**React JS Tutorial**

* What is React?
* React journey so far.
* Prerequisites to learn React.
* React important features.

What is React?

**React is an Open Source JavaScript library** used for creating dynamic and interactive user interfaces for mobile and web applications. It is highly flexible, declarative and efficient for developing scalable, simple, and fast front-end for web & mobile applications. In simple terms, React JS effectively handles the view layer of mobile and web application.

**React is flexible** in such a way that, in any application, we can use as little or as much React as you need. For example, react can be used in any existing web application to develop a new feature or even the application’s entire UI.

**React is only concerned with rendering data to the Document Object Model** (DOM) and so creating React applications usually requires the use of additional libraries for implementing things like state management and routing which we will discuss in our upcoming videos.

**ReactJS has become highly popular** across the globe because of its extra simplicity and flexibility. Many people are even referring to ReactJS as the future of web development.

Part of this huge popularity comes from the fact that top corporations such as Facebook, PayPal, Uber, Instagram, Airbnb etc use it to develop the user interfaces.

## React Journey

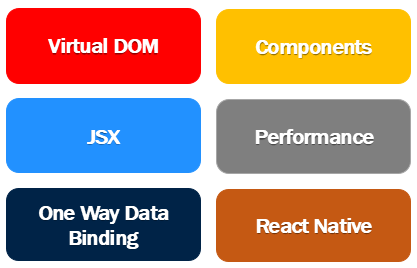
React was created by Jordan Walke, who is a software engineer at Facebook, It was first deployed on Facebook's News Feed in 2011 and later on Instagram in 2012. It was open-sourced at JSConf US in May 2013.

* React Native, which enables native Android and iOS development with React, was announced at Facebook's React Conf in February 2015 and open-sourced in March 2015.
* On April 18, 2017, Facebook announced React Fiber, a new core algorithm of React library for building user interfaces. React Fiber was to become the foundation of any future improvements and feature development of the React library.
* Like any other Library or Framework, even React has several versions as part of its evolution and 16.12 is the Current Release and we will discuss the various versions available in React and their important changes in our upcoming tutorials.

### Prerequisites to learn React

* HTML
* CSS
* Knowledge of JavaScript and ES6(let and constant, classes and Arrow functions)
* Basic Knowledge of Node and npm

We will discuss the basics of Node and NPM in our upcoming videos when we setup React



### Virtual DOM:

The most important and biggest feature introduced by React is Virtual DOM. In React, as a developer, we work with Virtual DOM instead of working directly with Real DOM. This will be discussed in our upcoming videos in detail.

### Components:

In react, every application UI is broken down into Components. The component is the most basic building block of the Application UI.

### JSX:

JSX stands for Javascript Syntax Extension. React uses JSX for building templates instead of regular JavaScript. It is not necessary to use it when we are working with React but JSX makes React a lot more elegant. This feature makes it much better than many other frameworks out there today.

### One-way data-binding:

React JS follows one-way data binding or unidirectional data flow that gives better control throughout the application.

### React Native:

React Native is an open-source mobile application framework while React is for websites(front-end). In React JS, React is the base abstraction of React DOM for the web platform, while with React Native, React is still the base abstraction but of React Native. So the syntax and workflow remain similar, but the components are different.

# ReactJS installation and setup

For setting up React in our local system, the first step is to Install NodeJs and npm.

## Install Nodejs

Node.js provides a runtime environment to execute JavaScript code from outside a browser. NPM, the Node package manager is used for managing and sharing the packages for either React or Angular.

NPM will be installed along with Nodejs. Node.js can be downloaded and installed from the official NodeJs website.

https://nodejs.org

Once the Installation of Node is complete. Open Node.Js Command Prompt and we can check the version as well.

## Install Create-React-App Tool

The next step is to install a tool called create-react-app using NPM. This tool is used to create react applications easily from our system. You can install this at the system level or temporarily at a folder level. We will install it globally by using the following command.

npm install -g create-react-app

Creating a new react project

After create-react-app is installed, we can create our first react application. Let's say I want to create the project or application in D:\React\_Programs. I will create this folder and let our command prompt point to it by using the change directory command.

Let's create a new Project now using the command.

create-react-app test-project

Remember not to create the project with an upper case character In it.

Running the React Application

Let's do CD to the Project we have created and run it locally on our system using npm start. Launch the browser and visit [http://localhost:3000](http://localhost:3000/). We can then see our first React Application response in the browser.

cd test-project

npm start

We have created a New Project using React and executed the Project.

But as a developer, we would be more interested to know about the Project which is created, its structure and we would like to play around with it. So it is time for us to get an Editor. When we think of IDE, we have a variety of choices like Visual Studio Code, React IDE, Sublime Editor, Atom Editor, Webstorm and a few others. We will use the VS Code as our Editor.

Visual Studio Code is a free IDE from Microsoft built for developing and debugging web applications. It has integrated Git control & terminal.  VS code’s IntelliSense allows Visual Studio Code to provide you with useful hints and auto-completion features while you code. So the next step is to install the Visual Studio Code.

Install Visual Studio Code

Download and install Visual Studio Code from the following URL

<https://code.visualstudio.com/download>

After the installation, open the Project we have created earlier using the VS Code. The Project has the following 3 folders

* Node\_modules
* Public
* src

The output we have seen when the Project is executed comes from a file called Index.html which resides inside the public folder.

In index.html we have one div tag with id as root.

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

To understand the relation between the output we see and this index.html, Open src/app.js file. The image and the text we see in the browser are coming from here. Let's make a small change in the text, save it and let's have a look at the browser. We can see the changes and it happens very fast.

How the index.html is linked to App.js will be discussed in our upcoming videos. With this we have the react environment setup on our local machine and we are ready to explore React.

## React online editors

Let's say we are in office, we have some free time and we’re interested in playing around with React, then you can use an online code playground like  CodePen, CodeSandbox, or Glitch.

For example, let's say we want to create react project using CodePen. In the browser, navigate to https://codepen.io/ and click on Start Coding.

Create a simple div in HTML section.

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

Followed by writing some JavaScript Code :

ReactDOM.render(

<h1>Welcome to React World</h1>,

document.getElementById('root')

);

This code will throw an error as we are missing the references to two Javascript files.

Go to Pen Settings section of Js and add,

https://unpkg.com/react/umd/react.development.js

https://unpkg.com/react-dom/umd/react-dom.development.js

One script file refers to React and the other refers to ReactDOM which is the Virtual DOM introduced by React. Set the Javascript Preprocessor to Babel.

With the above settings, you should have the output produced as expected.

Babel is a free and open-source JavaScript transcompiler that is mainly used to convert ECMAScript 2015+ code into a backwards-compatible version of JavaScript that can be run by older JavaScript engines. Babel is a popular tool for using the newest features of the JavaScript programming language. More about Babel will be discussed in our upcoming videos.

I hope we are clear on doing the React setup and creating our first Project using React.

# Introduction to React Element

In this Article, we will understand about what are React elements, how do we create Elements and how do we render Elements

Elements are the smallest building blocks of React apps. An element specifies what should be there in our UI. An Element is a plain object describing what we want to appear in terms of the DOM nodes.

Creating a React Element is Cheap compared to DOM elements.  An Element can be Created by using JSX or React without JSX. Lets create our first React Element using JSX. Open index.js file from src folder.

const element = <h1>Hello, world</h1>;

We have created an Object of type h1 and assigned it to a variable called as element. This element should be rendered into the Browser DOM, and for that we need a container. Open Index.html and there is a div with id as root and we will use that div as our container.

The above created Element can be rendered into it By writing

ReactDOM.render(element, document.getElementById('root'));

Lets say we want to assign some styles to this element. Create a css class with classname as testclass in index.css. Apply this class to the element using className attribute.

const element=<h1 className="testClass">Hi...</h1>

An Element contains type and properties, here h1 is the type and this element contains className as property. An Element can be as simple as the one we have created or an element can contain elements inside it.

I am creating an Element which contains div and div has a h1 tag and h2 tag and we render this element to the div container.

const element = (

<div>

<h1>Welcome to React Programming World</h1>

<h2>Understanding React Rendering…</h2>

</div>

);

ReactDOM.render(element, document.getElementById('root'));

Lets save these changes and observe the response in the browser.

The above code we have written using JSX would be compiled into Plain Javascript using Babel. JSX is not a requirement for using React. Using React without JSX is especially convenient when you don’t want to set up compilation in our build environment.

A React Element can also be created by using

const element = React.createElement("h1",null,'Hello World');

CreateElement method takes few parameters. First one is the type of Element we want to create like h1 or h2 or div. Second parameter specifies the Properties we want to apply to this element. Third one represents the Child Element or Elements that has be placed inside the Parent one. The above line of code tells React that we want a h1 element without properties or attributes applied and Hello World being the Child in it.

If we want to apply the css class we have created to this element then, lets add a Property called className

const element = React.createElement("div",{className:"testClass"},'Hello World');

Lets save this change and see the output in the browser.

Now lets say, we want to create one h1 tag and one h2 tag inside the div element. To achieve this, we have to pass two elements as child elements to this div. So lets go ahead and make changes.

const element=React.createElement("div",{className:"testClass"},

React.createElement("h1",null,'Welcome to SpringPeople Technologies'),

React.createElement('h2',null,'I am from h2 Tag'));

Lets save this change and see the output in the browser.

To understand the difference between React with JSX, react withour JSX much better, lets open https://codepen.io . Click Start Coding. Create a div tag in html section with id as lets say app.

Create a React Element without using JSX and render it in the container . Lets refer to the two Javascript files which are needed to run React Code. The code can be executed without setting babel as Javascript preprocessor.

# Introduction to Components in React

In this Article, we will understand about what are React Components, how do we create Components and how do we render Components.

Components are the building blocks of any React app. Components allow us split the UI into independent and reusable pieces.

A component is combination of

1. Template using HTML

2. User Interactivity using JS

3. Applying Styles using CSS

 A typical React app will have many components like header component, navbar component, footer component and content component. Conceptually a component is a JavaScript class or function that accepts inputs which are properties(props) and returns a React element that describes how a section of the User Interface should appear. A component can be created as Function Component or Class Component.

Open the demo-project using VS code and run this Project using Nodejs Command Prompt. Lets say we want to Create a Javascript function which takes an Object and returns a container which displays Employee Information.

function Employee(data) {

return <div><p>Name : {data.name}</p>

<p>Salary : {data.salary}</p></div>;

}

The above function is an example of Component and we can Call this by writing

const element = <Employee name="Sara" salary="12345" />;

and this element can be rendered using

ReactDOM.render(element, document.getElementById('root'));

In our Previous sessions, we have created React elements that represent DOM tags. However, elements can also represent user-defined components. When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object. We call this object “props”.

A function can also be created using Arrow function. Lets modify the same code using Arrow function.

const Employee=(data)=> {

return <div><p>Name : {data.name}</p>

<p>Salary : {data.salary}</p></div>;

}

Note: Always start component names with a capital letter. React treats components starting with lowercase letters as DOM tags. For example, <div /> represents an HTML div tag, but <Employee /> represents a component. Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail.

Now lets Say along with the Employee Details, we would like to Display Employee Department information as well. One way is to write the Code to display department information in the Employee Component. It is not a good Practice to keep everything in one component. To promote Code reusability, we will split them into different components. So lets create a New Department Component which will display Department Information and this Component can be used by any other component.

const Employee=(data)=> {

return (<div><p>Name : {data.name}</p>

<p>Salary : {data.salary}</p>

<Department dept={data.dept} head={data.head}/>

</div>);

}

const Department=(deptInfo)=>{

return <div><p>Dept Name is : <b>{deptInfo.dept}</b></p>

<p>Dept Head is : <b>{deptInfo.head}</b></p>

</div>;

}

const element = <Employee name="Sara" salary="12345" dept="Test" head="Head" />;

ReactDOM.render(element, document.getElementById('root'));

The components we have created until now are called as Function Components. We will understand about Class Components in our Next Session.

# Class Components in React

As discussed in our previous article, React lets us define components as classes or functions. we have discussed about Function Components in our last article and In this article, we will understand Class Components.

To define a React component class, We have to create a class and extend React.Component class. Lets say we want to create an Employee Component and this Component should return the Element which displays Employee Details.

Open Index.js file from our Demo-Project, Create Employee class and extend it from React.Component. Output of any Class Component we create is dependent on the return value of a Method Called render(). The render() method is the only required method needs to be implemented in a class component.

lets create a div element which displays employee details like ID, Name , Location, Salary and return the div from this Method. To access the attributes that will be passed to this Component Class, in React we use this.props. Attribute Name. this.props contains the props that were defined by the caller of this component.

class Employee extends React.Component {

render(){

return <div>

<h2>Employee Details...</h2>

<p>

<label>Name : <b>{this.props.Name}</b></label>

</p>

</div>;

}

}

Calling the Class Component and rendering remains as same as the Function Component.

const element=<Employee Name="SpringPeople"/>

ReactDOM.render(element,document.getElementById("root"));

Lets save these changes, navigate to the browser and we can see the output.

Now lets create Department Component also as Class Component. So we create a Class, Name it as Department and extend React.Component. We will return an Element which displays Department Information like Department Name, Head of the Department Name and Use this Component in Employee Component.

class Employee extends React.Component {

render(){

return <div>

<h2>Employee Details...</h2>

<p>

<label>Name : <b>{this.props.Name}</b></label>

</p>

<Department Name={this.props.DeptName}/>

</div>;

}

}

class Department extends React.Component {

render(){

return <div>

<h2>Department Details...</h2>

<p>

<label>Name : <b>{this.props.Name}</b></label>

</p>

</div>;

}

}

const element=<Employee Name="SpringPeople" DeptName="Dev"/>

ReactDOM.render(element,document.getElementById("root"));

Props are Read-Only

Whether we declare a component as a function or a class, it must never modify its own props.

To understand this with an Example, lets go and add a Constructor to the Employee Component Class and inside the Constructor, lets try to log the Property Object.

constructor(){

console.log( this.props);

}

This Code will throw the Error and It is expecting us to call the Base Class Constructor and we do that by using Super();

After adding the Base Class Constructor call, if we look at the Console in the browser, we get props value as undefined. To make sure that props object can be accessed in constructor, we have to add this parameter to the constructor and pass that to the base class constructor as well. Now if we save this, we can see that object data in the Console log.

Now if we try to Change the Salary using props object,  we can see an error and error says.

React is pretty flexible but it has a single strict rule. Props are read-only. Of course, application UIs are dynamic and change over time. a new concept named “state” allows React components to change their output over time in response to user actions without violating this rule. By now we are clear on how to create function component and A Class Component. Then the obvious question is when to use which one?

If we are expecting features like

1. Managing State of the Components

2. Adding Life Cycle Methods to Components

3. Need to Write Logic for Event Handlers

Then we will go for Class Component and in rest of the cases we can go for Function Component.

We will be discussing these features in detail in our upcoming articles.

# State in React

In this article, we will understand the concept of State in React and its importance. In our previous article, We have discussed about properties and their ready-only behavior.

Application UIs are dynamic and change over time. And any changes we do on the Component Class members for various user actions should get updated into Browser UI. Now question is as our props are read only , how do we address this. To understand it in Practical, I have opened our demo-project we have created in our Previous Sessions and running it in the local environment.

Lets Open index.js , create a Component Class called Employee and we implement render method. Lets return div which contains one h2 tag and display text called Employee Component and place a button in the Text with Text as Add Employee. Lets Create a function called AddEmployee, in which lets show a simple alert message and to call this function on Click of the button, we use onClick attribute and lets Pass the AddEmployee to it. Lets Call this Component and render it to our DOM. Lets save these changes, navigate to our browser. And we can see the Employee Component contents are being shown.

class Employee extends React.Component {

addEmployee = () => {

alert('Clicked on addEmployee Method');

}

render() {

return <div>

<h2>Employee Component...</h2>

<button onClick={this.addEmployee}>Add Employee</button>

</div>

}

}

const element1=<Employee></Employee>

ReactDOM.render(element1,document.getElementById("root"));

Click on the Button and we can see the alert is being shown with the message. Now lets say we want to keep a track on how many times the button is being Clicked. Lets create a counter variable and initialize with zero. In AddEmployee function, lets increment this counter and display that in the alert.

count=0;

addEmployee = () => {

this.count=this.count+1;

alert(this.count);

alert('Clicked on addEmployee Method');

}

Lets save these and navigate to the browser, Click on the button several times and one can see that counter is incremented each time when the button is Clicked. Lets delete the alerts we are showing and create one paragraph element and display the counter value in it. Lets save these and navigate to the browser, Click on the button several times and one can see that counter is not incremented each time when the button is Clicked. That’s because the change we are doing on the counter property in the component class is not getting rendered into our UI.

Lets see how we can solve this in React. React introduces, a new concept named “state” which allows React components to change their output over time in response to user actions without violating this rule.

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

State contains data specific to a given component that may change over time.

The state is user defined plain javascript object.

By adding a “local state” to a class, we can move data from the props onto a state which can be updated. Lets create state object, add counter property in that object with a default value of 0. When the addEmployee method is Called, we have to incremenet the counter and this change should be updated in the UI. React provides a method called setState()  for managing  component states.

setState() tells React that this component and its children (sometimes delayed and grouped into a single batch) should be re-rendered with the most updated state. Lets Pass an Object which contains counter and write an arrow function which increments the Counter.  Lets display the counter value from the state object we have created. Save these changes, go back to the browser and one can see that counter value is increment on multiple button clicks.

class Employee extends React.Component {

state={count:0};

addEmployee = () => {

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

}

render() {

return <div>

<h2>Employee Component...</h2>

<button onClick={this.addEmployee}>Add Employee</button>

<p>

<label>Add Employee Button is Clicked : <b>{this.state.count}</b></label></p>

</div>

}

}

To make our understanding better, lets look at another example. Lets say we want to implement a scenario which we all see in many websites. We want to have a textbox where user can enter a message and a label which should display the number of characters entered in the textbox. To implement this, we create a Component Class, Lets Create Constructor with props as the argument, lets make base class constructor call using super. Lets initialize state object with message property in it. Lets set default message as empty.

