**React JS:**

**what:**

js library for building UI.

Run in the browser.

Things happen instantly. u dont have to wait for server.

UI - using compoenents to build these.

Header, content, Footer, side bar - each and every thing we can think of compoenents.

these things can be built separately, if you want to change only that, write once and easily reuse it.

like custom html elements.

**First React code:**

**Why?**

UI state difficult to handle with js.

focus on business logic,

huge community, performance.

**React Alternatives:**

Angular React Vue

All are greate in creating application

JQuery - just traversing on dom.

**Single page vs multiple page:**

RenderReactJS(); - this is used to render the components to UI.

React will convert the function to components and reneder to the UI.

All those jsx are converted to javascript.

In single page there will be only one app(root container.) - All the widgets will be rendered by one.

In Multipage there are multiple widgets each are renedered by different container.

**Topics to Discuss:**

Basics

Debugging

Styling components

Components deep dive

http request to back end server.

Routing

Forms and validation

Redux

Authentication

Testing

Deployment

Few Animations and Next JS introduction(this is built from ReactJS)

**let and const - different ways of creating variables.**

var was there before - with ES6 two different keywords are used.

use let if u want to create a variable values.

use const - if u want to create a constant values.

**Arrow function**

const msg = 'Hello world'

//console.log(msg)

//myfunction('raj');

//test('sekar');

mul(5);

function myfunction(name){

console.log(name);

}

const test = (name) => {

console.log(name);

}

const mul = (number) => number\*2;

// no need to have return statement and paranthesis as there is only one statement

**Export and Import modules:**

default export:

import prs from '/person.js' -

import {basedata} from '/utility.js'

named export:

import {basedata as bd} from '/utility.js'

**Understanding classes:**

const pers = new Person();

const numbers = [1,2,3];

const newnumbers = [...numbers,5,6];

//console.log(newnumbers);

const person = {

name:'Raj'

}

const addperson = {

...person,

age:'30'

}

console.log(addperson);

**Destructuring:**

Spread - takes all properties and put it in variables.

Destructuring:

const array = [1,2,3]

const [a,b] = [1,2];

console.log(a)//print 1

console.log(b)//print 2

**Destructuring For Objects:**

it is nice way of polling element from an element or from an object.

const myobj = {

name:'raj',

age:'30'

}

const {name} = myobj;//{name}, it targets the name property

console.log(name)//prints the name.

console.log(age)//prints undefined

**Reference and Primitive types:**

const person = {

name:'max'

}

const secondperson = {

...person

}

person.name = 'manu'

console.log(secondperson.name);//it prints as max - using spread operator we are copying only the properties and values during creation of

second object. So second object is not depend on first person obj. when u change it will not get afffected.

const person = {

name:'max'

}

const secondperson = person;

now both the person and secondperson reference points to same object.

when u change the value of the object , both are changed.

person.name = 'manu'

console.log(secondperson.name)// it print manu

**npm install create-react-app -g**

this will create react js environment.

**index.html**

<div id="root"></div>

**index.js:**

ReactDOM.render(<App />, document.getElementById('root'));

**App.js**

function App() {

// return (

// <div className="App">

// <h1>Hi Welcome to React JS tutorial</h1>

// </div>

// );

return React.createElement('div',{className:'App'},React.createElement('h1',null,'Welcome to React JS !!'));

}

export default App;

Components are the core building block of React apps. Actually, React really is just a library for creating components in its core.

A typical React app therefore could be depicted as a component tree - having one root component ("App") and then an potentially infinite amount of nested child components.

Each component needs to return/ render some JSX code - it defines which HTML code React should render to the real DOM in the end.

JSX is NOT HTML but it looks a lot like it. Differences can be seen when looking closely though (for example className in JSX vs class in "normal HTML"). JSX is just syntactic sugar for JavaScript, allowing you to write HTMLish code instead of nested React.createElement(...) calls.

When creating components, you have the choice between two different ways:

Functional components (also referred to as "presentational", "dumb" or "stateless" components - more about this later in the course) => const cmp = () => { return <div>some JSX</div> } (using ES6 arrow functions as shown here is recommended but optional)

class-based components (also referred to as "containers", "smart" or "stateful" components) => class Cmp extends Component { render () { return <div>some JSX</div> } }

We'll of course dive into the difference throughout this course, you can already note that you should use 1) as often as possible though. It's the best-practice

props - give access to all properties that we pass from html

**Class component Naming:**

state={

persons:[

{name:'Max', age:20},

{name:'Raj', age:30},

{name:'Dhoni', age:35}

],

otherproperties:'someOtherValue'

};

here state is a keyword

**Manipulating state:**

we extended Component class - it has setState() methos -using things only u can change the state.

switchNameHandler = ()=>{

console.log("Button clicked");

// Should not do this waay

//Rect will not pick up this change.

