current;

console.log("element value=" + element.value);

}

return (

<div>

<form onSubmit={submitHandler}>

<input name="username" ref={usernameRef} />

<button>Submit</button>

</form>

</div>

);

}

**Main Phases in component lifecycle**

1. Mounting: Instance of component is created and added in DOM
2. Updating: component is re-rendered when there is change in props,state
3. Unmounting : component is removed from DOM
4. Error Handling: Runs when component or its children have error

**When component is mounted, these get executed in order**

1. constructor()
2. static getDerivedStateFromProps(props)
3. render()
4. componentDidMount()

**When component is updated, these get executed in order**

1. static getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. componentDidUpdate()

**When component is unmounted,**

componentWillUnMount() is the only method that gets executed before component will be umounted/destroyed

Don't call setState(\*) method as component is never rerendered after umounting

**Error Handling**

These methods are called when there is error in rendering or there is error in constructor of child component

static getDerivedStateFromError(error)

componentDidCatch(error, info)

render() method is required in class components, others are optional

**constructor() (optional):** called once, good place to set up state,

The firstline of a constructor should be super(props), this will call parent constructor, parent features should be initialized before anything else

**static getDerivedStateFromProps() (optional)**

Good place to set the state object based on the initial props.

It takes state as an argument, and returns an object which will be the new state.

static getDerivedStateFromProps(props, state) {

return {count: props.initialCount };

}

**render() (required )**

method that actually outputs the HTML to the DOM.

**componentDidMount() (optional)**

method runs after the component output has been rendered to the Dom

**shouldComponentUpdate(props,state)**

Returns true by default , we can implement it for performance , let's say returning false if the props and state if there is no change in existing props,done by comparing.

**getSnapshotBeforeUpdate(previousProps, previousState)**

In the getSnapshotBeforeUpdate() method we have access to props and states before the component was updated

**Error Boundaries**

We don't want javascript error in one component to break the whole app

1. Error boundaries are those react components that catch error in the child component tree.
2. They catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.
3. A class component becomes an error boundary if it defines either (or both) of the lifecycle methods static getDerivedStateFromError() or componentDidCatch()

Example below shows a DisplayEven Component which throws error if number passed as props is not even,

function App(props) {

return (

<div>

<DisplayEven numberValue={2} />

<DisplayEven numberValue={3} />

<DisplayEven numberValue={4} />

</div>

);

}

function DisplayEven({numberValue}){

if(numberValue%2!==0){

throw new Error("number is not even");

}

return(

<h3>number is {numberValue} </h3>

)}

Error Bounday defined as below

import { Component } from "react"

class ErrorBoundary extends Component{

constructor(props){

super(props);

this.state={hasError:false}

}

static getDerivedStateFromError(error){

return {hasError:true};

}

render(){

if(this.state.hasError){

return(

<h3>there was some problem in rendering component</h3>

)

}

return( this.props.children);

}

}

export default ErrorBoundary;

Usage of created ErrorBoundary component is simple, now if one of the component instance fails then error will be handled by ErrorBoundaryComponent and other components will be rendered

<ErrorBoundary>

<DisplayEven numberValue={2} />

</ErrorBoundary>

<ErrorBoundary>

<DisplayEven numberValue={3} />

</ErrorBoundary>

<ErrorBoundary>

<DisplayEven numberValue={4} />

</ErrorBoundary>

**Hooks**

1. Included for allowing React features without writing class
2. They don't work in component class

**Reasons why Classes should be avoided**

1. this keyword plays differently in javascript compared to other object oriented languages
2. Need to bind event Handlers
3. Classes don't minify vey well and cause problem in hot reloading
4. No proper way to reuse stateful component logic. Hoc does solve this problem though

**useState**

Example

let [currentState, setNewState] = useState({ firstName: undefined, lastName: undefined, age: undefined });

usState hook takes initial state and returns array in which the first element represents currentState and second element represents the function which will be used to set new state

**useEffect**

1. The Effect Hook helps in performing side effects in function components
2. Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects
3. It runs both after the first render and after every update or we can say after every render in simple words.
4. First argument in useEffect is the function which will be executed every time component is rendered
5. Second argument is the array of state variables when change should get component rerendered.
6. To only run useEffect one time on first render, pass empty array
7. To run cleanup code like unsubscribe, return the function in useEffect callback function which will do the clean up
8. You can have multiple useEffects in a component

**useContext**

Context provides a way to pass data through component tree

**useSelector(hook in react-redux)**

Allows to extract data from the Redux store state

The selector may return any value as a result, not just an object. The return value of the selector will be used as the return value of the useSelector() hook.

**useDispatch(hook in react-redux)**

Used to dispatch action in react redux

**Redux**

1. Library for js apps
2. State Container for js app

**Store**: holds the state of the app

**Action:** describes change in state of the app

**Reducer:** this actually changes state of the app by applying action on the state

**Shallow Compare**

shallowCompare performs a shallow equality check on the current props and nextProps objects as well as the current state and nextState objects.

When comparing simple types (numbers, strings) it compares their values

When comparing objects, it does not compare their data - only their references are compared

**Few useful React snippet shortcuts**(vscode)

rce: For creating class component

imr: for importing React

imrc : for importing React and Component

imprc: for imping React and PureComponent

enf : for generating arrow function

**React Testing**

npm install @testing-library/react @testing-library/jest-dom

react-test-renderer

**Lifecycle**