Lets implement render method, I place a textbox and a label. When user enters the message in the textbox, this count message needs to displayed in the label. Lets create function named onMessageChange which will accept the entered text in the textbox as parameter. Change the state object message to display that Message has these many number of characters. Lets call the function when the input changes by using onChange Event. Lets pass the text in the input element as parameter to onMessageChange function.

class CountCharacters extends React.Component{

constructor(props){

super(props);

this.state={

message:'',

counter:10

};

}

onMessageChange(text){

this.setState({

message:'Message has '+text.length+' number of Characters'

});

}

render(){

return <div>

<h2>Welcome to Count Characters Component...</h2>

<p>

<label>Enter Message : <input type="text"

onChange={e=>this.onMessageChange(e.target.value)}></input></label>

</p>

<p>

<label>{this.state.message}</label>

</p>

<p>

<label>{this.state.counter}</label>

</p>

</div>

}

}

Lets Call this in the Component and render this.

const element=<CountCharacters></CountCharacters>

ReactDOM.render(element,document.getElementById("root"));

Save the changes, go back to the browser and as we keep entering the text, we can see that is being displayed in the label. Lets add one more variable to the state object named count and assign it to lets say a default value of 10. Now when the text is changed, we are triggering testClick function and inside that we are calling setState method by passing the State Object which contains only message. We are not passing the counter value. Lets display the Counter Value as well in another label.

Save the changes, go back to the browser and as we keep entering the text, we can see that Message is being displayed in one label. Counter value is also displayed in another label. State object is holding both Message and counter though we are not passing it again. I believe we have got a fundamental understanding on how we can start managing the state in a react based application, we will discuss more about session management in react in our upcoming articles.

# Interaction between Components in React

The UI of every React application we develop, gets broken down into Components. Every react application we develop will be comprising of multiple components. There will be one Root Component and this component can have one or more Child Components in it. And this nesting can go further as the Application UI gets developed.

Lets take a look at one example. We want to Develop one Employee Component, in which we will be having sections like Employee Personal Info, Project Details, Salary Details Section and Department section.

So the Application UI can be designed in such a way that, we will create components like,

PersonalInfo Component

ProjectDetails Component

SalaryDetails Component

DepartmentComponent

And in EmployeeComponent we will use the above Components.

EmployeeComponent becomes the Parent Component and the rest can be used as Child Components inside EmployeeComponent. It is a very common requirement between these components to share the data. Either from parent to Child, Child to parent or between Siblings. In this article, we will understand how do we pass the data from parent to Child and Child to Parent.

So lets start with the first One, from Parent to Child. There are various ways of making this communication to happen from Parent to Child. The simplest and straight forward way of doing this is through properties.

Lets Open Index.js file, Create Employee Component which will display Employee Details. To save our time, I have kept the Code ready by ing from out last sessions and pasting it here.

class Employee extends React.Component{

constructor(props){

super(props);

}

render(){

return <div>

<h1>Employee Component...</h1>

<p>

<label>Id : <b>{this.props.Id}</b></label>

</p>

<p>

<label>Name : <b>{this.props.Name}</b></label>

</p>

<p>

<label>Location : <b>{this.props.Location}</b></label>

</p>

<p>

<label>Total Salary : <b>{this.props.Salary}</b></label>

</p>

</div>

}

}

Now we want to display the Salary Breakup details.

Lets go ahead and Create Salary Component which will display Employee Salary Information like basic Salary, HRA and Special Allowance. I have kept the Code ready and pasting it here.

class Salary extends React.Component{

constructor(props){

super(props);

}

render(){

return <div>

<h1>Salary Details...</h1>

<p>

<label>Basic Salary : <b>{this.props.BasicSalary}</b></label>

</p>

<p>

<label>HRA : <b>{this.props.HRA}</b></label>

</p>

<p>

<label>Special Allowance : <b>{this.props.SpecialAllowance}</b></label>

</p>

</div>

}

}

Lets Call this Salary Component from Employee Component.

<Salary BasicSalary={this.props.BasicSalary} HRA={this.props.HRA} SpecialAllowance={this.props.SpecialAllowance}></Salary>

Now Employee is the Parent and Salary Component is the Child. Parent Component is passing the Data to Child Components through properties.

Lets Call the Employee Component and lets render it

const e=<Employee Id="101" Name="SpringPeople" Location="Bangalore" Salary="50000" BasicSalary="25000" HRA="10000" SpecialAllowance="15000"></Employee>

ReactDOM.render(e,document.getElementById("root"));

Lets save these changes , navigate to the browser and we can see the Output. We are passing the data from Employee Component to Salary Component.

Now We want to allow People to change the salary details let it be Basic or HRA or Special Allowance , Resulting Updated Total Salary in the Employee Component should get displayed.

 That means we have to Pass the data from Child to Parent. To allow users to change the salary details, lets create state object in the constructor, add respective properties and initialize them with the data from our props.

constructor(props){

super(props);

this.state={

basic:this.props.BasicSalary,

hra:this.props.HRA,

sa:this.props.SpecialAllowance

};

Lets display the salary details in textboxes by placing input elements and assign the defaultValue by reading from State Object.

<p>

<label>Basic Salary :<input type="text" defaultValue={this.state.basic} ref="BasicSalary"/></label>

</p>

<p>

<label>HRA : <input type="text" defaultValue={this.state.hra} ref="HRA"/></label>

</p>

<p>

<label>Special Allowance : <input type="text" defaultValue={this.state.sa} ref="SpecialAllowance"/></label>

</p>

Place a Button with the text as Update and Call a function Called as UpdateSalary on the Click.

<button onClick={this.updateSalary}>Update</button>

Implement updateSalary function. In this function we have to calculate TotalSalary based on Basic Salary, HRA and Special Allowance. To access the input values in this function, we can either handle onChange event on every input element and change the state object data accordingly as discussed in our previous session or associate a reference to each input field.

In this example we add a reference to each input field using ref. So I go and add ref=”basicSalary” ref=”HRA” ref=”SpecialAllowance” to our input fields.

Now lets do the total Salary Calculation in our UpdateSalary function by using refs.referenceName.value.

updateSalary=()=>{

let salary=parseInt(this.refs.BasicSalary.value)+parseInt(this.refs.HRA.value)+ parseInt(this.refs.SpecialAllowance.value);

}

This salary should be pushed from Child to Parent and we can do that by using Callbacks.

So after the Salary is Calculated, I push this salary into a new Property called onSalaryChanged and use this property as Callback.

this.props.onSalaryChanged(salary);

Now lets go to the place where this Salary Component is being Called from, add a New Property Called onSalaryChanged and assign a function name assuming getUpdatedSalary from EmployeeComponent which should be Called.

<Salary BasicSalary={this.props.BasicSalary} HRA={this.props.HRA} SpecialAllowance={this.props.SpecialAllowance} onSalaryChanged={this. getUpdatedSalary }></Salary>

Lets implement getUpdatedSalary function in EmployeeComponent. getUpdatedSalary function will receive the salary from Salary Component and we can store this in state object using setState method

getUpdatedSalary = (salary) => {

this.setState({updatedSalary:salary});

}

and this can be displayed in our Employee Component.

<p>

<label>Updated Salary : <b>{this.state.updatedSalary}</b></label>

</p>

Lets save these Changes go back to the browser, make some changes to the salary details. Now on Clicking on Update, we can see that Updated Salary gets displayed.

Now we have seen how to pass the data from Parent to Child and child to parent components.

# Component Communication using Context

In a typical React application, data is passed top-down (parent to child) via props, but this can be difficult for certain types of props (e.g. locale preference, UI theme) that are required by many components which are Nested at different levels within an application. In this article, we will understand how we use Context to pass the data between components at different nesting levels.

Lets take a look at one example. When an Employee is Logged into the React application, we have a Nesting of Components which are making our UI. They are App Component, Employee Component and Salary Component. App Component has an Employee Object and this data is needed by Employee Component and Salary Component in Order to function.

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

Context provides a way to share values between components without having to explicitly pass a prop through every level of the tree.

Context is primarily used when some data needs to be accessible by many components at different nesting levels.

Lets Open index.js file from our demo project, lets create three components. Lets look at this Code.

class App extends React.Component {

constructor(props){

super(props);

}

render() {

return <div>

<h2>Welcome to App Component</h2>

</div>

}

}

class Employee extends React.Component {

render() {

return <div>

<h2>Welcome to Employee Component...</h2>

</div>

}

}

class Salary extends React.Component {

render() {

return <div>

<h2>Welcome to Salary Component...</h2>

</div>

}

}

Lets Call the Salary Component from Employee Component and Call the Employee Component from App Component. Call the App Component and render it to our DOM.

Lets save these Changes, navigate to the browser and we can see the Output. Now we go to the App Component Class Constructor, create state object, add a new Property called data and assign an Employee Object.

this.state={

data:{

Id:101,

Name:'SpringPeople Tech'

}

};

This Object should be passed from App Component to Employee Component and to Salary Component. We don’t want to Pass the data through Properties from App Component to Employee Component and Pass that data again from Employee Component to Salary Component through Properties.

Lets create context object using React.createContext Method.

const EmployeeContext = React.createContext();

In App Component, We have to store the employee data with in this context object and then access this object in Employee Component and Salary Component from the context.

Every Context object comes with a Provider React component that allows consuming components to subscribe to context changes. Context Provider Accepts a value property to be passed to consuming components that are descendants of this Provider. One Provider can be connected to many consumers. Providers can be nested to override values deeper within the tree. All consumers that are descendants of a Provider will re-render whenever the Provider’s value prop changes. So we modify the way how Employee Component is being Called from App Component

<EmployeeContext.Provider value={this.state.data}>

<Employee />

</EmployeeContext.Provider>

Before we go and make changes to Employee and Salary Components to access this Object, lets display the Employee ID in the App Component by reading it from State Object. Now lets go to Employee Component class, lets create a static variable and assign the EmployeeContext Object.

static contextType = EmployeeContext;

Lets go to render method in the Employee Component, Display the Employee ID by reading it from the context object.

<p>

<label>Employee Id : <b>{this.context.Id}</b></label>

</p>

Lets repeat the same for Salary Component as well. Lets save these changes, Navigate to the Browser and we can see the Output.

If we make changes to this Employee Data in the App Component, it gets reflected in both Employee Component and Salary Component. To demonstrate this, lets place a button in the App Component, call changeEmpData function onClick of this button.

<p>

<button onClick={this. changeEmpData }>Change</button>

</p>

In the changeEmpData function, lets change the Id value and pass it to our setState method.

changeEmpData=()=>{

this.setState({data:{Id:102}});

}

Save these changes, navigate to the browser and lets test this by clicking on the button.

# Component Communication using Context Part-2

In our last article, we have discussed about using context in React to pass the data from Parent Component to Child Components which are placed at different Nesting Levels. It is often necessary to update the context from a component that is nested somewhere deeply in the component tree. In this article, we will understand how do we Update Context from a Nested Component and how to handle that Update event in the Parent component.

Lets Open Index.js file from our demo-project. I have copied the App Component and Employee Component Code from our Last article, Pasting it here.

class App extends React.Component {

constructor(props){

super(props);

this.state = {

data: {Id:101}

};

}

render() {

return <div>

<h2>Welcome to App Component</h2>

<p>

<label>Employee Id : <b>{this.state.data.Id}</b></label>

</p>

<EmployeeContext.Provider value={this.state}>

<Employee />

</EmployeeContext.Provider>

</div>

}

}

class Employee extends React.Component {

static contextType = EmployeeContext;

render() {

return <div>

<h2>Welcome to Employee Component...</h2>

<p>

<label>Employee Id : <b>{this.context.data.Id}</b></label>

</p>

</div>

}

}

const element=<App/>

ReactDOM.render(element,document.getElementById('root'));

Lets create context object, and to this context object lets add one Property with name data and initialize to empty and add New Function named changeEmployeeInfo.

We pass this function down through the context to allow child components to update the context.

const EmployeeContext = React.createContext({

data: '',

changeEmployeeInfo: () => {},

});

Lets create a function named updateEmployeeDetails, change the state object data through the setState Method.

updateEmployeeDetails = () => {

this.setState({ data:{Id:102} });

};

Add a New Property to the state object named changeEmployeeInfo whose name is same as the one we have created in our Context Object and Pass this function name as its Value.

this.state = {

data: {Id:101},

changeEmployeeInfo: this. updateEmployeeDetails,

};

Place a button in the Employee Component and call this context function on Click of the Button.

<p>

<button onClick={this.context.changeEmployeeInfo}>Change</button>

</p>

Lets save these changes, navigate to the Browser. We can see that App Component contents and Employee Component contents are being displayed. Lets click on the button and observe that Employee details are changed in both the Components.

We can achieve the same even if the Components are Nested at different Levels.

# Iterating through Lists

It is often necessary to fetch a List of Objects like either List of Employees or List of Products or List of Posts from the Data Store and display them in our Application. In this article, we will understand how to deal with List of Objects in React.

Lets go ahead and create Array of Employee Objects.

const employees = [

{Id:101,Name:'Abhinav',Location:'Bangalore',Salary:12345},

{Id:102,Name:'Abhishek',Location:'Chennai',Salary:23456},

{Id:103,Name:'Ajay',Location:'Bangalore',Salary:34567}

];

 We have to create a Component which will display the employee details by reading from this Array and In this example, we will create this as a function Component. Lets create a function and name it as Employees. This function component will accept the list of employees as a Parameter.

function Employees(props) {

}

As we know that to a Component, we can pass data through properties and assuming that array of employees will be passed to this Component through a Property name employeeList, lets access this from the function parameter.

const list = props.employeeList;

This list has the employees data. We have to navigate through this list, read the data from each object and display the details. we loop through the list using the JavaScript map() function and assign the output of this function to a variable named listElements.

const listElements = list.map((emp) =>

);

We will create another function component to which we will pass each Employee Object, and this component returns a div container in which we will display the Employee Details by reading from the function Parameter.

function Employee(props) {

return <div style={{border:"3px solid red"}}>

</div>;

}

Lets apply an inline style to this div and set the border. With in this div, we will display the Employee Details like ID, Name, Location and Salary by reading from the property Named as data through which employee information is passed to this component.

<p>Employee ID : <b>{props.data.Id}</b></p>

<p>Employee Name : <b>{props.data.Name}</b></p>

<p>Employee Location : <b>{props.data.Location}</b></p>

<p>Employee Salary : <b>{props.data.Salary}</b></p>

Lets call this Employee Component with in the map method of  Employees Component and to this Employee component we will pass the Employee Details through a Property called as data.

It is important to pass the Key to each Element with in this map method.  Keys help React identify which items have changed, are added, or are removed. The best way to pick a key is to use a string that uniquely identifies a list item among its siblings. Most often we would use IDs from our data as key.

const listElements = list.map((emp) =>

<Employee key={emp.Id} data={emp} />

);

Lets return the contents of this listElements.

return (

<div>

{listElements}

</div>

);

Lets call this Employee Component and render it to our root Container.

Save these changes, lets navigate to the browser and we can see that Employee Details are being displayed.

# Calling REST API

Irrespective of UI Framework or Library we think of using for our application development, let it be React or Angular, it is very much needed for our Application’s UI to interact with REST API’s for Performing Various Operations like getting the data from the Database or to store the Data into the database or to delete the data from the database or any other business logic execution which should happen on the server.

In this article, we will understand how do we make API Call from React to get the List of Employees. In Order to make this happen, we have to create two different Projects. First one is building a Web API which will return the List of Employees. Second One is developing a React based Application which will call this API, get the List of Employees and display in the Application.

So lets start building our REST API using ASP.NET Web API Platform. I use Visual Studio to build our Web API. Lets Open Visual Studio. Click on Create a New Project option available on the Right handside. Lets search for Asp.Net Web Application, Select the Asp.Net Web Appplication template and Click on Next. Name this as WebAPI\_React\_Client, Select the Respective location where you want to store this Project and Click on Create.

As One can see Empty Template is Selected by default and we will use the same. On the Right hand side of the Window, Please Select Web API Checkbox under Add Folders and Core References Section and Click on Create. Now lets go to Solution Explorer, right Click on Models folder and create a New Class and we can name it as Employee. Lets add Properties like Id, Name, Location and Salary to this Class. Now lets go to Solution Explorer, right Click on Controller folder and select Add Controller Option. Lets Select Web API 2 Controller-Empty template and Click on Add.

We will name this Controller as Employee Controller and lets Click on Add. Now the EmployeeController Class is generated. I will go and add a Method GetAll which will be returning List of Employees. I have the Code handy and Pasting it here.

public List<Employee> GetAll()

{

List<Employee> empList = new List<Employee>

{

new Employee{Id=101,Name="Abhinav",Location="Bangalore",Salary=12345},

new Employee{Id=102,Name="Abhishek",Location="Chennai",Salary=23456},

new Employee{Id=103,Name="Akshay",Location="Bangalore",Salary=34567},

new Employee{Id=104,Name="Akash",Location="Chennai",Salary=45678},

new Employee{Id=105,Name="Anil",Location="Bangalore",Salary=56789}

};

return empList;

}

In Order to allow any other Projects to access this Web API, we have to enable CORS. So lets go to Solution Explorer again, right click on references, Select Manage NuGet Packages.

Lets switch to Browse Tab, search for Microsoft.AspNet.WebApi.Cors. From the search results, lets select the first one and click on the Install button which is available on the right hand side.

Click on Ok button in the dialog box which is Prompted. Click on the I  Accept button and Complete the Process. Lets go to Solution Explorer, expand App\_Start, one can notice WebApiConfig.cs and Open this file.

Enable the CORS support for our Project by calling config.EnableCors method with in Register Method. Now lets go to our EmployeeController Class, apply EnableCors attribute to this Class. We can find this EnableCors Attribute Class with in System.Web.Http.Cors namespace. To this EnableCors attribute, we have to pass the list of Origins, headers and methods we want to allow. We will allow any origin, any header and any method to access this api by passing \*, \* and \*. With this we are done with Creating our Rest API. Lets run this Project using Ctrl + F5.

In the browser, lets try to access our Employee Api by appending api/Employee to the URL and we can see that list of Employees are being returned from our Api. So we are done with Part 1 and now we have to call this API from our React Project to get this list of Employees.