//you cannot mutate like this.

//this.state.persons[0].name = 'Rajasekar';

this.setState({

persons:[

{name:'Maxmillian',age: 20},

{name:'Rajasekar',age: 35},

{name:'Mahendra Singh Dhoni', age:39}

]

});

}

**onClick= it is a jsx attribute.**

this is a function which will be called onClick - which inturn call setState() method in component class and change the state.

useState - allows us to manage state in a functional component.

this method returns array with two elements, one with original other with the updated.

**Passing method references between components:**

It is like passing methods present in the App.js to Person.js as a props.

clickme = {this.switchNameHandler.bind(this,'Cricketer')}

this allows to change the data present in the parent component.

**Two ways:**

1. use bind method directly pass this reference and pass argument to the method.

2. use {() => this.switchNameHandler('Craig McMillan')}

<button onClick={() => this.switchNameHandler('Craig McMillan')}> Switch Name</button>

In App.js:

<button onClick={() => this.switchNameHandler('Craig McMillan')}> Switch Name</button>

<Person

clickme = {this.switchNameHandler.bind(this,'Cricketer')}

name={this.state.persons[0].name}

age={this.state.persons[0].age}/>

<Person

changed = {this.nameChangeHandler}

name={this.state.persons[1].name}

age={this.state.persons[1].age}>

My Hobbies : Racing</Person>

In Person.js:

<div className="Person">

<p onClick={props.clickme}>Iam a person {props.name} of age {props.age}</p>

<p>{props.children}</p>

<input type="text" value={props.name} onChange={props.changed}/>

</div>

**props.children:**

refers to any element between opening and closing component.

It can be even text or even another element.

Ex:

<Person name={}> My Hobiies are </person>

Here My hobbies are can be accessed using props.children

**Two way databinding:**

<input type="text" value={props.name} onChange={props.changed}/>

Styling:

Person.css

.Person{

width: 60%;

margin:16px auto;

border:1px solid #eeeeee;

box-shadow:0 2px 3px #cccccc;

padding: 16px;

text-align: center;

}

in Person.js file

import './Person.css';

<div className="Person">

Inline styling:

render(

const style = {

backgroundColor:'white',

font:'inherit',

border:'1px solid blue',

padding:'8px',

cursor:'pointer'

};

return(

<button style={style}

Useful Resources & Links

create-react-app: https://github.com/facebookincubator/create-react-app

Introducing JSX: https://reactjs.org/docs/introducing-jsx.html

Rendering Elements: https://reactjs.org/docs/rendering-elements.html

Components & Props: https://reactjs.org/docs/components-and-props.html

Listenable Events: https://reactjs.org/docs/events.html

------------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------------

**To display person list:**

in App.js:

toggleListDisplay = ()=>{

const doesShow = this.state.showPersons;

this.setState({showPersons:!doesShow});

}

<button style={style} onClick={() => this.toggleListDisplay()}> ToggleDisplayList</button>

{

this.state.showPersons === true ?

<div>

<Person

clickme = {this.switchNameHandler.bind(this,'Cricketer')}

name={this.state.persons[0].name}

age={this.state.persons[0].age}/>

<Person

changed = {this.nameChangeHandler}

name={this.state.persons[1].name}

age={this.state.persons[1].age}>

My Hobbies : Racing</Person>

<Person

name={this.state.persons[2].name}

age={this.state.persons[2].age}/>

</div> : null

}

**Handling Dynamic content javascript way:**

render(){

let personList = null;

if(this.state.showPersons === true){

personList = (

<div>

<Person

clickme = {this.switchNameHandler.bind(this,'Cricketer')}

name={this.state.persons[0].name}

age={this.state.persons[0].age}/>

<Person

changed = {this.nameChangeHandler}

name={this.state.persons[1].name}

age={this.state.persons[1].age}>

My Hobbies : Racing</Person>

<Person

name={this.state.persons[2].name}

age={this.state.persons[2].age}/>

</div>

);

return(

{personList}

**React JS way of iterating the list:**

let personList = null;

if(this.state.showPersons){

personList = (

<div>

{

this.state.persons.map(person =>{

return <Person name={person.name} age={person.age}/>

})

}

</div>

);

}

**Deleting the object in the list:**

deletePersonHandler = (personIndex) =>{

const persons = this.state.persons;

persons.splice(personIndex,1);

this.setState({persons:persons});