Lets Open Index.js file from our demo Project using Visual Studio Code. Lets create a class EmployeeComponent which extends React.Component Class.

class EmployeeComponent extends React.Component {

}

We will create a Constructor which accepts props as the parameter and lets pass this props to the baseclass constructor. Lets create state object, add a property employees and initialize it to empty array.

constructor(props) {

super(props);

this.state = {

employees: []

};

}

Now we will implement render method. From this render method, we will return a div container. In this div, we will create a table which is displaying the list of employees.

render() {

return (

<div>

<h2>Employees Data...</h2>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

<th>Location</th>

<th>Salary</th>

</tr>

</thead>

<tbody>

</tbody>

</table>

</div>

);

}

Now we have to iterate through the employees array from our state object and we do that using map method. We  create the number of table rows based on the number of elements with in this employees array. With in every row, we will display the employee details by placing this code.

{this.state.employees.map(emp => (

<tr key={emp.Id}>

<td>{emp.Id}</td>

<td>{emp.Name}</td>

<td>{emp.Location}</td>

<td>{emp.Salary}</td>

</tr>

))}

Lets call this Component and render it to our root Container.

const element=<EmployeeComponent></EmployeeComponent>

ReactDOM.render(element,document.getElementById("root"));

Save these changes, lets navigate to the browser and observe the output. Our table is empty without any data. That is because we are yet to call our Rest API from our react application.

Now question is where should we write the code of sending the API request with in our EmployeeComponent Class. We want to execute some code which fetches the list of employees from Rest API whenever the EmployeeComponent is rendered to the DOM. This is called “mounting” in React. We can declare special methods on the component class to run some code when a component mounts.

These methods are called “lifecycle methods”. React Component Classes has lifecycle methods, one of such method is componentDidMount.  using this method we can run some code when the respective component is rendered to the DOM. Lets go ahead and implement this method in our Component Class.

componentDidMount() {

}

Now the next question is how do we send the AJAX Request from our React Application. This can be done by using any AJAX library we like with React. Some popular ones are Axios, jQuery AJAX, and the browser built-in window.fetch. In this example, we will make use of fetch to make API Call and about the usage of other AJAX libraries in React, we will discuss in our upcoming sessions.

We call fetch method and we will pass the Url of Web API. This is the Url where our Web API is available and I paste it here. We call the Promise method using then, convert the response into JSON objects, and then we assign this result list to our state object employees property using setState method.

Fetch(“https://localhost:44301/api/Employee”)

.then(res => res.json())

.then(

(result) => {

this.setState({

employees: result

});

}

);

Lets save these changes, navigate to the browser and we can see the employee details in our table. As one can see, It is very easy to make API Communication from our React.

# How to Post data to ASP.NET WEB API

In this article, we will understand how do we Create a New Employee using React, how do we Post this data to the Asp.Net Web API and insert this data into the DB.

In this Demonstration, We will use Sql Server as Our Database server, I will Open the Sql Server installed in my local machine. We will use ReactAppDB as our Database and in this database we have a table called as EmployeeInfo. When we create a New Employee from our React Application, this is the table into which the data should be stored.

Lets Open the Web API Project we have created in our Last Video. We will create a New Post Method which will receive employee Object and this object data we will insert into our DB Table. We will return Boolean value which indicates the status of the new record insertion. We use ADO.NET to do this.

public bool Post(Employee employee)

{

SqlConnection conn = new SqlConnection(@"server=DESKTOP-89HQ4RL\SQLEXPRESS;database=ReactAppDB;integrated security=true");

string query = "insert into EmployeeInfo values(@Id,@Name,@Loc,@Sal)";

SqlCommand cmd = new SqlCommand(query, conn);

cmd.Parameters.Add(new SqlParameter("@Id", employee.Id));

cmd.Parameters.Add(new SqlParameter("@Name", employee.Name));

cmd.Parameters.Add(new SqlParameter("@Loc", employee.Location));

cmd.Parameters.Add(new SqlParameter("@Sal", employee.Salary));

conn.Open();

int noOfRowsAffected = cmd.ExecuteNonQuery();

conn.Close();

return noOfRowsAffected > 0 ? true : false;

}

With this we are done with Creating the required Post Method in Rest API. Lets run this Project using Ctrl + F5.

Lets Open Index.js file from our demo Project using Visual Studio Code. Lets create a class EmployeeComponent which extends React.Component Class. We will create a Constructor which accepts props as the parameter and lets pass this props to the baseclass constructor. Lets create state object, add a property message and initialize it to empty single quotes.

class EmployeeComponent extends React.Component {

constructor(props) {

super(props);

this.state = {

message: ‘’

};

}

}

Now we will implement render method. From this render method, we will return a div container. In this div, we will create all the input elements which are required to create a New Employee like Employee Id, Name, Location and Salary. We assign a reference to each input element using which we can read the value of that input.

render(){

return(

<div>

<h2>Please Enter Employee Details...</h2>

<p>

<label>Employee ID : <input type="text" ref="Id"></input></label>

</p>

<p>

<label>Employee Name : <input type="text" ref="Name"></input></label>

</p>

<p>

<label>Employee Location : <input type="text" ref="Location"></input></label>

</p>

<p>

<label>Employee Salary : <input type="text" ref="Salary"></input></label>

</p>

</div>

)

}

Lets Place a button, give the Text as Create and call a function on the click of this button.

<button onClick={this.onCreateEmployee}>Create</button>

Lets go and implement this onCreateEmployee function. Lets create an object which contains Id, Name, Location and Salary properties, assign them the values by reading the values from our input elements. This is something We have already discussed this in our previous sessions.

onCreateEmployee=()=>{

let empInfo={

Id:this.refs.Id.value,

Name:this.refs.Name.value,

Location:this.refs.Location.value,

Salary:this.refs.Salary.value

};

}

Now we have to send this object to our Web API through a Post request, so that this object details can be inserted into the Database. We call fetch method, we will pass the Url of Web API.

Next we will pass an object using which we specify the method type as POST and pass the employee object through the body property. Most importantly we should not miss to pass the content-type through the headers. We will pass application/json as the content-type.

We call the Promise method using then, convert the response into JSON object, and then set the message property of the state object using setState method upon receiving true from our Web API.

fetch('https://localhost:44306/api/Employee',{

method: 'POST',

headers:{'Content-type':'application/json'},

body: empInfo

}).then(r=>r.json()).then(res=>{

if(res){

this.setState({message:'New Employee is Created Successfully'});

}

});

Lets display this message in a paragraph tag inside our Employee Component. Now we will call this component and render it to our root container. Lets save these changes, navigate to the browser.

Provide the Employee Details, Click on Create button. One can see a message that New Employee is Created Successfully.

# Building forms in React

In this article, We will understand how to **build forms** in React.js

**Building Forms** are very common in any web application development. Unlike angular and angularJS, that gives form validation out of the box, we have to handle forms ourself in React. That includes complications like

* How to get form values.
* How to manage the form state.
* How to validate the form on the fly.
* How to show validation messages.

**In HTML**, form elements such as **<input>, <textarea>, and <select>** typically maintain their own state and update it based on user input.

**In React**, state of these input elements is typically kept in the state property of components and only updated with setState().

### Controlled Component & Uncontrolled component

An input form element whose value is controlled by React in this way is called a “**controlled input or Controlled Component**”.

The alternative is **uncontrolled inputs or components**, where form data is handled by the DOM itself. We will use ref to get the input values and Perform Operations using this data.

We have discussed both of these ways of handling input changes in our previous sessions.

Though uncontrolled inputs are the simplest way to implement the form inputs they are not powerful enough to build our real-world applications. In this demonstration, we will use Controlled Inputs and definitely we all will understand the importance of them by the end of this Video.

To understand things in Practical lets open Index.js file. We will create a **Class** called**EmployeeComponent** and extend this from **React Component**.

Add a **Constructor**which accepts props as a **parameter**and pass it to the **base class constructor**.

Let's create a **state object**, add a Property called as **Id**and initialize it to **empty**.

class EmployeeComponent extends React.Component{

constructor(props){

super(props);

this.state = {

Id:''

};

}

Now let's implement the **render method**.

* In render Method, we will create a form tag and lets place an input whose type is text.
* Assign **Id** of the state object to value and **onChange** of the input value, let's call a function.

<form>

<p>

<label>Employee ID : <input type="text" name="Id" value={this.state.Id} onChange={this.changeHandler} ></input></label>

</p>

</form>

In the below function, we save the input text box value into our state object using the **setState** method. We can get the input value using **e.target.value.**

changeHandler = e => {

this.setState({

Id:e.target.value

});

}

We will create another function called onCreateEmployee and log the Id value on to the console within this function.

onCreateEmployee=()=>{

console.log(this.state.Id);

}

Let's place a button in our render method and call this function on click of this button.

<button onClick={this.onCreateEmployee}>Create</button>

Let's call this Employee Component, render this element to our root Container.

* Save these changes, navigate to the browser and we can see that we have an input element in our page. Now enter some text into this textbox, click on the button and this value can be seen in the console window of the browser by using Developer Tools.
* Now let's back to Visual studio code, Now we have to add more inputs for Name, Location and Salary.
* In order to save these input values into our state object, let's add a new Employee object to our state and to this object, add Id, Name, Location and Salary as Properties.

this.state = {

employee:{

Id:'',

Name:'',

Location:'',

Salary:''

}

};

Let's make the change for the input element value attribute so that it points to Id property of the employee Object.

Now we will add the input elements for Name, Location and Salary as well.

<p>

<label>Employee ID : <input type="text" name="Id" value={this.state.employee.Id}

onChange={this.changeHandler} ></input>

</label>

</p>

<p>

<label>Employee Name : <input type="text" name="Name"

value={this.state.employee.Name}

onChange={this.changeHandler}></input>

</label>

</p>

<p>

<label>Employee Location : <input type="text" name="Location"

value={this.state.employee.Location}

onChange={this.changeHandler}></input>

</label>

</p>

<p>

<label>Employee Salary : <input type="text" name="Salary"

value={this.state.employee.Salary}

onChange={this.changeHandler}></input>

</label>

</p>

If we observe this code, we are calling the same function for onChange event on every input element and it is the right way to do so. Because we cant create 20 functions if we have 20 inputs.

* With this function, we have to handle the onChange event for all the inputs and update the state object using setState.

Let's get the input element Name using

const name = event.target.name;

Get the value of that input using

const value = event.target.value;

Now we call setState method and we pass an employee object.

The employee object should contain all the previous property values. We get them using Spread Operator.

we should add a new Property to this employee object. Name of this property will be the input element name and this property will hold the value of that respective input element. We make a name as the property by writing it in the square brackets and assign the value.

this.setState({

employee:{

...this.state.employee,

[name]: value

}

});

Whenever User makes a change on any of the input elements, we are updating our state object using the setState method.

Within the onCreateEmployee function, lets log this employee object into the console.

Save these changes, navigate to the browser.

Enter the data into these inputs, click on the create button and we can see that employee object details are logged into the Console window.

Now we have understood how to create a simple form.

But to this form we want to add validations, we want to keep track of the visited fields, and we have to handle form submission,

Formik is one of the popular library available and let's use this library for building powerful forms in our react application.

Formik is a small group of React components and hooks for building forms in React and React Native. It helps with the three most parts:

1. Getting values in and out of form state
2. Validation and error messages
3. Handling form submission

Let's navigate to our node js command prompt. Run a command to install formik into our project.

npm install formik –save

After the installation is successful, run our project using

Npm start

Let's switch back to Visual studio code, let's delete all the code we have written.

Lets import use Formik from formic.

Let's create EmployeeComponent class and extend it from React. Component class.

class NewEmployeeComponent extends React.Component {

}

Let's implement the render method

Let's call useFormik function, pass an object which contains initial values for Id, Name, Location and Salary.

And also pass a onSubmit function () to the useFormik(). In this function, we will display the forms of current values using an alert.

Let's return the div container in which we will place our form tag.lets pass formik. handleSubmit to the onSubmit attribute of the forms.

<form onSubmit={formik.handleSubmit}>

Let's place input elements for Id, Name, Location and Salary.

To that input element onChange event, we will pass handleChange function of formik object and we will bind the value from the formik object as well.

<p>

<label htmlFor="Id">Employee ID </label>

<input

id="Id"

name="Id"

type="text"

onChange={formik.handleChange}

value={formik.values.Id}

/>

</p>

Let's repeat the same for the rest of the input elements.

Let's add a button with in the form.

Let's Call this Component and render it to our root container.

Save these changes and let's navigate to the browser. Enter the Employee details and let's click on Create Button.

We can see that employee data is displayed in the alert as a JSON object. We can pass this data to our Web API as we have discussed in our Last Video.

If we look at the code what we have written,

We pass our form's initialValues and a submission function (onSubmit) to the useFormik() function. The function then returns us an object using which we will access

handleSubmit which is A submission handler

handleChange which is A change handler to pass to each <input>, <select>, or <textarea>

values: using which we access Our form's current values.

import React from 'react';

import ReactDOM from 'react-dom';

import { useFormik } from 'formik';

class NewEmployeeComponent extends React.Component {

render(){

const formik = useFormik({

initialValues: {

Id: '',

Name: '',

Location: '',

Salary:''

},

onSubmit: values => {

alert(JSON.stringify(values));

},

});

return (

<div>

<h2>Enter Employee Details...</h2>

<form onSubmit={formik.handleSubmit}>

<p>

<label htmlFor="Id">Employee ID </label>

<input

id="Id"

name="Id"

type="text"

onChange={formik.handleChange}

value={formik.values.Id}

/>

</p>

<p>

<label htmlFor="Name">Employee Name </label>

<input

id="Name"

name="Name"

type="text"

onChange={formik.handleChange}

value={formik.values.Name}

/>

</p>

<p>

<label htmlFor="Location">Employee Location </label>

<input

id="Location"

name="Location"

type="text"

onChange={formik.handleChange}

value={formik.values.Location}

/>

</p>

<p>

<label htmlFor="Salary">Employee Salary </label>

<input

id="Salary"

name="Salary"

type="text"

onChange={formik.handleChange}

value={formik.values.Salary}

/>

</p>

<button type="submit">Submit</button>

</form>

</div>

);

}

};

const element=<NewEmployeeForm></NewEmployeeForm>

ReactDOM.render(element,document.getElementById("root"));

# How to Validate Forms Data in React

We all know how important is doing the Validation of the Entered data before we do any operations using that data. In this article, We will understand how to Validate Forms in React.

Assuming that Our Form has total five inputs, Id, name, Location, Salary and EmailId. Now to this Form, We have to add Validation. From this forms input data, lets say we want to add Validation for Employee Name, Employee Location and Employee Email ID.

Lets define Our Validation rules before we proceed and write the code.

-Employee Name

        1. Required

        2. MaxLength 20 Characters

-Employee Location

        1. Required

-Employee Email ID

        1. Required

        2. Email Pattern

Now lets go ahead and create a function with a name as ValidateEmployee in our index.js file. This function will receive our employee object as input data. Lets create an object using which we will return error messages. Lets check if employee Name is having the data or not and if it doesnot have the data, set a New Property to our errors object and write the Error Message. If Employee Name is Entered, we have to check for maxlength validation.

Now lets repeat the same for Location and for location we do only required field validation. And we do the same for Email ID plus we will do regular expression validation to check the Email ID format. From this function, we will return the errors object.

Pass this function name to the validate Property of the useFormik function. Now lets go ahead and add the code required to display the Error Messages. It is important to display the messages only when user touch the input. To take advantage of touched, we can pass formik.handleBlur to each input's onBlur prop.

So after the input field, lets check if Name is input field is touched or not and if it is touched, check if the Name property has any errors or not. If Name has Errors, lets display the Error message in a span. Lets repeat the same for rest of the input elements.

Lets save these changes, navigate to the browser. As we keep entering the data we can see different error messages are being displayed here.

The Complete Code looks like below.

import React from 'react';

import ReactDOM from 'react-dom';

import {useFormik} from 'formik';

const validateEmployee = empData => {

const errors = {};

if (!empData.Name) {

errors.Name = 'Please Enter Employee Name';

} else if (empData.Name.length > 20) {

errors.Name = 'Name cannot exceed 20 characters';

}

if (!empData.Location) {

errors.Location = 'Please Enter Employee Location';

}

if (!empData.EmailId) {

errors.EmailId = 'Please Enter Email ID';

} else if (!/^[A-Z0-9.\_%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$/i.test(empData.EmailId)) {

errors.EmailId = 'Invalid email address';

}

return errors;

};

const EmployeeComponent=()=>{

const formik=useFormik({

initialValues:{

Id:'',

Name:'',

Location:'',

Salary:'',

EmailId:''

},

validate:validateEmployee,

onSubmit:values=>{

alert(JSON.stringify(values));

}

});

return (

<div>

<h2>New Employee Form...</h2>

<form onSubmit={formik.handleSubmit}>

<p>

<label htmlFor="Id">Employee ID : </label>

<input type="text" name="Id" id="Id" value={formik.values.Id}

onChange={formik.handleChange}></input>

</p>

<p>

<label htmlFor="Name">Employee Name : </label>

<input type="text" name="Name" id="Name" value={formik.values.Name}

onChange={formik.handleChange} onBlur={formik.handleBlur}></input>

{formik.touched.Name && formik.errors.Name ? <span style={{color:'red'}}>{formik.errors.Name}</span> : null}

</p>

<p>

<label htmlFor="Location">Employee Location : </label>

<input type="text" name="Location" id="Location" value={formik.values.Location}

onChange={formik.handleChange} onBlur={formik.handleBlur}></input>

{formik.touched.Location && formik.errors.Location ? <span style={{color:'red'}}>{formik.errors.Location}</span> : null}

</p>

<p>

<label htmlFor="Salary">Employee Salary : </label>

<input type="text" name="Salary" id="Salary" value={formik.values.Salary}

onChange={formik.handleChange}></input>

</p>

<p>

<label htmlFor="EmailId">Employee Email ID : </label>

<input type="text" name="EmailId" id="EmailId" value={formik.values.EmailId}

onChange={formik.handleChange} onBlur={formik.handleBlur}></input>

{formik.touched.EmailId && formik.errors.EmailId ? <span style={{color:'red'}}>{formik.errors.EmailId}</span> : null}

</p>

<button type="submit">Create</button>

</form>

</div>

)

}

const element=<EmployeeComponent></EmployeeComponent>

ReactDOM.render(element,document.getElementById("root"));

## ****Validation using Yup Library****

By now we have seen how to write the validation code for our form inputs. But we are free to use any third party validation library available and do the form validation.

Formik's authors/a large portion of its users use Yup library for object schema validation. Since Formik authors/users love Yup so much, Formik has a special configuration option for Yup called validationSchema which will automatically transform Yup's validation errors messages into a pretty object whose keys match our forms input values.

 Lets navigate to our node js command prompt. Run a command to install yup into our project.

npm install yup –save

After the Installation is successful, run our project using

Npm start

Lets switch back to Visual studio code, lets delete all the validation code we have written. Lets import everything from yup.

Add a new property called as validationSchema in useFormik function and assign the object function available. To this function, we pass our validation rules. First lets do it for name. we mention the data type followed by adding the validations for max length and required. Lets add the validations for Location and Email Id.

Save these changes, Navigate to the browser and we can see the validation is happening for our form data as we enter the values.

If we see our input elements, we are writing so much of code to set values for different attributes like id, value, onchange, onblur. However, to save you time, useFormik() returns a helper method called formik.getFieldProps(). Lets go and modify our input elements to make use of it and to this method we will pass the Name of our input. We will do the same for rest of the input elements as well.

Save these changes, Navigate to the browser and we can see the behaviour we have earlier.

The Complete Code looks like below.

import React from 'react';

import ReactDOM from 'react-dom';

import {useFormik} from 'formik';

import \* as yup from 'yup';

const EmployeeComponent=()=>{

const formik=useFormik({

initialValues:{

Id:'',

Name:'',

Location:'',

Salary:'',

EmailId:''

},

validationSchema: yup.object({

Name: yup.string()

.max(20, 'Name should not exceed 20 Characters')

.required('Please Enter Employee Name'),

Location: yup.string()

.required('Please Enter Employee Location'),

EmailId: yup.string()

.email('Invalid email address')

.required('Please Enter Email Id'),

}),

onSubmit:values=>{

alert(JSON.stringify(values));

}

});

return (

<div>

<h2>New Employee Form...</h2>

<form onSubmit={formik.handleSubmit}>

<p>

<label htmlFor="Id">Employee ID : </label>

<input type="text" name="Id" {...formik.getFieldProps("Id")} ></input>

</p>

<p>

<label htmlFor="Name">Employee Name : </label>

<input type="text" name="Name" {...formik.getFieldProps("Name")} ></input>

{formik.touched.Name && formik.errors.Name ? <span style={{color:'red'}}>{formik.errors.Name}</span> : null}

</p>

<p>

<label htmlFor="Location">Employee Location : </label>

<input type="text" name="Location" {...formik.getFieldProps("Location")} ></input>

{formik.touched.Location && formik.errors.Location ? <span style={{color:'red'}}>{formik.errors.Location}</span> : null}

</p>

<p>

<label htmlFor="Salary">Employee Salary : </label>

<input type="text" name="Salary" {...formik.getFieldProps("Salary")} ></input>

</p>

<p>

<label htmlFor="EmailId">Employee Email ID : </label>

<input type="text" name="EmailId" {...formik.getFieldProps("EmailId")} ></input>

{formik.touched.EmailId && formik.errors.EmailId ? <span style={{color:'red'}}>{formik.errors.EmailId}</span> : null}

</p>

<button type="submit">Create</button>

</form>

</div>

)

}

const element=<EmployeeComponent></EmployeeComponent>

ReactDOM.render(element,document.getElementById("root"));

# Forms Validation Part-2

This is Continuation to our previos article in which we have discussed about building Forms and Validating Forms in React using useFormik function. In this article, we will understand the **Formik Component.**

If we look at the code what we have written in our last session, though we have tried to make our code better by using getProps method, still we have to pass that to each input and we have to the code written for displaying the Validation Messages. To save us even more time and to make our code much better and easier, Formik comes with few built in components. They are **<Formik />, <Form />, <Field />,** and **<ErrorMessage />**. **<Formik/>** is a component that helps us with building forms. Internally, Formik uses useFormik function to create the <Formik> component. useFormik() is a custom React hook that will return all Formik state and helpers directly. Despite its name, it is not meant for the majority of use cases.

Let's now swap out the useFormik() hook for the Formik's <Formik> component. Since it's a component, we'll convert the object passed to useFormik() to JSX, with each key becoming a prop. That means initialValues, validationSchema and onSubmit will be passed to this Formik Component as properties. Lets go to Index.js file from our demo project, in our EmployeeComponent’s return Method, Call a Component Called as Formik and to this Component, lets pass initialValues, validationSchema and onSubmit as properties. We will pass the same values we have used in useFormik function object.

return (

<Formik initialValues= {{

Id: '',

Name: '',

Location: '',

Salary:'',

EmailId:''

}} validationSchema={yup.object({

Name:yup.string().max(20,'Name should not exceed 20 Characters').

required('Please Enter Employee Name'),

Location: yup.string()

.required('Please Enter Employee Location'),

EmailId: yup.string()

.email('Invalid email address')

.required('Please Enter Email Id')

})} onSubmit= {values => {

alert(JSON.stringify(values));

}}>

Lets add a div container and with in this Container, place a h2 tag with text as New Employee Form. Now lets call Form Component as Child Component from this Formik Component. Add a paragraph tag and place a label with Text as Employee ID. Now instead of placing an input, We will use another Component called as Field. To this field we will pass what type of input we want like text or number or email.

<p>

<label htmlFor="Id">Employee ID </label>

<Field name="Id" type="text"></Field>

</p>

Now lets repeat the same for Rest of the Form inputs like Name, Location, Salary and Email ID. Now We should display the Error messages when there is any Validation Error. In order to do this, we will use another Component called as ErrorMessage and specify property name for which the error should be displayed. We use an attribute called as name to specify the Property Name from our Formik values object. Now we will repeat the same for Location and Email ID as well. Lets save our changes, navigate to the browser.

As we keep playing with the form inputs, respective validation messages are displayed. When we click on the button, we can see that forms data is displayed in an alert as a json object. Now if we look at the Code, Formik is the Parent Component and it has a nesting of Child Components. They are Form, Field and Error Message. Formik uses React Context internally to pass the data between these Components.

We may think of using useFormik function if we don’t want to make use of React Context because we may want to use some other ways to share the data. In rest of the cases, it is recommended to use Formik Component. One must remember that We cant use Form, field, Error Message Components inside useFormik function. In our example we have used Field for generating the input elements. We can use Field Component to generate other html elements as well.

Now lets say we want to have a dropdownlist using which we can select the designation of the Employee. Lets add a new Property to our initialValues Object with a Name called Designation and initialize it to empty. Now with in our Form, add a new Field, assign the name as designation and to this field add a new attribute called as as and set the value as select. Add the respective options we want. For example, lets say we want to have Software Engineer, Senior Software Engineer and Lead as Designations.

Save these changes, navigate to the browser. Now we can see the dropdownlist available for us. Fill the form, click on the button. We can see the designation is added to our JSOn object as well.

Now lets say we want to disable the submit button if the form data is invalid. Formik Component has several Properties and we have used few properties like initialValues, validationSchema and onSubmit. Formik has another property called as isValid. IsValid Returns true if there are no errors  or if the errors Object is empty. And returns false if otherwise.

But in order to access any of the property from the Formik Component, we have to make a change. With in Formik Component, we will write an arrow function and properties of this formic component will be passed as a parameter to this function and this function will have our UI Code. Lets move all of our Form Code into this function.  Now to the button, add a new attribute called disabled and pass the isValid property. Now save these changes, navigate to the browser. We can see that button gets disabled when there are validation errors.

import React from 'react';

import ReactDOM from 'react-dom';

import { useFormik, Formik, Form, Field, ErrorMessage } from 'formik';

import \* as yup from 'yup';

const EmployeeComponent =()=> {

return (

<Formik initialValues= {{

Id: '',

Name: '',

Location: '',

Salary:'',

EmailId:'',

Designation:''

}} validationSchema={yup.object({

Name:yup.string().max(20,'Name should not exceed 20 Characters').

required('Please Enter Employee Name'),

Location: yup.string()

.required('Please Enter Employee Location'),

EmailId: yup.string()

.email('Invalid email address')

.required('Please Enter Email Id')

})} onSubmit= {values => {

alert(JSON.stringify(values));

}}>

{props=>(

<div>

<h2>Enter Employee Details...</h2>

<Form>

<p>

<label htmlFor="Id">Employee ID </label>

<Field name="Id" type="text"></Field>

</p>

<p>

<label htmlFor="Name">Employee Name </label>

<Field name="Name" type="text"></Field>

<ErrorMessage name="Name"></ErrorMessage>

</p>

<p>

<label htmlFor="Location">Employee Location </label>

<Field name="Location" type="text"></Field>

<ErrorMessage name="Location"></ErrorMessage>

</p>

<p>

<label htmlFor="Salary">Employee Salary </label>

<Field name="Salary" type="text"></Field>

</p>

<p>

<label htmlFor="EmailId">Employee Email ID </label>

<Field name="EmailId" type="text"></Field>

<ErrorMessage name="EmailId"></ErrorMessage>

</p>

<p>

<label>Employee Designation :

<Field name="Designation" as="select">

<option value="red">Software Engineer</option>

<option value="green">Senior Software Engineer</option>

<option value="blue">Lead</option>

</Field>

</label>

</p>

<button type="submit" disabled={props.isValid==false}>Submit</button>

</Form>

</div>

)}

</Formik>

);

}

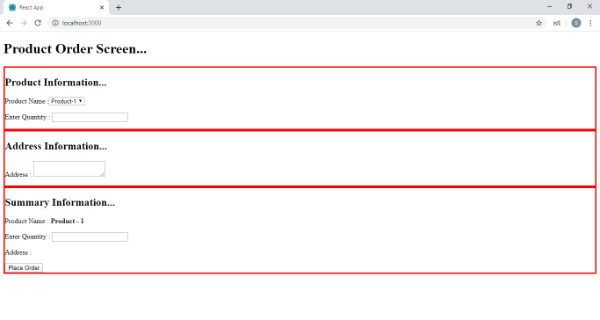
const element=<EmployeeComponent></EmployeeComponent>

ReactDOM.render(element,document.getElementById("root"));

# Lifting State Up

Lets assume that we are doing online shopping using one of the e-commerce website or through a mobile app. Assuming you want to Order that Product.

Now this is the Page which is being displayed to you.



If we see this Web Page, it has three sections.

1. Product Information Section where User can select what is the product user is Ordering and the Quantity number required.

2. Address Section where User can enter the Delivery Address.

3. Summary Section. We display the summary details of the previous two sections here.

When a User Enters the Product details and Address details we display that information in the summary section. But assuming that we have a textbox available where User Can change the product quantity even in the Summary Section. Because just to change the Product quantity, we don’t want User to navigate all the way back to Product Information section. If we change the Quantity in Summary section, we want that change to be displayed in Product Information section and vice versa like how it is happening in this Web Page.

Now question is how can we develop this in React.

What we can do here is When we enter the Product quantity either in the Product Information section or Summary Section, rather than maintaining local state in the respective components, we can create a shared state in the closest common ancestor component of these two components and that state can be accessed and updated by both of these components.

This Concept is Called as **Lifting State Up** in React.

This way the data can be stored in the state by the Parent Component and we also meet other important principle that There should be a single “**source of truth**” for any data that changes in a React application.

In Order to achieve this, we will create four Components.

They are

1. Order Component

2. ProductInformation Component

3. Address Component

4. Summary Component

Order Component will be the Parent Component and the rest are Child Components to Order Component.

Lets Create Four Different Component Classes. Starting with Order Component. Lets add a constructor and this constructor accepts properties as a parameter and pass it to the base class constructor using super. Lets write render method.

We will this class and create the rest of three classes in the same way.

Now lets go to Product Information Component’s render method.

In our Product Information component, we have to place a dropdown where user can select the Product he wants and a textbox where he can enter the quantity.

return (

<div style={{border:'3px solid red;'}}>

<h2>Order Information...</h2>

<p>

<label>Product Name :

<select>

<option value="Product-1">Product-1</option>

<option value="Product-2">Product-2</option>

<option value="Product-3">Product-3</option>

</select> </label>

</p>

<p>

<label>Enter Quantity : <input type="text"></input></label>

</p>

</div>

);

The product name what user selects and quantity data what user enters here should be accessible in the summary component.

If user makes any change for the quantity in summary component, that change should be displayed in this component. So we are going to store this input details in the Parent Component that is Order Component instead of storing it in the Current Component. This is where we are going to lift the state up. The value of this quantity input comes from the properties which are passed to this Product Component from Order Component. Order Component can instruct both the components to have values that are consistent with each other.

So we write **value={this.props.quantity}**

Whenever user makes a change to this input, we will pass that change to the Parent Component through properties. Lets add onChange attribute to this input and call a function where there is a change. I name that function as handleChange.

**onChange={this.handleChange}**

Now lets define handleChange function and with in this function, Assuming that ProductInfo Component receives a property called as onQuantityChange, and to this property we will pass the quantity value as an input.

**this.props.onQuantityChange(e.target.value);**

With this we are done with doing the changes required for our ProductInfo component.

Address Component will have the inputs required to provide our address. So assuming that we will place a multiline textbox using which we will enter the address information, whenever user enters the address we will store this address information as well in the Parent Component. That means whatever we have done for quantity input data in the ProductInfo component, we will repeat the same for Address.

Next lets move on to summary Component class.

In this component we will display the product name, product quantity and address details. For this demonstration, we will keep our focus on accessing quantity and address from the Parent Component. So I keep product name static.

Now lets display the Address first  by accessing it using **{this.props.address}**

Now the quantity should be displayed in a textbox. And this should display the quantity value what user has entered in the Product Info section and any change on this input should reflect in the ProductInfo section.

So we repeat the same thing we have done for this input in the ProductInfo section. So I paste the same Code we have written for Quantity field in the ProductInfo Component and we handle the onChange event as well. And on change we update this value to the Parent Component that is Order Component.

Then comes the most important class that is our Parent Component which is OrderComponent.

Lets create a state object in the constructor of this class and to this state object we will add two properties. One is to store quantity value and the other one is to store address value. We initialize both of them to empty.

Lets implement two functions, one will be called when there is a change in the quantity and the other will be called when there is a change in the address. In both of these functions, we will update the respective details to our state object using setState method.

orderInfoChanged=(val)=>{

this.setState({quantity:val});

}

addressChanged=(val)=>{

this.setState({address:val});

}

Lets go to render method and we will call these three components one by one. Lets start with ProductInfo component. To this Component we have to pass the quantity and onQuantityChange as properties. Quantity will get the value from the state object quantity property and onQuantityChange will hold the orderInfoChaned function.

<ProductInfoComponent quantity={this.state.quantity}

onQuantityChange={this.orderInfoChanged}></ProductInfoComponent >

Lets repeat the same for AddressComponent and Summary Component. To summary component we have to pass both quantity and address through properties.

Now lets call this OrderComponent and render this Component to our root container.

Save these changes, navigate to the browser. As we can see we have three sections. Fill the quantity and address, we can see that these changes are reflected into summary component. Now lets go to summary component, change the quantity. We can see that it gets updated into Product Info Component as well. For both Summary Component and Product Info Component, the quantity value is coming from Parent Component that is Order Component.

Both Product Info Component and Summary Component are sharing the state through Parent Component. This is what we call as Lifting state up.

import React from 'react';

import ReactDOM from 'react-dom';

class OrderComponent extends React.Component{

constructor(props) {

super(props);

this.state={quantity:'', address:''};

}

orderInfoChanged=(val)=>{

this.setState({quantity:val});

}

addressChanged=(val)=>{

this.setState({address:val});

}

render(){

return(

<div>

<h1>Welcome to Product Order Screen...</h1>

<OrderInfoComponent quantity={this.state.quantity}

onQuantityChange={this.orderInfoChanged}></OrderInfoComponent>

<AddressComponent address={this.state.address}

onAddressChange={this.addressChanged}></AddressComponent>

<SummaryComponent quantity={this.state.quantity} address={this.state.address}

onQuantityChange={this.orderInfoChanged}></SummaryComponent>

</div>

);

}

}

class OrderInfoComponent extends React.Component{

constructor(props) {

super(props);

}

handleChange=e=>{

this.props.onQuantityChange(e.target.value);

}

render(){

return (

<div style={{border:'3px solid red'}}>

<h2>Order Information...</h2>

<p>

<label>Product Name :

<select>

<option value="Product-1">Product-1</option>

<option value="Product-2">Product-2</option>

<option value="Product-3">Product-3</option>

</select> </label>

</p>

<p>

<label>Enter Quantity : <input type="text" value={this.props.quantity}

onChange={this.handleChange}></input></label>

</p>

</div>

);

}

}

class AddressComponent extends React.Component{

constructor(props) {

super(props);

}

handleChange=e=>{

this.props.onAddressChange(e.target.value);

};

render(){

return (

<div style={{border:'3px solid red'}}>

<h2>Address Information...</h2>

<p>

<label>Address : <textarea value={this.props.address} onChange={this.handleChange}></textarea></label>

</p>

</div>

);

}

}

class SummaryComponent extends React.Component{

constructor(props) {

super(props);

}

handleChange=(e)=>{

this.props.onQuantityChange(e.target.value);

}

render(){

return (

<div style={{border:'3px solid red'}}>

<h2>Summary...</h2>

<p>

<label>Product Name : <b>Product - 1</b></label>

</p>

<p>

<label>Product Quantity : <input type="text" value={this.props.quantity} onChange={this.handleChange}></input></label>

</p>

<p>

<label>Address : <b>{this.props.address}</b></label>

</p>

<button>Place Order</button>

</div>

);

}

}

const element=<OrderComponent></OrderComponent>

ReactDOM.render(element,document.getElementById("root"));

# Error Boundaries

In this article We will understand about a Concept in React called as **Error Boundaries.**

To understand things better, I am accessing our react application which we have developed in our last session from the browser. This is the Product Ordering screen which we have developed and it has three sections and they are Product Information section, Address Section and Summary Section. Now our new requirement is that in the Address Section, we have to display the Current user’s Preferred Address list which user has entered previously.