}

if(this.state.showPersons){

personList = (

<div>

{

this.state.persons.map((person,index) =>{

return <Person

click={() => this.deletePersonHandler(index)}

name={person.name}

age={person.age}/>

})

}

</div>

);

}

//Using spread operator to create the copy of the array or list. slice() is also one of the way to get the copy of the list that is older way.

deletePersonHandler = (personIndex) =>{

//const persons = this.state.persons.slice;

const persons = [...this.state.persons];

persons.splice(personIndex,1);

this.setState({persons:persons});

}

**Update List:**

nameChangeHandler = (event,id)=>{

const personIndex = this.state.persons.findIndex(p=>{

return p.id === id;

});

// dupicate the object using spread operator.

const person = {

...this.state.persons[personIndex]

};

//const person = Object.assign({},this.state.persons[personIndex]);

person.name = event.target.value;

const persons = [...this.state.persons];

persons[personIndex] = person;

this.setState({persons:persons});

**Stying React components and list:**

setting styles dynamically

if(this.state.showPersons){

personList = (

<div>

{

this.state.persons.map((person,index) =>{

return <Person

clickme={() => this.deletePersonHandler(index)}

name={person.name}

age={person.age}

key={person.id}

changed={(event) =>this.nameChangeHandler(event,person.id)}/>

})

}

</div>

);

style.backgroundColor = "red";

}

const classes=[];

if(this.state.persons.length <= 2){

classes.push('red');

}

if(this.state.persons.length <=1){

classes.push('bold');

}

<p className={classes.join(" ")} >Welcome to all persons who have onboarded today.</p>

**Adding and Using Radium:**

To Apply styles like hover we are using Radium.

render(){

const style = {

backgroundColor:'white',

font:'inherit',

border:'1px solid blue',

padding:'8px',

cursor:'pointer',

':hover':{

backgroundColor:'green',

color:'yellow'

}

};

export default Radium(App);

Using Radium for Media queries:

**Debugging react app:**

install reach plugin on the chrome.

put f12 , in the console tab, search for source tab, put a debug point. Step through using f9, f10

for some error it will show the line number in the page itself

**Component Deep Dive:**

global styles are mentioned in index.css

App.js:

render()

Project structure:

**Stateless and stateful components:**

class based vs functional based:

class have state and lifecycle hooks.

Access state and props using this.

this.state.xy, this.props.xy

Functional can use the state(access to state) but no lifecycle hooks

access props via props(a parameter that is passed.)

Use it for all presentational component.

When you want to pass the value from one compoenent to other component, put it as a attrbute.

Then in the other compoenent - get it as props.

Even for App.js - you can pass props from index.js. <App title="Senior Software Engineer"/>

Then in the App.js, you can get it like {this.props.title}

**Class component lifecycle:**

only available in class based

Lifecycle:

contructor(props) - if u add ur own constructor, then u should call

super(props) - this as the first line

getDeriveStateFromProps(props,state) - not much use.

render() - prepare and structure your jsx

Render child compoenents

componenentDidMount() - it will trigger rerender cycle

class App extends Component{

constructor(props){

super(props);

console.log("App js constructor.");

this.state={

persons:[

{id:'asdasd', name:'Raj', age:30},

{id:'fsdsfs',name:'Max', age:20},

{id:'gggd',name:'Dhoni', age:35}

],

showPersons:false

}

}

static getDerivedStateFromProps(props,state){

console.log("[App.js] get derived state from props .",props);

return state;

}

componentDidMount(){

console.log("Component did mount is called..");

}

componentWillMount(){

console.log("Component will mount is called...");

}

Component update lifecycle for prop changes:

class Persons extends Component{

// static getDerivedStateFromProps(props,state){

// console.log("[Persons.js] get derived state from props .",props);

// return state;

// }

shouldComponentUpdate(nextProps, nextState){

console.log("[Persons.js] shouldComponentUpdate() ");

return true;

}

getSnapshotBeforeUpdate(prevProps, prevState){

console.log("[Persons.js] getSnapshotBeforeUpdate() ");

return null;

}

componentDidUpdate(){

console.log("[Persons.js] componentDidUpdate() ");

}

render(){

console.log("[Persons.js] is rendering...");

return this.props.personlistprop.map((person,index) =>{

return (

<Person

clickme={() => this.props.clickdeleteprop(index)}

name={person.name}

age={person.age}

key={person.id}

changed={(event) =>this.props.changed(event,person.id)}/>

);

});

}

}

export default Persons;

Component update lifecycle for state changes:

shouldComponentUpdate() - used for performance improvements.

componentDidUpdate() - used to fetch data from server.