We will create a New Class called as UserPreferredAddressList component class and we will implement render method.

class UserPreferredAddressList extends React.Component{

constructor(props){

super(props);

}

render(){

return (

<div>

<h2>Your Existing Addresses...</h2>

<p>

Office<br></br>

Marathahalli, Bangalore-560037

</p>

</div>

);

}

}

Lets Call this Component from our Address Component Class. Lets save these changes. Navigate to the browser. We can see that Preferred Address List is displayed in the Address Section.

Now assuming that we are having some problems to load the User Preferred Address List and that code is throwing us an error. Then what we expect the application to do is it should show us a message that we are not able to Load your preferrences. Please Enter the Address and Continue Ordering the Product.

But that will not happen by default.

Lets go to UserPreferredAddressList class and throw an error from the rendered method.

throw new Error("Not able to Fetch the Addresses at this moment");

Lets save this Change. Navigate to the browser. We can see that Error is displayed and we can see the Complete Error Stack in the browser.

Now how do we handle this error and show the Custom message to the user that we are not able to load the Preferred Address List now and please Enter the Address and Continue placing your Order.

This is where we will make use of Error Boundaries in React.

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods **static getDerivedStateFromError()** or **componentDidCatch()**.

We Use static getDerivedStateFromError() to render a fallback UI after an error has been thrown.

We Use componentDidCatch() to log error information.

Lets see how we can create Error Boundary. We will create a Component Class and extend it from React.Component Class. Lets add Constructor and make a call to base class constructor.

Lets create a state object and add a property called hasError to this object. Lets implement both life cycle methods.

One method is static getDerivedStateFromError, we use this method to update our state object so that the next render will display the Custom UI instead of the error.

The second method is componentDidCatch, we can use this method to log the error details either to Console or to any other service. This componentDidCatch accepts two arguments. One is the error object which contains the error information and the other one is errorInfo which contains the component stack. That component stack gives us parent component details from where the call is made to the respective component.

Now lets implement render method. In render method, we will check our hasError property and if it has the error, we will return a div which will display the custom error message. If there is no error, we will return its children as is.

class CustomErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: null };

}

static getDerivedStateFromError(error) {

return { hasError: true };

}

componentDidCatch(error, errorInfo) {

console.log(error);

console.log(errorInfo);

}

render() {

if (this.state.hasError) {

// Error path

return (

<div>

<h2>We are having Problems to Load your Preferred Addresses. Please Select...</h2>

</div>

);

}

// Normally, just render children

return this.props.children;

}

}

Now lets call our UserPreferredAddressList Component with in the boundaries of this Error Boundary Class we have created.

Save these changes. Navigate to the browser. We can see that we get the Custom Error is displayed for us and the rest of the application works as it is. We have logged our error and errorInfo objects to the Console window. Lets open the developer tools, navigate to the console tab and we can see that both error and errorInfo object values. As we can see that error displays the error information and errorInfo shows us the component stack. It shows us that this error occurred from the Order Component as that is our Parent Component.

Error boundaries do not catch errors inside event handlers.

React doesn’t need error boundaries to recover from errors in event handlers. Unlike the render method and lifecycle methods, the event handlers don’t happen during rendering. So if they throw, React still knows what to display on the screen.

If you need to catch an error inside event handler, use the regular JavaScript try / catch statement

Error boundaries do not catch errors also for:

•  Any Asynchronous code we write (e.g. setTimeout )

• For Server side rendering code

• For Errors thrown in the error boundary component class itself (rather than its children)

Error boundaries works like a JavaScript catch {} block, but for components. Only class components can be error boundaries.

The granularity of error boundaries is up to us. We may wrap top-level component with in the scope of Error boundary or We may wrap a specific component with in the scope of Error boundary.

import React from 'react';

import ReactDOM from 'react-dom';

class OrderComponent extends React.Component{

constructor(props){

super(props);

this.state={quantity:'',address:''};

}

orderInfoChanged=val=>{

this.setState({quantity:val});

}

addressChanged=val=>{

this.setState({address:val});

}

render(){

return(

<div>

<h1>Product Order Screen...</h1>

<ProductInformationComponent quantity={this.state.quantity}

onQuantityChange={this.orderInfoChanged}></ProductInformationComponent>

<AddressComponent address={this.state.address}

onAddressChange={this.addressChanged}></AddressComponent>

<SummaryComponent quantity={this.state.quantity} address={this.state.address}

onQuantityChange={this.orderInfoChanged}></SummaryComponent>

</div>

);

}

}

class ProductInformationComponent extends React.Component{

constructor(props){

super(props);

}

handleChange=e=>{

this.props.onQuantityChange(e.target.value);

}

render(){

return (

<div style={{border:'3px solid red'}}>

<h2>Product Information...</h2>

<p>

<label>Product Name :

<select>

<option value="Product-1">Product-1</option>

<option value="Product-2">Product-2</option>

<option value="Product-3">Product-3</option>

</select> </label>

</p>

<p>

<label>Enter Quantity : <input type="text" value={this.props.quantity}

onChange={this.handleChange} ></input></label>

</p>

</div>

);

}

}

class AddressComponent extends React.Component{

constructor(props){

super(props);

}

handleChange=e=>{

this.props.onAddressChange(e.target.value);

};

render(){

return (

<div style={{border:'3px solid red'}}>

<h2>Address Information...</h2>

<p>

<label>Address : <textarea value={this.props.address}

onChange={this.handleChange}></textarea></label>

</p>

<CustomErrorBoundary>

<UserPreferredAddressList/>

</CustomErrorBoundary>

</div>

);

}

}

class CustomErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: null };

}

static getDerivedStateFromError(error) {

return { hasError: true };

}

componentDidCatch(error, errorInfo) {

console.log(error);

console.log(errorInfo);

}

render() {

if (this.state.hasError) {

// Error path

return (

<div>

<h2>We are having Problems to Load your Preferred Addresses. Please Select...</h2>

</div>

);

}

// Normally, just render children

return this.props.children;

}

}

class UserPreferredAddressList extends React.Component{

constructor(props){

super(props);

}

render(){

throw new Error("Not able to Fetch the Addresses at this moment");

return (

<div>

<h2>Your Existing Addresses...</h2>

<p>

Office<br></br>

Marathahalli, Bangalore-560037

</p>

</div>

);

}

}

class SummaryComponent extends React.Component{

constructor(props){

super(props);

}

handleChange=e=>{

this.props.onQuantityChange(e.target.value);

}

render(){

return (

<div style={{border:'3px solid red'}}>

<h2>Summary Information...</h2>

<p>

<label>Product Name : <b>Product - 1</b></label>

</p>

<p>

<label>Enter Quantity : <input type="text" value={this.props.quantity}

onChange={this.handleChange} ></input></label>

</p>

<p>

<label>Address : <b>{this.props.address}</b></label>

</p>

<button>Place Order</button>

</div>

)

}

}

const element=<OrderComponent></OrderComponent>

ReactDOM.render(element,document.getElementById("root"));

# Fragments

In this article We will understand about **Fragments** in React.

We have seen how do we create components in React and how do we return elements from the Component. But any component we create in React, can only return one element. If we look at the code what we have developed in our previous video, if we observe all the components are returning one div container.

Now what will happen if We return multiple elements from one Component. React will throw us an error. But it is very common scenario we face in our day to day programming life that we want to return multiple elements from one component.

It might be creating a component which is responsible for iterating through a list and returning multiple table row elements or iterating through an object and returning multiple td elements or returning multiple div containers.

Lets have a look at our Order Component Class. We have created the outer div just because our Order Component class wants to return multiple elements from the render method. Now what if we don’t want to enclose our ProductInfo Component, Address Info Component and Summary Component inside div.

Lets take another example. Now lets go to our Error Boundary class we have created, lets assume that from render method we want to return two div containers. One Container returns the contents of the error object and the other container returns the contents of the errorInfo object which shows us the component stack.

Lets the existing div contents and paste it again. We want to return two div containers. This will throw us the error. Now we can solve this error is by creating one parent div and we can place these two div containers as child elements inside that parent div.

But keep adding too many unnecessary DOM elements makes the DOM heavy, that may have an impact on the performance and will also introduce UI issues like styling, alignment.

In order to avoid adding new DOM elements, React introduces a concept called as Fragments using which we can return multiple elements from one component without creating additional DOM nodes.

Now lets add a new Element called as React.Fragment and place the two div containers with in the scope of this.

<React.Fragment>

<div>

<h2>We are having Problems to Load your Preferrences now.</h2>

</div>

<div>

<h2>We are having Problems to Load your Preferrences now.</h2>

</div>

</React.Fragment>

Save these changes. Navigate to the browser. We can see the output.

Now lets remove the additional div we have created in the Order Component render method as well by using Fragments. Before we go further, lets open developer tools, inspect our Product Information component contents, we can see that we have our root div container. Inside that we have another div and with in this child div we have our contents.

Lets replace the inner div using Fragments.

There is a new, shorter syntax we can use for declaring fragments. It looks like empty tags.

<>

<h1>Product Order Screen...</h1>

<ProductInformationComponent quantity={this.state.quantity}

onQuantityChange={this.orderInfoChanged}></ProductInformationComponent>

<AddressComponent address={this.state.address}

onAddressChange={this.addressChanged}></AddressComponent>

<SummaryComponent quantity={this.state.quantity} address={this.state.address}

onQuantityChange={this.orderInfoChanged}></SummaryComponent>

</>

Lets save these changes. Navigate to the browser. open developer tools, inspect our Product Information component contents, we can see that we have our root div container. Inside the root container we can see we have our contents directly.

**Fragment only supports one attribute**. That is the key attribute, used when mapping a collection to an array of components.

In the future, React may add support for additional attributes, such as event handlers.

# Refs and the DOM

In this article We will understand about Refs in React.

Refs provide a way to access DOM nodes or React elements created in the render method.

We have discussed about using props and state to manage the components data in React. But we have few use cases where having a reference to an element helps us to develop better react applications.

We will discuss all these use cases programmatically to get better understanding. So we will start with the first Use case.

Use Case – 1:

https://www.pragimtech.com/blog/contribute/article_images/1320200421204305/Picture1.png

Now we want to implement this in our React Application. Here we have a textbox and one button to increment the value and one button to decrement the value. It could have been a slider as well.

Lets open index.js file from our demo-project.

Lets create a class called as QuantityIncrement which extends React.Component class. Add the Constructor.

Lets implement render method. In render method, I will place an alert with a simple text message. Return div container from render method. Inside the Div, lets place an input whose type is text and we will place a button next to the input.

Lets call a function named as IncrementQuantity on click of this button. Lets implement this function.

Now with in this function, we have to increment the quantity value each time when user clicks on the button.

Lets create a state object in our constructor, add a property called as quantity and initialize it to 0. We can assign this value to our input element.

Now with in IncrementQuantity function, lets try to increment the quantity value of our state object. Now call this QuantityIncrement component and render it to our root container.

Lets save these changes. Navigate to the browser. We have the alert and We can see a textbox followed by a button.

Now when we click on the button, we get the alert again and the value in the textbox gets incremented.

But this has two problems right now.

First One is for every button click, our component is getting rendered again. Resulting we are getting the alert message. We don’t want to render our component for every button click.

The second one is now our textbox has become read only. Now if we focus on textbox and if we try to enter the custom value, we will not be able to do that. That is because our textbox is ready only.

Open developer tools and we can spot this error.

React is expecting us to handle onChange event on that input element.

Now how do we solve these two issues. This is where we will make use of **refs** in react.

We will create a reference object and we will assign it our input element. Refs are created when a component is rendered and can be defined either in the componentDidMount() or in the constructor().

Now lets go to the constructor. Refs are created using **React.createRef()** method. and now we attach this to our input element using the ref attribute.

On the button click, we will access this ref instance and increment the value.

When a ref is passed to an element in render, a reference to the element becomes accessible using current attribute of the ref.

• When the ref attribute is used on an HTML element, the ref created in the constructor with React.createRef() receives the underlying DOM element as its current property.

• When the ref attribute is used on a custom class component, the ref object receives the instance of the component as its current and through we can access the props and the methods of that component.

Now lets go to incrementQuantity function, remove the line of code we have written earlier.

We say **this.quantityRef.current**. and we access the input element value using value property. And we will increment this.

Save these changes, navigate to the browser and we can test the functionality.

As one can see we don’t get alert each time when we click on the button and we can go and enter value in the textbox and from there the increment can happen as well.

**Use Case – 2:**

Lets create a LogIn Component , I have the code handy and pasting it here.

class LogIn extends React.Component{

constructor(props){

super(props);

}

render(){

alert('Test Message');

return(

<div>

<h2>LogIn Screen...</h2>

<p>

<label>UserName : <input type="text"></input></label>

</p>

<p>

<label>PassWord : <input type="text"></input></label>

</p>

<button>LogIn</button>

</div>

);

}

}

Now lets render this Component instead of our Previous Component. Save these changes, navigate to the browser.

If we observe the output, by default focus is no-where. We want our UserName input should be focussed by default when the component is rendered.

We can set the focus to the input element by accessing its reference.

Lets go ahead and create the reference object and assign it to our UserName input.

With in the componentDidMount lifecycle method, we can set the focus using our reference object.

componentDidMount(){

this.usernameRef.current.focus();

}

Save these changes, navigate to the browser. We can see that UserName input field is focused by default. In the same way, when user clicks on LogIn button without entering either UserName or Password, we can set the focus to that respective input on the fly.

**Use Case – 3:**

Now lets say we want to Video Player Component using which we can play a video or pause a video.

I have one mp4 file placed in Assets folder which is inside src folder. This is the video We to play in our application.

Lets create a VideoPlayer Component class and add the constructor.

Lets implement render method,

Return a div container. We will place a video element inside and we will set the width and height.

Now we have to set the source for this Video. Now lets import the Video from the Assets folder.

Now assign that video to src attribute of our source element and set the type as video/mp4.

Now lets place two buttons using which we will control the Play and Pause of this Video.

Now lets create a reference to our video element. Assign this reference to our video element.

Now lets handle onClick events for both of these buttons.

Now lets create playvideo function and inside that call play method using the reference object we have created for video element and we do the same for pause Video.

this.myVideo.current.play();

 Lets call this Component and render it to our root container.

Save these changes. Navigate to the browser.

We can click on Play or Pause and accordingly video gets played or paused.

The other use cases where refs can be used are :

• Triggering imperative animations.

• Integrating with third-party DOM libraries.

Avoid using refs for anything that can be done declaratively.

We have to be very careful on when to use Refs , when to Use Props and when to Use State.

import ReactDOM from "react-dom";

import React, { Component } from "react";

import video from '../src/assets/Draft\_1.mp4';

class QuantityIncrement extends React.Component{

constructor(props){

super(props);

this.quantityRef=React.createRef();

}

incrementQuantity=()=>{

this.quantityRef.current.value++;

}

render(){

alert('Text Message');

return(

<div>

<p>

<label>Enter Quantity : <input type="text" ref={this.quantityRef}></input>

<button onClick={this.incrementQuantity}>+</button>

</label>

</p>

</div>

)

}

}

class LogIn extends React.Component{

constructor(props){

super(props);

this.userNameRef=React.createRef();

}

componentDidMount(){

this.userNameRef.current.focus();

}

render(){

return(

<div>

<h2>LogIn Screen...</h2>

<p>

<label>UserName : <input type="text" ref={this.userNameRef}></input></label>

</p>

<p>

<label>PassWord : <input type="text"></input></label>

</p>

<button>LogIn</button>

</div>

);

}

}

class VideoPlayer extends React.Component{

constructor(props){

super(props);

this.videoRef=React.createRef();

}

playVideo=()=>{

this.videoRef.current.play();

}

pauseVideo=()=>{

this.videoRef.current.pause();

}

render(){

return(

<div>

<video ref={this.videoRef} width="300" height="200" controls>

<source src={video} type="video/mp4"></source>

</video>

<div>

<button onClick={this.playVideo}>Play</button>

<button onClick={this.pauseVideo}>Pause</button>

</div>

</div>

);

}

}

const element=<VideoPlayer></VideoPlayer>

ReactDOM.render(element,document.getElementById("root"));

# Refs Part 2

In our last article, we have seen how do we apply reference to an element. Remember that element can be a HTML element or custom react element. Remember that all the reference objects we have created were for HTML elements, but we can pass the reference to a Custom React Elements as well provided they are Class Components.

Lets Open index.js file from our demo-project.

Now assuming that we are developing an application for an Elevator Manufacturing Company. Lets say we have to develop two components as part of doing the application development. One is Elevator Component and Summary Component. I have the Code handy and pasting it here

class Elevator extends React.Component{

constructor(props){

super(props);

this.elevatorRef=React.createRef();

}

render(){

return(

<div>

<h2>Welcome to Elevator Ordering Screen...</h2>

<p>

<label>Elevator Name : <input ref={this.elevatorRef} type="text"></input></label>

</p>

<p>

<label>Elevator Speed : <input type="text"></input></label>

</p>

<p>

<label>Elevator Load : <input type="text"></input></label>

</p>

<Summary></Summary>

</div>

);

}

}

class Summary extends React.Component{

constructor(props){

super(props);

}

render(){

return (

<div>

<h2>Summary Details...</h2>

<p>

<label>Elevator Name : <b>Name - 1</b></label>

</p>

<p>

<label>Elevator Speed : <b>10 m/s</b></label>

</p>

<p>

<label>Elevator Load : <b>550 Kg</b></label>

</p>

</div>

);

}

}

const element=<Elevator></Elevator>

ReactDOM.render(element,document.getElementById("root"));

Now customer has a requirement that when we click on a given field, focus should go back to the respective textbox.

So now when user Clicks on Elevator Name in  Summary Component, focus should go back to Elevator Name textbox in the Elevator Component. For doing that we have to access the reference of our Elevator Component input element in our Summary Component. This technique is Called as Forwarding refs.

Ref forwarding is a technique for automatically passing a ref through a component to one of its children.