Before React 16 functional component are used only as a presentational component.

Like class component life cycle hook - we have functional compoenent life cycle hook called react hook called useEffect().

useEffect(()=>{

console.log("[cockpit js ] rendered again..");

});

it executes for every render. even when u refresh the page it renders

This will run only when the persons list is changed. the second parameter[props.persons] keeps watch for the change in the persons object

useEffect(()=>{

console.log("[useEffect() cockpit js ] rendered again..");

},[props.persons]);

This will run only once - that is first time

useEffect(()=>{

console.log("[useEffect() cockpit js ] rendered again..");

},[]);

**Clean up work using useEffect and lifecycle hooks:**

**Class based component:**

This method will be called whenever component is removed.

componentWillUnmount(){

}

**Functional based component:**

useEffect(()=>{

console.log("[useEffect() cockpit js ] rendered again..");

return(()=>{

console.log("Cockpit is removed -- clean up activity...")

});

},[]);

useEffect(()=>{

console.log("[2nd -- useEffect() cockpit js ] rendered again..");

return(()=>{

console.log("2nd --- Cockpit is removed -- clean up activity...")

});

});

return(()=>{

console.log("Cockpit is removed -- clean up activity...")

});

this will be called whenever we remove the component.

This is called every time when we toggle, also cleanup called before the actual

`return(()=>{

console.log("2nd --- Cockpit is removed -- clean up activity...")

});

**How React updates the DOM:**

render()

it has the old virtual dom and compare it with new virtual dom and only the differences it updates in the new virtual dom.

it will not completely re-render everything only the differences it will be updated.

**Rendering Adjacent jsx elements:**

As of now we are putting everything on one root element as div and then we are returning.

class Person extends Component{

render(){

console.log("[Person js] is rendering..");

return (

<p onClick={this.props.clickme}>Iam a person {this.props.name} of age {this.props.age}</p>

<p>{this.props.children}</p>

<input type="text" value={this.props.name} onChange={this.props.changed}/>

);

}

}

you will get an error.

**Solution 1:**

If we put them in an array and separate each and every html elements as comma - then it works

class Person extends Component{

render(){

console.log("[Person js] is rendering..");

return (

[

<p onClick={this.props.clickme}>Iam a person {this.props.name} of age {this.props.age}</p>,

<p>{this.props.children}</p>,

<input type="text" value={this.props.name} onChange={this.props.changed}/>

]

);

}

}

export default Person;

**Solution 2:**

Using Auxiliary

class Person extends Component{

render(){

console.log("[Person js] is rendering..");

return (

<Auxiliary>

<p onClick={this.props.clickme}>Iam a person {this.props.name} of age {this.props.age}</p>

<p>{this.props.children}</p>

<input type="text" value={this.props.name} onChange={this.props.changed}/>

</Auxiliary>

);

}

}

export default Person;

**Auxiliary.js:**

import React from 'react';

const auxiliary = (props) => props.children;

export default auxiliary;

Auxiliary is an custom built higher order components. HOC

**Solution 3:**

We can use inbuilt React.Fragments and wrap all the elements inside that.

class Person extends Component{

render(){

console.log("[Person js] is rendering..");

return (

<React.Fragment>

<p onClick={this.props.clickme}>Iam a person {this.props.name} of age {this.props.age}</p>

<p>{this.props.children}</p>

<input type="text" value={this.props.name} onChange={this.props.changed}/>

</React.Fragment>

);

}

}

export default Person;

**HOC: Introduction:**

Aux : it wraps the other component. just a technical wrapper - it does not add any logic.

**Before HOC WithClass.js:**

In App.js

<div className="App">

<button onClick={()=>{this.setState({showCockpit: false})}}>RemoveCockpit</button>

{

this.state.showCockpit ?

<Cockpit title={this.props.title} persons={this.state.persons} clicked={this.toggleListDisplay}></Cockpit>

: null

}

{personList}

</div>

WithClass.js:

import React from 'react';

const withClass = (props) =>(

<div className={props.classes}>

{props.children}

</div>

);

export default withClass

in App.js - wrap the entire thing with instead of div.

<WithClass classes={classes.App}>

</WithClass>

**Using HOC as non component**

this is like adding some div component over the wrapped component.- here u saw we have applied the className as wrapper for the Wrapped component.

then in the actual component - use <Auxiliary>

then in the export of actual component - invoke like this (like a wrapper u have to call the wrapped component)

export default withClass(Person,classes.Person);

in withClass.js:

const withClass = (WrappedComponent,className) =>{

console.log("The class name passed is "+className);

return props => (

<div className={className}>

<WrappedComponent {...props}/>

</div>

);

};

export default withClass;

class Person extends Component{

render(){

console.log("[Person js] is rendering..");

return (

<Auxiliary>

<p onClick={this.props.clickme}>Iam a person {this.props.name} of age {this.props.age}</p>

<p>{this.props.children}</p>

<input type="text" value={this.props.name} onChange={this.props.changed}/>

</Auxiliary>

);

}

}

export default withClass(Person,classes.Person);

**Setting the state correctly:**

u have to write a function , get the prev state and then do operations on top of it and then return it.

//this.setState({persons:persons,changeCounter:this.state.changeCounter+1});

this.setState((prevState,props) =>{

return{

persons:persons,

changeCounter:prevState.changeCounter+1

};

});

**Using propTypes:**

if anyther ppl used prop wrongly then it is good to have - so that from the error they will get to know how to use the component.

npm install --save prop-types

in Person.js

import PropTypes from 'prop-types';

Person.propTypes = {

click: PropTypes.func,

name:PropTypes.string,

age:PropTypes.number

}

**Using Refs:**

if you want to focus on the last element on the page.

ref - is the keyword provided by react.

solution 1: function based

in Person.js:

componentDidMount(){

this.inputElement.focus();

}

ref = {(inputEl) =>{this.inputElement = inputEl}}

type="text"

value={this.props.name}

onChange={this.props.changed}

/>

</Auxiliary>

);

solution 2: - Constructor based:

constructor(props){

super(props);

this.inputElementRef = React.createRef();

}

componentDidMount(){

//this.inputElement.focus();

this.inputElementRef.current.focus();

}

key="i3"

// ref = {(inputEl) =>{this.inputElement = inputEl}}

ref = {this.inputElementRef}

**Props chaining problem:**

where you have to pass the props from one component to other component.

Ex: authenticated boolean, have to pass from App.js to persons.js and then in the person.js - based on the value display a text along with html element.

button created in cockpit.js, have to pass that from App.js ,so that during the event it will change the state of authenticated from false to true.

then again pass this to persons.js and then from there pass it to person.js

Problem:

here persons.js does not using the value it is just passing the value to person.js

**Solution:**

context API

create a file called auth-context.js

import React from 'react';

const authContext = React.createContext({

authenticated:false,

login:{}

});

Using the file:

return (

<div className = "App">

<button onClick={()=>{this.setState({showCockpit: false})}}>RemoveCockpit</button>

<AuthContext.Provider

value={{authenticated:this.state.authenticated,login:this.loginHandler}}>

{

this.state.showCockpit ?

<Cockpit

title={this.props.title}

persons={this.state.persons}

clicked={this.toggleListDisplay}

login={this.loginHandler}>

</Cockpit>

: null

}

{personList}

</AuthContext.Provider>

in person.js:

render(){

console.log("[Person js] is rendering..",this.props.isAuth);

return (

<Auxiliary>

<AuthContext.Consumer>

{

context => context.authenticated ? <p> Authenticated</p> : <p>Please Login</p>

}

</AuthContext.Consumer>

in cockpit .js:

<AuthContext.Consumer>

{context =><button onClick={context.login}>Login</button> }

</AuthContext.Consumer>

**ContextType and UseContext:**

In a class base component u need not use <AuthContext.Consumer> - instead you can use directly.

Without using the consumer u can use it wherever we want, in case if u want to send in the http request.

in person.js

static contextType = AuthContext;

componentDidMount(){

//this.inputElement.focus();

this.inputElementRef.current.focus();

console.log(this.context.authenticated);

}

**In a function based component - u can use useContext() hook.**

In cockpit.js:

import React,{useEffect,useContext} from 'react';

const authContextObj = useContext(AuthContext);

<button onClick={authContextObj.login}>Login</button>

export default authContext;

Planning React App:

component tree/structure.

Application state

component vs containers

Layout

Webpack:

as a module - help us to organize javascript - single file contain everything. it contain scripts and dependencies

break all js into own island - break application into multiple file.

easy to access third party code

index.html

app.js, react.js, react-dom.js, utils.js

Before web-pack:

u have to manually load all these files in an order

After web pack:

will have different modules like public(index.html, bundle.js) src(app.js, utils.js) node\_modules(react.js, react-dom.js)

during build all these source codes are packed as a single file called bundle.js and then refered in the index.html.

it is very easy in managing the files.