Now we will pass the reference to summary component from Elevator Component. I give the property name as elevatorRef and to this we will pass our elevator name input element reference.

Lets trigger onClick event on our paragraph tag in our Summary component. Lets call a function called as focusElevatorName.

Now lets implement this function. With in this function, lets access the reference using the properties and we can call the focus method.

Until now we have seen how to use refs in Class Component. Now lets see how can we handle refs in function components.

I have a testComponent function handy and pasting it here.

function testComponent(){

function handleClick() {

}

return (

<div>

<input type="text" />

<input type="button" value="Focus the text input" onClick={handleClick}/>

</div>

);

}

Now lets declare a Variable and assign it to null. Call the focus method on that variable inside handleClick function.

Now assign the input element to that variable we have created using ref. But here if we observe, we are using a Callback.

Now Call this function component and render it to our root container.

Save these changes, navigate to the browser. On Click of the button we can see that textbox is focused.

We have seen how do we use ref for an input element inside a function Component. Remember that we can,  use the ref attribute inside a function component as long as you refer to a DOM element or a class component.

Refs cannot be attached to function components that is because function components do not have instances so you can’t reference them.

If we want to allow people to take a ref to our function component, you can use **forwardRef** method in React, or you can convert the component to a class.

Lets create a demo Function Component using **React. forwardRef**

Method and we will pass an arrow function and this function can receive the properties which will be passed to this component as one parameter and the ref as another parameter.

Lets return a button from this Component. Now call a function on Click of this button and we will implement this function.

With in this function, we will call the focus method using the reference which is being passed.

Now lets call this function component from our Elevator Component.

Now call this Elevator Component and render it to our root container.

Save these changes, navigate to the browser. Click on the button, we can see that the elevator name textbox is focused.

import ReactDOM from "react-dom";

import React, { Component } from "react";

const DemoComponent=React.forwardRef((props,ref)=>{

function testClick(){

ref.current.focus();

}

return(

<button onClick={testClick}>Click</button>

)

});

class Elevator extends React.Component{

constructor(props){

super(props);

this.elevatorRef=React.createRef();

}

render(){

return(

<div>

<h2>Welcome to Elevator Ordering Screen...</h2>

<p>

<label>Elevator Name : <input ref={this.elevatorRef} type="text"></input></label>

</p>

<p>

<label>Elevator Speed : <input type="text"></input></label>

</p>

<p>

<label>Elevator Load : <input type="text"></input></label>

</p>

<Summary innerRef={this.elevatorRef}></Summary>

<DemoComponent ref={this.elevatorRef}></DemoComponent>

</div>

);

}

}

class Summary extends React.Component{

constructor(props){

super(props);

}

focusInput=()=>{

this.props.innerRef.current.focus();

}

render(){

return (

<div>

<h2>Summary Details...</h2>

<p onClick={this.focusInput}>

<label>Elevator Name : <b>Name - 1</b></label>

</p>

<p>

<label>Elevator Speed : <b>10 m/s</b></label>

</p>

<p>

<label>Elevator Load : <b>550 Kg</b></label>

</p>

</div>

);

}

}

function testComponent(){

let testRef=null;

function handleClick() {

testRef.focus();

}

return (

<div>

<input type="text" ref={e=>testRef=e} />

<input type="button" value="Focus the text input" onClick={handleClick}/>

</div>

);

}

const element=<Elevator></Elevator>

ReactDOM.render(element,document.getElementById("root"));

# Higher Order Components

A **Higher-Order component** (HOC) is an advanced technique in React for reusing component logic.

Higher Order Component is a function that takes a component as input and returns a new component.

Higher Order Components promote Code Reusability. We all know how important it is to develop reusable code.

Lets look at one Example. Assuming that we are developing an Admin Dashboard which will show Employee Data, Department Data and Project Data in the form of Reports.

We will have our REST API which will be giving us the data and the react application should display this data by fetching from the API.

We will use the Same REST API we have developed in our Previous Videos, Lets Open the ASP.NET Web API we have created using Visual Studio. As we can see here that this Project is having two different API Controllers, One returns the Employee Data and the other one returns the Department Data. Lets run this API Project.

Now lets Open Index.js file from our Demo Project using Visual Studio Code.

Here we will create three Component Classes. AdminDashboard, EmployeeReports and Department Reports. EmployeeReports Component will have the code to call the API, get the list of Employees and display them.

DepartmentReports Component will have the code to call the API, get the list of Departments  and display them.

AdminDashboard will be calling these two components and display the Reports.

And finally we will call the AdminDashboard Component and we will render it to our root container.

I have the Code handy and I am pasting it here. As this has been discussed already in our previous videos.

class EmployeeReports extends React.Component {

constructor(props) {

super(props);

this.state = {

employees: []

};

}

componentDidMount() {

fetch("https://localhost:44306/api/Employee")

.then(res => res.json())

.then(

(result) => {

this.setState({

employees: result

});

}

);

}

render() {

return (

<div>

<h2>Employees Data...</h2>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

<th>Location</th>

<th>Salary</th>

</tr>

</thead>

<tbody>

{this.state.employees.map(emp => (

<tr key={emp.Id}>

<td>{emp.Id}</td>

<td>{emp.Name}</td>

<td>{emp.Location}</td>

<td>{emp.Salary}</td>

</tr>

))}

</tbody>

</table>

</div>

);

}

}

class DeptReports extends React.Component {

constructor(props) {

super(props);

this.state = {

dept: []

};

}

componentDidMount() {

fetch("https://localhost:44306/api/Dept")

.then(res => res.json())

.then(

(result) => {

this.setState({

dept: result

});

}

);

}

render() {

return (

<div>

<h2>Dept Data...</h2>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

<th>Revenue</th>

</tr>

</thead>

<tbody>

{this.state.dept.map(d => (

<tr key={d.Id}>

<td>{d.Id}</td>

<td>{d.Name}</td>

<td>{d.Revenue}</td>

</tr>

))}

</tbody>

</table>

</div>

);

}

}

class AdminDashboard extends React.Component{

constructor(props){

super(props);

}

render(){

return (

<React.Fragment>

<EmployeeReports></EmployeeReports>

<DeptReports></DeptReports>

</React.Fragment>

);

}

}

const element=<AdminDashboard></AdminDashboard>

ReactDOM.render(element,document.getElementById("root"));

Now lets save these changes, navigate to the browser. We can see that both employee reports and department reports are displayed.

Now if we look at our EmployeeReports Component Class and DepartmentReports Component class, though they are sending a request to different Rest API, though they have different sets of columns and data that is getting rendered, on a high level they do the same job. Send API request, get the reports data and display the reports. We have to write the code on the similar lines even for showing Project Reports. Why cant we reuse the code instead of repeating the code again and again. This is where Higher Order Components comes into picture.

Remember that Higher-order component is a function that takes a component and returns a new component

Lets go ahead and create a function which accepts two inputs. First one is the Component and the second one is the input using which we will pass the information like the url and how many columns we want in the reports and any other component specific data.

As we know that Higher Order Component takes a Component as an input and returns a new Component, lets return a Component class

And with in class lets create a constructor and add one state object. To this state object add properties like url, columns and data. We initialize these properties by reading the data from the inputData parameter.

Lets implement ComponentDidMount method in which we will have the code to send the API request. I have the code and pasting it here.

We are sending the request and when we get the response we are updating our state object.

Now lets implement render method. With in this render method, we call another component to which we will pass this state data through a property and that component is responsible for displaying the data.

Now lets create this Data Component. Lets implement the render method.

Lets return a div container. Lets display the header data using h2 tag. We will access the header value through data property which is passed to this component.

Lets add a table, and the header columns are dependent on the columns which are passed through properties. Lets navigate through the columns array using map function and we will add the header.

Add table body and we have to display the data based on the number of records in the data which is passed to this component through properties. Lets loop through the data and for every row, we have to add the respective number of columns.

Lets create one Reports Component class and this component is going to be used as input component to our Higher Order Component.

Lets return div from the render method of this class.

Now we will create all the report component classes which are required. First lets start with EmployeeReports,

So we say const EmployeeReports=reportsHOC() and to this function, we will pass our Reports Component as input and we will pass another object which will contain the url, columns and header.

The Higher Order Component returns us a Component and we have assigned it to EmployeeReports. Now lets see how easy it becomes to generate DepartmentReports. We will call the reportsHOC function again and we will pass the other details through an Object.

Now we can repeat the same for any number of Report Components we want to generate.

Lets save these changes, navigate to the browser and we can see that both the Reports are being displayed.

Using HOC’s are common in any Enterprise application we develop using React.

import ReactDOM from "react-dom";

import React, { Component } from "react";

function reportsHOC(InputComponent, inputData) {

return class extends React.Component {

constructor(props) {

super(props);

this.state = {

data:[],

columns:inputData.columns,

header:inputData.header

};

}

componentDidMount() {

fetch(selectData.Url)

.then(res => res.json())

.then(

(result) => {

this.setState({

data: result

});

}

);

}

render() {

return (

<Data data={this.state}></Data>

);

}

};

}

class Data extends React.Component{

constructor(props){

super(props);

//alert(JSON.stringify(props));

}

render(){

return (

<div>

<h2>{this.props.data.header}...</h2>

<table>

<thead>

<tr>

{this.props.data.columns.map(c => (

<th>{c}</th>

))}

</tr>

</thead>

<tbody>

{this.props.data.data.map(emp => (

<tr key={emp.Id}>

{this.props.data.columns.map(c => (

<td>{emp[c]}</td>

))}

</tr>

))}

</tbody>

</table>

</div>

);

}

}

class Reports extends React.Component {

constructor(props) {

super(props);

}

render() {

return(

<div></div>

);

}

}

const EmployeeReports=reportsHOC(Employee,

{Url:'https://localhost:44306/api/Employee',

columns:['Id','Name','Location','Salary'],header:'Employee Data'});

const DeptReports=reportsHOC(Employee,{Url:'https://localhost:44306/api/Dept',

columns:['Id','Name','Revenue'],header:'Dept Data'});

class AdminDashboard extends React.Component{

constructor(props){

super(props);

}

render(){

return (

<React.Fragment>

<EmployeeReports></EmployeeReports>

<DeptReports></DeptReports>

</React.Fragment>

);

}

}

const element=<AdminDashboard></AdminDashboard>

ReactDOM.render(element,document.getElementById("root"));

# Portals

In this article, we will discuss about **Portals**in React.

**Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.**

Normally, when you return an element from a component’s render method, it’s mounted into the DOM as a child of the nearest parent node.

However, sometimes it’s useful to insert a child into a different location in the DOM.

Lets understand this with an example. Lets open the index.js file from our demo-project.

I have copied the Employee Component from our last session and Pasting it here. This Component displays the list of employees. And we are rendering this element to our root container.

class Employee extends React.Component {

constructor(props) {

super(props);

this.state = {

employees: []

};

}

componentDidMount() {

fetch("https://localhost:44306/api/Employee")

.then(res => res.json())

.then(

(result) => {

this.setState({

employees: result

});

}

);

}

render() {

return (

<div>

<h2>Employees Data...</h2>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

<th>Location</th>

<th>Salary</th>

</tr>

</thead>

<tbody>

{this.state.employees.map(emp => (

<tr key={emp.Id}>

<td>{emp.Id}</td>

<td>{emp.Name}</td>

<td>{emp.Location}</td>

<td>{emp.Salary}</td>

</tr>

))}

</tbody>

</table>

</div>

);

}

}

Lets Call this Component and render this Component to our root container.

Now if we look at the code, we are returning one div container from the render method. We have a table inside the div. Now this div contents will be placed inside a div whose id is root. We can observe the same in the browser. Save these changes, navigate to the browser. We can see the employees data is being displayed.

Now open developer tools, We can observe that the table is placed inside a div and this div is placed in the root div container.

Now our requirement is that we have to place edit button in every row of our table and when we click on the edit button, we want the employee data should be displayed in a Modal Popup.  That modal popup contents may be displayed in the center of our browser.

That means we have render an element outside of its parent’s document.

This is where we will make use of a Concept in React called as Portals.

Now lets see how we will display a modal popup outside of our div using Portal.

Lets create one EmployeeModal Component class which will display the Employee Details in edit Mode.

I have the code handy and I am pasting it here.

class EmployeeModal extends React.Component{

constructor(props){

super(props);

}

render(){

return(

<div>

<h2>Employee Details...</h2>

<p>

<label>Employee ID : <input type="text" value={this.props.employee.Id}></input></label>

</p>

<p>

<label>Employee Name : <input type="text" value={this.props.employee.Name}></input></label>

</p>

<p>

<label>Employee Location : <input type="text" value={this.props.employee.Location}></input></label>

</p>

<p>

<label>Employee Salary : <input type="text" value={this.props.employee.Salary}></input></label>

</p>

<input type="submit" value="Save"></input>

</div>

)

}

}

We are returning a div container in which we are displaying the employee data in the textboxes. We have a save button.

Now lets add a new column to our table and with in this column we will place one edit button.

We have display the modal pop up on click of the edit button. Lets Call a function named editEmployee on the click.

Lets add a new property to our state object using which we will track the show and hide of our modal.

Update this property value with in our openModal function based on its current state in our editEmployee function. That means if it is Opened now, we will close the Modal PopUp. We do the other way if not.

Lets create one modal component. To this we will pass the children that needs to rendered, onClose function and modal popup visibility status as parameters.

If the popus status is true, then we will display the the children of this modal pop up in a div but that div conainer should be rendered outside of the parent div. so we use a method called as ReactDOM.createPortal method and to this method we will pass the contents that needs to be displayed and the container. Lets give the DOM body as the container.

Lets add a close button. Lets apply a class to this div using which we are setting the some css property values. I have the css class code handy and I place it in index.html file.

Now lets call this modal component from our table. To this modal we will pass open and onClose as properties.

With in this Modal Component, we will call the EmployeeModal component and we will pass the employee data to this component class through property.

Save these changes. Navigate to the browser. Click on Edit button of any row and we can see that employee details are displayed in a Modal Popup. We are rendering the contents of the EmployeeModal Component Class outside of our root container using Portals in React.

Whenever the contents of specific component needs to be rendered outside of the current container, like showing tooltips, or  new window that is where we will make use of Portals in React.

Remember that Even though a portal can be anywhere in the DOM tree, it behaves like a normal React child in every other way. Features like context work exactly the same regardless of whether the child is a portal, as the portal still exists in the React tree regardless of position in the DOM tree.

This includes event bubbling. An event fired from inside a portal will propagate to ancestors in the containing React tree, even if those elements are not ancestors in the DOM tree.

import ReactDOM from "react-dom";

import React, { Component } from "react";

class Employee extends React.Component {

constructor(props) {

super(props);

this.state = {

employees: [],

showModal:false

};

}

componentDidMount() {

fetch("https://localhost:44306/api/Employee")

.then(res => res.json())

.then(

(result) => {

this.setState({

employees: result

});

}

);

}

editEmployee=()=>{

this.setState({showModal:!this.state.showModal});

}

render() {

return (

<div>

<h2>Employees Data...</h2>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

<th>Location</th>

<th>Salary</th>

<th>Actions</th>

</tr>

</thead>

<tbody>

{this.state.employees.map(emp => (

<tr key={emp.Id}>

<td>{emp.Id}</td>

<td>{emp.Name}</td>

<td>{emp.Location}</td>

<td>{emp.Salary}</td>

<td>

<button onClick={this.editEmployee}>Edit</button>

<Modal open={this.state.showModal} close={this.editEmployee}>

<EmployeeModal employee={emp}></EmployeeModal>

</Modal>

</td>

</tr>

))}

</tbody>

</table>

</div>

);

}

}

class Modal extends React.Component{

constructor(props){

super(props);

}

render(){

return(

this.props.open?ReactDOM.createPortal(

<div className="modal">

<button onClick={this.props.close}>X</button>

{this.props.children}

</div>,document.body):null

);

}

}

class EmployeeModal extends React.Component{

constructor(props){

super(props);

}

render(){

return(

<div>

<h2>Employee Details...</h2>

<p>

<label>Employee ID : <input type="text" value={this.props.employee.Id}></input></label>

</p>

<p>

<label>Employee Name : <input type="text" value={this.props.employee.Name}></input></label>

</p>

<p>

<label>Employee Location : <input type="text" value={this.props.employee.Location}></input></label>

</p>

<p>

<label>Employee Salary : <input type="text" value={this.props.employee.Salary}></input></label>

</p>

<input type="submit" value="Save"></input>

</div>

)

}

}

const element=<Employee></Employee>

ReactDOM.render(element,document.getElementById("root"));

<style>

.modal {

background-color: whitesmoke;

position: fixed;

height: 50%;

width: 50%;

top: 25;

left: 25%;

align-items: center;

}

.modal button{

position: absolute;

top: 5px;

right: 3px;

font-size: 1rem;

font-weight: bold;

}</style>

# Profiler

The **Profiler** measures how often a React application renders and what the “cost” of rendering is. Its purpose is to help identify parts of an application that are slow so that we can work on optimizing them.

Lets take a look at one example. Assuming that we are developing a Reports Dashboard component for one of the leading bank in the market.

This Reports Dashboard Component will be using multiple components in it. They are

1. NewAccountsReports

2. LoansRepaymentReports

3. CreditCardsReports

Assuming that we have developed all these components and our ReportsDashboard Component is ready. We can perform various operations on these reports like entering some inputs to generate reports, triggering few events.

We have discussed in our previous videos that our components gets re-rendered for various operations we perform on these components. Assuming that for some reason the ReportsDashboard Component is not very responsive.

Now we have to identify which component is getting re-rendered how many times and the amount of time it is taking for re-rendering the Tree. so that it will help us to fix the Problem.

This is where we will make use of Profiler in React.

Lets open index.js file from our demo-project. We will create three different components. They are NewAccountsReports, LoanRepaymentReports and ReportsDashboard Component Classes.

ReportsDashboard Component Class will be calling both the Components.

I have the Code handy and I am pasting it here.

class NewAccountReports extends React.Component{

constructor(props){

super(props);

this.state={

FromDate:'',

ToDate:''

};

}

handleChange=e=>{

let name=e.target.name;

let value=e.target.value;

this.setState({

[name]:value

});

}

render(){

return(

<div>

<h2>Welcome to New Accounts Reports Component...</h2>

<p>

<label>From Date : <input type="text" name="FromDate"

onChange={this.handleChange} value={this.state.FromDate}></input></label>

</p>

<p>

<label>To Date : <input type="text" name="ToDate"

onChange={this.handleChange} value={this.state.ToDate}></input></label>

</p>

<input type="submit" value="Generate"></input>

</div>

)

}

}

class LoansRepaymentReports extends React.Component{

constructor(props){

super(props);

}

render(){

return(

<div>

<h2>Welcome to Loans Repayment Reports Component...</h2>

</div>

);

}

}

class ReportsDashboard extends React.Component{

constructor(props){

super(props);

}

render(){

return (

<React.Fragment>

<h2>Welcome to Reports Dashboard...</h2>

<NewAccountReports></NewAccountReports>

<LoansRepaymentReports></LoansRepaymentReports>

</React.Fragment>

);

}

}

const element=<ReportsDashboard></ReportsDashboard>

ReactDOM.render(element,document.getElementById("root"));

Lets save these changes, navigate to the browser and we can see the Output. As we perform various operations on these components, the components will be getting re-rendered. We want to Log all the details of how many times the respective component is getting re-rendered, how much time each update is taking, when is the start time and when is the commit time.

This is where we will make use of Profile in react. Lets see how do we do that.

A Profiler can be added anywhere in a React tree to measure the cost of rendering that part of the tree. It requires two props: an id (string) and an onRender callback (function) which React calls any time a component within the tree “commits” an update.

Now lets go and add the Profiler to our NewAccountsReports Component. Lets add two attributes. One is Id, Using Id we will track each profiler. And the other attribute is onRender and to this lets pass the function name

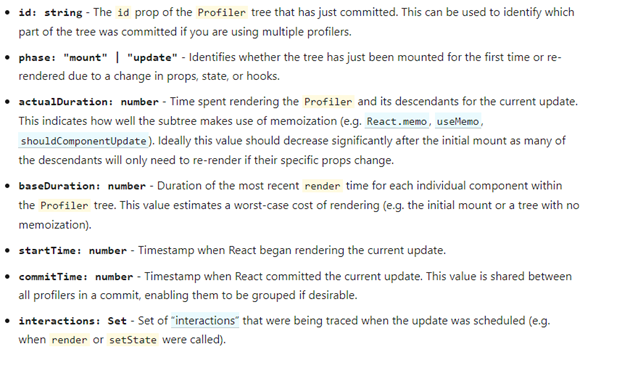
<Profiler id="newAccounts" onRender={this.callbackFunction}>

<NewAccountReports></NewAccountReports>

</Profiler>

React calls this callbackFunction any time a component within the profiled tree “commits” an update.

Lets go ahead and implement this callbackFunction. It receives parameters describing what was rendered and how long it took.



We can log these parameter values to our console.

callbackFunction=(id,phase,actualDuration,baseDuration, startTime,

commitTime, interaction)=>{

console.log('Id is : '+id+', Phase is : '+phase);

console.log('Actual Duration is : '+actualDuration+' and Base Duration is :'+

baseDuration);

}

Lets save these changes. Navigate to the browser. Open developer tools, we can find one set of logs in the console. That is the result of first Render and this phase is called as Mount. We can enter From Date value and we can see the logs again. This is the result of re-render. This phase is called as Update.

Multiple Profiler components can be used to measure different parts of an application.

We can apply the same Profiling for LoansRepaymentReports Component as well.

 Profiler components can also be nested to measure different components within the same subtree.

Although Profiler is a light-weight component, it should be used only when necessary; each use adds some CPU and memory overhead to an application.

Profiling adds some additional overhead, so it is **disabled in the production build**.

To opt into production profiling, React provides a special production build with profiling enabled. Read more about how to use this build at **fb.me/react-profiling**

# Render Props

In this article, we will discuss about**Render Props** in React.

The term “**render prop**” refers to a technique for sharing code between React components using a prop whose value is a function.

A component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic.

In our Previous Videos, we have seen how do we use properties to pass data to the components and we have also seen how do we assign a callback function as value to a Property of a Component.

Now we will be learning about Render Props in React. Lets take an example. In our React application, assume that we are displaying Departments list and Projects list.

We have the Departments Component which is getting the data from lets say a Web API and display the data.

We have the Projects Component which is getting the data from localStorage through a Proerty and displaying the Data.

We have an App Component which is Calling both these Components. I have the Code handy and pasting it here.

import ReactDOM from "react-dom";

import React, { Component, Profiler } from "react";

class Department extends React.Component{

constructor(props){

super(props);

this.state={

list:['Dev','Big Data','Mobility']

};

}

render(){

return(

<div>

<h2>Department List...</h2>

<ul>

{this.state.list.map(d=>(

<li>{d}</li>

))}

</ul>

</div>

);

}

}

class Project extends React.Component{

constructor(props){

super(props);

}

render(){

return(

<div>

<h2>Projects List...</h2>

<ul>

{this.props.list.map(d=>(

<li>{d}</li>

))}

</ul>

</div>

);

}

}

class Page extends React.Component{

constructor(props){

super(props);

}

render(){

return(

<React.Fragment>

<Department></Department>

<Project list={['P-1','P-2','P-3']}></Project>

</React.Fragment>

)

}

}

const element=<Page></Page>

ReactDOM.render(element,document.getElementById("root"));

If we observe both Departments Component and Projects Component, though they are getting the data from different sources and but the rendering pattern remains same. Now why cant we reuse the rendering code across multiple components. That way we will be making our code better and we will also be meeting Single Responsibility Principle as well.

We can make One Component Responsible to get the Data and we have another component which renders the data.

Now lets take out the rendering code which is common in both the Component classes.

Lets create a class called as DisplayData and extend it from React.Component class. Add a constructor. Lets implement render method.

Assuming that this component receives the list through properties and lets create ul, loop through the list.

class DisplayData extends React.Component{

constructor(props){

super(props);

}

render(){

return(

<ul>

{this.props.list.map(d=>(

<li>{d}</li>

))}

</ul>

);

}

}

Lets go to Department Component Class render method, take out the code which is displaying the department data and call a method called as render using the properties object and pass the list data. We are calling a Property called as Render as a function by passing the list.

We will do the same in Project Class as well.

<div>

<h2>Department List...</h2>

{this.props.render(this.state.list)}

</div>

Then lets go to Page class, We will slightly change the way how we call Department Component. To this Department Component, we will add a Property called as render and to that render property, lets pass a function. This function receieves the list and that list should be passed to DisplayData Component class which will render the list.

We will repeat the same for Project Component as well.

Save these changes, navigate to the browser. We can see the output. We are getting the same output but improved code reusability.

More concretely, a render prop is a function prop that a component uses to know what to render.

It’s important to remember that just because the pattern is called “render props” we don’t have to use a prop named render to use this pattern. In fact, any prop that is a function that a component uses to know what to render is technically a “render prop”.

Basically we are calling a function which is rendering the contents of Department Component.

The Concept of Render Props is considered as one of the Design Pattern while developing a React application.

To deepen our understanding about this concept, lets write one more example. I have copied our Employee Reports Component Class which we have created and pasting it here.

class EmployeeReports extends React.Component {

constructor(props) {

super(props);

this.state = {

employees: []

};

}

componentDidMount() {

fetch("https://localhost:44306/api/Employee")

.then(res => res.json())

.then(

(result) => {

this.setState({

employees: result

});

}

);

}

render() {

return (

<div>

<h2>Employees Data...</h2>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

<th>Location</th>

<th>Salary</th>

</tr>

</thead>

<tbody>

{this.state.employees.map(emp => (

<tr key={emp.Id}>

<td>{emp.Id}</td>

<td>{emp.Name}</td>

<td>{emp.Location}</td>

<td>{emp.Salary}</td>

</tr>

))}

</tbody>

</table>

</div>

);

}

}

Now lets create a new Component class called as DisplayEmployees. Add the constructor and we will keep the code of displaying the Employees in the render method.

Now lets go to EmployeeReports Component Class render method and call render property by passing employees list.

Now lets call the EmployeeReports Component, pass render property and assign a function. This function receives employees data and pass that to DisplayEmployees Component through a property and that Component renders the Employees Data.

**With Render Props:**

class DisplayEmployees extends React.Component{

constructor(props){

super(props);

}

render(){

return(

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

<th>Location</th>

<th>Salary</th>

</tr>

</thead>

<tbody>

{this.props.employees.map(emp => (

<tr key={emp.Id}>

<td>{emp.Id}</td>

<td>{emp.Name}</td>

<td>{emp.Location}</td>

<td>{emp.Salary}</td>

</tr>

))}

</tbody>

</table>

)

}

}

class EmployeeReports extends React.Component {

constructor(props) {

super(props);

this.state = {

employees: []

};

}

componentDidMount() {

fetch("https://localhost:44306/api/Employee")

.then(res => res.json())

.then(

(result) => {

this.setState({

employees: result

});

}

);

}

render() {

return (

<div>

<h2>Employees Data...</h2>

{this.props.content(this.state.employees)}

</div>

);

}

}

const element=<EmployeeReports content=

{(input)=><DisplayEmployees employees={input}></DisplayEmployees>}>

</EmployeeReports>

ReactDOM.render(element,document.getElementById("root"));

# Pure Components

React components let us split the UI into independent, reusable pieces, and think about each piece in isolation. React components can be defined by extending from **React.Component** Class or **React.PureComponent** Class.

We have created different Component classes until now and all these classes were extending React.Component Class.

There are few use cases where we can use React Pure Components.

Lets look at one example.

We are developing one application in which we are displaying the Reports. Let the reports be Employee Reports or Stock Market Reports.

For this assume that we have created GetReports Component Class.

This Component Class gets the Reports by Calling a Web API and in the UI, we will have one Reload button using which user can reload the Reports so that he will get Updated Reports Data.

Now we have another Component which will be used to show if there are changes to the Reports Data. Lets say for every 5 seconds, this Component sends a request to the Web API and get the flag which tells whether there are any changes are not. If there are any changes, we will show a notification to the User, so that using Reload button user can get the updated Reports.

Lets open Index.js file from our demo-project

Lets create two Component Classes. One is DetectChanges Component Class and Other one is Reports Component Class. I have the Code handy and Pasting it here.

class ChangeDetection extends React.Component {

constructor(props) {

super(props);

this.state = {

employeeCount:0

};

setInterval(() => {

fetch("https://localhost:44306/api/Employee")

.then(res => res.json())

.then(

(result) => {

this.setState({

employeeCount: result.length

});

}

);

}, 5000);

}

render() {

return (

<div>

<h2>Welcome to Pure Component Demonstration...</h2>

<p>

<label>Number of Employees are : <b>{this.state.employeeCount}</b></label>

</p>

</div>

);

}

}

class Reports extends React.Component {

constructor(props) {

super(props);

this.state = {

employees:[]

};

}

componentDidMount=() => {

this.getEmployees();

}

getEmployees() {

fetch("https://localhost:44306/api/Employee")

.then(res => res.json())

.then(

(result) => {

this.setState({

employees: result

});

}

);

}

loadEmployees=()=>{

this.getEmployees();

}

render() {

return (

<div>

<h2>Employees Data...</h2>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

<th>Location</th>

<th>Salary</th>

</tr>

</thead>

<tbody>

{this.state.employees.map(emp => (

<tr key={emp.Id}>

<td>{emp.Id}</td>

<td>{emp.Name}</td>

<td>{emp.Location}</td>

<td>{emp.Salary}</td>

</tr>

))}

</tbody>

</table>

<p>

<button onClick={this.loadEmployees}>Reload</button>

</p>

</div>

);

}

}

class App extends React.Component{

constructor(props){

super(props);

}

render(){

return(

<React.Fragment>

<ChangeDetection></ChangeDetection>

<Reports></Reports>

</React.Fragment>

)

}

}

const element=<App></App>

ReactDOM.render(element,document.getElementById("root"));

Save the Changes, Navigate to the Browser and we can see the Output.

We might feel that our code is working fine and we are getting the Results. But there are few issues with the code we have written here. Lets understand that.

For every 5 seconds, our changeDetection Component is sending a Web API request, getting the data and setting into the State Object.

 Each time when we set the data into our state object or when we assign the property values, the component gets re-rendered.

Right now we have five employees data that is being shown. We have not added any new record . so each time when the ChangeDetection Component sends the request to Web API, it gets the Count as 5 but what happens is as we are setting the Data into state object , though it is the same data that means the Employee Count is 5 but still our component gets re-rendered. Lets add an alert in the render method of ChangeDetection Component, Save the Changes and navigate to the browser.

We can see that for every 5 seconds we get that alert notification. That means our component is getting re-rendered though there is no change in the data. That is because our Component is not Pure.

In order to avoid re-rendering of our ChangeDetectionComponent when there are no changes to the state data,  we will make ChangeDetection Component as Pure.

PureComponent is similar to Component. The difference between them is that Component doesn’t implement **shouldComponentUpdate**(), but PureComponent implements it with a shallow comparison of properties and state data.

PureComponent’s shouldComponentUpdate() method returns true or false  If the previous state and props data is the same as the next props or state, resulting the component is not Re-rendered.

If our React component’s render() function renders the same result given the same props and state, we can use React.PureComponent for a performance boost.

Lets go to ChangeDetection Component Class, instead of extending it from react.Component Class, extend it from React.PureComponent Class. Save the Changes, navigate to the browser. We can observe that we will not get the alert Notification repeatedly.

That is because there is no change in the Employee Count so that Component has no need to re-render.

Lets go to our Web API and add a record to our list. Build this Project. Lets go to our React Application. Now we will get the Alert notification because the count value is changed.

Now we can click on the Reload button in the Reports Component to get the New Employee Record.

React Components re-renders in the following scenarios:

1. “setState” is called in Component

2. “props” values are updated

3. this.forceUpdate() is called

In the case of Pure Components, the React components do not re-render blindly without considering the updated values of React “props” and “state”. If updated values are the same as previous values, render is not triggered.

React.PureComponent’s shouldComponentUpdate() performs only **shallow comparison** of the data. If data is either of type Objects or  complex data structures, it may result in wrong rendering behaviors.

Now the question is what is Shallow Comparison?

Shallow compare checks for equality. When comparing values of type number or string it compares their values.

When comparing objects, it checks whether both the objects are pointing to the Same location or not. The Property Values of the Objects are not Compared.

That means if we have two objects named emp and emp1, and if we shallow Compare them, then it returns false.

let emp={

Id:101,

Name:’SpringPeople’};

let emp1={

Id:101,

Name:’SpringPeople’}

But if we assign emp to emp1 and do the Comparison, then it returns true.

let emp={

Id:101,

Name:’SpringPeople’};

let emp1=emp;

Lets go to our ChangeDetection Component Class, to the state object, lets add one employee object.

In setInterval method, though we assign the same object to the employee property of the state object, still we get the Alert Notification. Its because of the Shallow Comparison. Because it looks for the object which is holding the address but not the same values.

Now lets assign the same state object employee property. Save these changes, navigate to the browser. We can see that we don’t get the Alert Notification recurrently.

Pure Components are introduced for **performance enhancement**. You can use this optimization to improve the performance of your components. But remember that it works well provided our state and props are Primitive types only but not reference types like object or Array.

# Component Lifecycle Methods

Each component in React has several “lifecycle methods” using which we can run different pieces of code at particular times in the component processing.

If we remember from our previous articles, in every session we are creating a component, calling the component and rendering that component.

We will create one Component Called as Employee Component.

class Employee extends React.Component{

constructor(props){

super(props);

}

render(){

return(

<div>

<h2>Welcome to Employee Component...</h2>

</div>

)

}

}

Lets Call this Component and render that to our root container.

Save these changes, navigate to the browser. We call see that our Employee Component is rendered.

Our Employee Component is inserted into the DOM Tree. This Phase is Called as Mounting.

Commonly used lifecycle methods when an instance of a component is being created and inserted into the DOM are:

#### ****constructor()****

The constructor of a React component is called before it is mounted. When implementing the constructor for a React.Component subclass, we  should call super(props) before any other statement. Otherwise, this.props will be undefined in the constructor, which can lead to bugs.

Typically, in React constructors are only used for two purposes:

• Initializing local state by assigning an object to this.state.

• Binding event handler methods.

Remember that we should not call setState() in the constructor. Constructor is the only place where we  should assign this.state directly. In all other methods, we  need to use this.setState() instead.

#### ****render()****

The render() method is the only required method in a class component. Output of a Component is dependent on what we return from this method.

Render method returns one of the following types:

• React elements

• Arrays and fragments.

•  Portals

• String and numbers.

• Booleans or null.

We have discussed about how to return React Elements, Fragments and Portals from render method in our previous Videos. The render() function should be pure, meaning that it should not modify component state

#### ****componentDidMount()****

componentDidMount() is invoked immediately after a component is mounted (inserted into the tree). If we  need to load data from a remote endpoint, this is a good place to instantiate the network request. We have discussed about usage of each one of this in our previous articles.

Now lets go to our Employee Component. Lets create a state object in the Constructor and we will add Name property to this object and we will initialize it to empty. Lets go to render method,  we will place one input element using which we can Enter Employee Name. Lets call onChange event on this input and we will update the state object when the onChange event is triggered.

Whenever there is a change in the **state data or props data or if forceUpdate method** is called, then the component gets re-rendered.

The Commonly used methods when a component is being re-rendered are:

• render()

• componentDidUpdate()

componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render. We can Use this method to operate on the DOM when the component has been updated. We will see an example on using this method in our upcoming articles.

Then comes the next phase **unmounting**.

When we develop one application, we develop multiple components as part of it. User will be navigating from one component to another component. Just like going from Home to AboutUs and then navigating to Contact Us tabs.

When we go from one Component to the Other Component, the Previous component will be removed from the DOM and the new Component contents will be displayed in the UI.

That is called **Unmounting**.

When that happens, we want to Perform any necessary cleanup required, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in componentDidMount().

**componentWillUnmount()** is invoked immediately before a component is unmounted and destroyed.

Lets understand this with one example. We will use the same component classes we have developed in our last article. They are ChangeDetection Component Class and Reports Component Class.

Now lets call ChangeDetection Component class and render that to our root container.

Now we will place a button the changedetection component class render method. Lets give the text as Load Reports Lets add onClick attribute and lets call a function.

We will implement the LoadReports function and on click of this function, we will render our Reports Component.

Lets save these changes, navigate to the browser.

We can see the contents of ChangeDetection Component are being displayed. Click on the button, we can see the Contents of Reports Component.

When we have moved from ChangeDetection Component to Reports Component, the ChangeDetection Component gets unmounted and Reports Component will be mounted.

In the ChangeDetection Component Class, getEmployeesCount function, lets display an alert message. Save these changes, navigate to the browser.

Open Developer Tools, we can see the log entry in the console for every 5 seconds. Now lets click on Load Reports Button, we moved away from ChangeDetection Component but our setInterval function code will be getting executed continuously. Resulting we get the alertd even after we moved away from that Component. But that should not happen. We have clear the Interval which we have set.

This is where we will make use of componentWillUnmount()

Method. With in this method, we will call clearInterval Javascript function using which we will clear the setInterval .

Lets save these changes, navigate to the browser. Repeat the same Process again but this time we will not see logs after our ChangeDetection Component is Unmounted.

We will use componentWillUnmount method to Perform clean up tasks.

We have discussed about Commonly used Lifecycle methods of Components.

There are some rarely used lifecycle methods. They are

* getDerivedStateFromProps
* shouldComponentUpdate
* getSnapshotBeforeUpdate

We have discussed about shouldComponentUpdate method in our last article and we will discuss few of the remaining in our upcoming articles.

import ReactDOM from "react-dom";

import React, { Component } from "react";

class ChangeDetection extends React.Component {

constructor(props) {

super(props);

this.state = {

employeeCount:0

};

setInterval(this.getEmployeesCount, 5000);

}

componentWillUnmount(){

clearInterval()

}

getEmployeesCount=()=>{

alert('Fetching the Employee Count from the REST API');

fetch("https://localhost:44306/api/Employee")

.then(res => res.json())

.then(

(result) => {

this.setState({

employeeCount: result.length

});

}

);

}

componentDidMount(){

this.getEmployeesCount();

}

showReports=()=>{

ReactDOM.render(<Reports></Reports>,document.getElementById("root"));

}

render() {

return (

<div>

<h2>Welcome to Component Lifecycle Methods Demonstration...</h2>

<p>

<label>Number of Employees are : <b>{this.state.employeeCount}</b></label>

</p>

<button onClick={this.showReports}>Show Reports</button>

</div>

);

}

}

class Reports extends React.Component {

constructor(props) {

super(props);

this.state = {

employees:[]

};

}

componentDidMount=() => {

this.getEmployees();

}

getEmployees() {

fetch("https://localhost:44306/api/Employee")

.then(res => res.json())

.then(

(result) => {

this.setState({

employees: result

});

}

);

}

loadEmployees=()=>{

this.getEmployees();

}

render() {

return (

<div>

<h2>Employees Data...</h2>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

<th>Location</th>

<th>Salary</th>

</tr>

</thead>

<tbody>

{this.state.employees.map(emp => (

<tr key={emp.Id}>

<td>{emp.Id}</td>

<td>{emp.Name}</td>

<td>{emp.Location}</td>

<td>{emp.Salary}</td>

</tr>

))}

</tbody>

</table>

<p>

<button onClick={this.loadEmployees}>Reload</button>

</p>

</div>

);

}

}

const element=<ChangeDetection></ChangeDetection>

ReactDOM.render(element,document.getElementById("root"));

# Introduction to Hooks

We know that a Component in React can be created as a **Class Component** or a **function Component**. We have discussed that when we want features like Managing State in React Components or responding to Lifecycle methods then we will opt for using Class Components.

Developers have encountered a wide variety of seemingly unconnected problems in React over five years of writing and maintaining tens of thousands of components.

And Majority of them are coming because of using Class Components.

Now the question is What are the Problems of existing Class Components

1. Wrapper Hell

2. Huge Components

3. Confusing Classes

4. classes don’t minify very well

We will discuss few of them in detail in this article and others in our upcoming articles.

React want to present an API out of the box that makes React easier to build great UIs with the best Performance.

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

**Hooks are functions** that let you “hook into” React state and lifecycle features from function components. Hooks don’t work inside classes — they let you use React without classes.

To make function components more powerful, React has introduced several built in hooks.

Hooks in React are Classified into Basic Hooks, Additional Hooks.

**Basic Hooks**

• useState

• useEffect

• useContext

**Additional Hooks**

• useReducer

• useCallback

• useMemo

• useRef

• useImperativeHandle

• useLayoutEffect

• useDebugValue

We can also create Custom Hooks.

We will discuss about useState Hook in this article and we will discuss about others in our upcoming articles. In this Process, we will also understand how Hooks make our React Code better.

Lets say we want to develop one Employee Component and we will create it as a Class Component.

We have created the below Component in our last article.

class Employee extends React.Component{

constructor(props){

super(props);

this.state={

Name:''

}

}

changeName=(e)=>{

this.setState({Name:e.target.value});

}

render(){

return(

<div>

<h2>Welcome to Employee Component...</h2>

<p>

<label>Employee Name :

<input type="text" value={this.state.Name} onChange={this.changeName}></input>

</label>

</p>

<p>

Entered Name is : <b>{this.state.Name}</b>

</p>

</div>

)

}

}

Lets Call this Employee Component and we will render that to our root container.

Save the changes, navigate to the browser. We can see the Output.

If we look at the code we have written, for a simple use case, we are writing more lines of code. That includes writing constructor, calling baseclass constructor. All that involves additional overhead to the application performance. As the application complexity grows, even our code becomes more and it becomes unmanageable.

But as a developer, we don’t want our code to become unmanageable.

Now we will develop a NewEmployee Component but this time we will create it as a function component.

If we look at our Employee Class Component, we have added one property to our state object called as Name and this is initialized to empty. Then we have created one function called as changeName using which we are updating the state object.

In the Same way, even we want to have one Name property and one function to update the Name in our function Component as well.

We wanted our Name to be Stateful. We want to store this Name in the State object and we want to update the Name.

This is where we will make use of useState hook in react.

Remember that hook is a function. So now we are going to call a function called as useState.

This useState function can take one argument that is initialiState value. Unlike with classes, the state doesn’t have to be an object. We can keep a number or a string if that’s all we need. In our case, we wanted the state for Name, so we empty.

useState function returns a pair of values: the current state and a function that updates it.

So we can write

**const [name,setName]=useState();**

Here Name is the Property Name and setName is the function using which we update the value of Name into our state Object.

Lets write one function which will be called when there is a change in the Name.

With in this function, we will update the Name value to our state object usin setName function.

function changeName(e){

setName(e.target.value);

}

Lets return the div container, add one h2 Tag with text as Welcome to New Employee Component…

Lets Place one input element and value of that input element should be the name from our state object and we will call changeName function when there is a change.

Next lets display the Entered Name.

return(

<div>

<h2>Welcome to New Employee Component...</h2>

<p>

<label>Employee Name :

<input type="text" value={name}

onChange={changeName}></input>

</label>

</p>

<p>

Entered Name is : <b>{name}</b>

</p>

</div>

)

Now instead of Calling the Employee Component, we will call our NewEmployee Component. Save the changes, navigate to the browser.

We can see that as we keep entering the Name, the entered Name is displayed down to our input element.

We are able to achieve same functionality through a functional component.

If we observe our NewEmployee Component Code, we don’t have a constructor, we are not calling the base class constructor. We have not implemented any render method.

We are Calling a function and it is returning us the output.

Now lets say we want to initialize our name value to SpringPeople. So we can go to our useState function and pass the value as the input to the function.

**const [name,setName]=useState('SpringPeople);**

We can save the changes. Navigate to the browser.

We can see that our textbox is holding the value by default and we can change if needed.

we have introduced ourselves to the concept of Hooks in React. We have a lot to learn and We will continue discussing more about Hooks in our Upcoming articles.

we have just seen an intro to Hooks in React. We will discuss more about Hooks in our Upcoming articles.

# useState Hook

In this article, we will discuss about **useState**deeper in React.

We have created NewEmployee Component in our last article.

Our Employee Component is having one input element. Now lets say we want to have another input using which we can enter Employee Location.

We can repeat the same steps for Location as well.

Lets the paragraph tag and paste.

We can declare Multiple State Variables.

const [location,setLocation]=useState();

Lets display the entered location as well in the Paragraph tag.

<p>

<label>Employee Location :

<input type="text" name="Location" value={location}

onChange={changeEmployeeLocation}></input>

</label>

</p>

Save the changes, navigate to the browser. We can see the output.

Now if we want to build an employee creation form where we will also have inputs for Employee Id and employee salary as well, the current way of creating the multiple state variables may not be the right approach.

State variables can hold objects and arrays just fine, so you can still group related data together.

Lets create an object and pass it to the useState function. The object will hold employee Id, name, location and Salary.

Lets create a function named as changeEmployee and with in this function we will call setEmployee function and we will update the state object.

We will use spread operator to pass the current employee object data and we will update the respective elements value.

Now we will update our input elements to use this objects data. Don’t forget to assign the name to each input element name. Save these changes. Navigate to the browser. We can see the output as we type in the data.

Lets save the changes, navigate to the browser. We can see the output.

import ReactDOM from "react-dom";

import React, { Component, useState } from "react";

function NewEmployee(){

const [employee,setEmployeeData]=useState({Id:'',Name:'',Location:'',Salary:''});

function changeEmployeeInfo(e){

setEmployeeData({...employee,[e.target.name]:e.target.value});

}

return(

<div>

<h2>Welcome to Employee Component...</h2>

<p>

<label>Employee ID :

<input type="text" name="Id" value={employee.Id}

onChange={changeEmployeeInfo}></input>

</label>

</p>

<p>

<label>Employee Name :

<input type="text" name="Name" value={employee.Name}

onChange={changeEmployeeInfo}></input>

</label>

</p>

<p>

<label>Employee Location :

<input type="text" name="Location" value={employee.Location}

onChange={changeEmployeeInfo}></input>

</label>

</p>

<p>

<label>Employee Salary :

<input type="text" name="Salary" value={employee.Salary}

onChange={changeEmployeeInfo}></input>

</label>

</p>

<p>

Employee ID is : <b>{employee.Id}</b>, Name is : <b>{employee.Name}</b> ,

Location is : <b>{employee.Location}</b> and Salary is : <b>{employee.Salary}</b>

</p>

<SalaryComponent salary={employee.Salary} onSalaryChange={changeEmployeeInfo}></SalaryComponent>

</div>

)

}

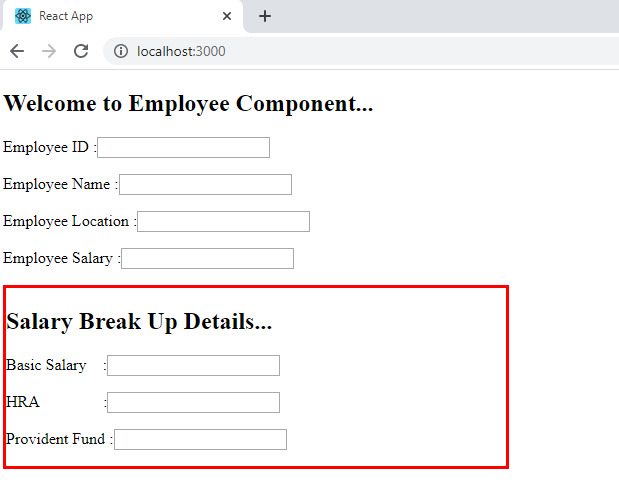
const element=<NewEmployee></NewEmployee>

ReactDOM.render(element,document.getElementById("root"));

# useState Part-2

In this article, we will discuss about passing data from Parent Compnonent to child Component and from Child Component back to Parent Component when the Components are function Components.

Lets assume that our application is having two components. One is Employee Component and the other one is Salary Component. It looks like below image.



Employee Component is the Parent component and the Salary Component will be used as child component.

When user enters the Salary in the Employee Component, There is a formula based on which the Salary Component details like Basic Salary, HRA and PF gets populated.

We can modify the Salary Break Up details in Salary Component and accordingly Salary should be Calculated and displayed in the Employee Component.

That means we want to pass the data from Employee component to the Salary component and from Salary Component back to Employee Component.

We have seen this in the case of class components and Lets see how we can achieve the same using function components.

We have already created Employee Component in our last article and now assuming that we will create Salary Component, lets call the Salary Component from our Employee Component. To this Salary Component we will pass the Salary as one input through property and we can also pass changeEmployeeInfo function to the Salary Component through another Property.

Now lets go ahead and create our Salary Component. This Salary Component function receives two inputs. One is the Salary value and the other one is the callback function.

I the salary input element code from the Employee Component and placing it here.

Save the changes. Navigate to the browser. We can see that If we enter Salary in Employee Component, it gets updated in Salary Component and if we change it Salary Component, it gets updated into Employee Component.

import ReactDOM from "react-dom";

import React, { Component, useState } from "react";

function NewEmployee(){

const [employee,setEmployeeData]=useState({Id:'',Name:'',Location:'',Salary:''});

function changeEmployeeInfo(e){

console.log(e);

setEmployeeData({...employee,[e.target.name]:e.target.value});

}

return(

<div>

<h2>Welcome to Employee Component...</h2>

<p>

<label>Employee ID :

<input type="text" name="Id" value={employee.Id}

onChange={changeEmployeeInfo}></input>

</label>

</p>

<p>

<label>Employee Name :

<input type="text" name="Name" value={employee.Name}

onChange={changeEmployeeInfo}></input>

</label>

</p>

<p>

<label>Employee Location :

<input type="text" name="Location" value={employee.Location}

onChange={changeEmployeeInfo}></input>

</label>

</p>

<p>

<label>Employee Salary :

<input type="text" name="Salary" value={employee.Salary}

onChange={changeEmployeeInfo}></input>

</label>

</p>

<p>

Employee ID is : <b>{employee.Id}</b>, Name is : <b>{employee.Name}</b> ,

Location is : <b>{employee.Location}</b> and Salary is : <b>{employee.Salary}</b>

</p>

<SalaryComponent onSalaryChange={changeEmployeeInfo} salary={employee.Salary}></SalaryComponent>

</div>

)

}

const SalaryComponent=({onSalaryChange,salary})=>{

function changeSalary(e){

onSalaryChange(e);

}

return(

<div style={{border:'3px solid red', width:'500px'}}>

<h2>Welcome to Salary Component</h2>

<p>

<label>Employee Salary :

<input type="text" name="Salary" value={salary}

onChange={changeSalary}></input>

</label>

</p>

</div>

);

}

const element=<NewEmployee></NewEmployee>

ReactDOM.render(element,document.getElementById("root"));

We have seen how to pass state data between function components.

We have discussed about useState hook in React in detail.

There are few important Points one has to remember about Hooks.

# useEffect

Many times, we want to run some additional code after React has updated the DOM.

That code can be getting the Data by Calling a Web API or Setting up subscriptions or writing the logs after the DOM is ready.

If we want to write such additional code in Class Components, we have lifecycle methods like componentDidMount, componentDidUpdate methods.

What if, if want to write such code in the case of function components.

Lets Open index.js file from our demo-project.

Lets create EmployeeReports Component using which we display the list of Employees by fetching it from API.

We know how to send a Web API request and get the data from our React Application. But now question is where should we write the code.

Remember that we want to execute that code after DOM is ready.

If we think of writing just before our return statement, just lets do one thing. We will write a console log here which shows us the DOM status.

console.log(document.readyState);

Call our Employee Reports Component and render that to our root container.

At this point of time, our table will be empty without data because we are yet to send the Web API request to get the data.

Save the changes, navigate to the browser.

We can see the table but no data. Lets open developer tools, we can see the status. It says that loading.

That means we should not be writing the Code here.

So where should we write the code which should get executed after our DOM is ready.

This is where we will make use of another hook in React called as **useEffect**.

**useEffect is a**function that runs when the component is first rendered, and on every subsequent re-render/update.

We can think of **useEffect**Hook as **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount** combined.

Lets call useEffect function and it takes one callback function as a parameter.

We will write one arrow function. Inside that function, lets place the same console.log statement we have written earlier. Save the changes, navigate to the browser.

Open developer tools and we can see that the status is Complete.

Our DOM is ready and we can do all the other operations we want now.

Lets go back to Visual studio code, now within this useEffect, lets place the code using which we will send the API request. We have discussed this already in our previous sessions. I am pasting that code here. When we get the response from our Web API, we will call our setEmployees function and we will pass the list to that.

Save the changes, navigate to the browser. We can see that employee details are being displayed here.

When the DOM is ready, we are sending the API request, getting the employees data and updating our employees state variable by calling setEmployees method. But this has a Problem. Remember that when there is any change in the properties data or state data of a component, then that component gets re-rendered.

Resulting our useEffect function gets called again. It will send an API request, get the data and assign it to employees state variable. That will make the component gets re-rendered and it will go into infinite loop.

Lets add an alert. Save the changes, navigate to the browser. We can see that we get the alert again and again.

If you want to run an effect only once (on mount and unmount), you can pass an empty array ([]) as a second argument. This tells React that your effect doesn’t depend on any values from props or state, so it never needs to re-run.

import ReactDOM from "react-dom";

import React, { Component, useState, useEffect } from "react";

function EmployeeComponent(){

const [employees,setEmployees]=useState([]);

useEffect(()=>{

alert('We are in useEffect function');

fetch("https://localhost:44306/api/Employee")

.then(res => res.json())

.then(

(result) => {

setEmployees(result);

}

);

},[]);

return(

<div>

<h2>Employees Data...</h2>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

<th>Location</th>

<th>Salary</th>

</tr>

</thead>

<tbody>

{employees.map(emp=>(

<tr key={emp.Id}>

<td>{emp.Id}</td>

<td>{emp.Name}</td>

<td>{emp.Location}</td>

<td>{emp.Salary}</td>

</tr>

))}

</tbody>

</table>

</div>

)

}

const element=<EmployeeComponent></EmployeeComponent>

ReactDOM.render(element,document.getElementById("root"));

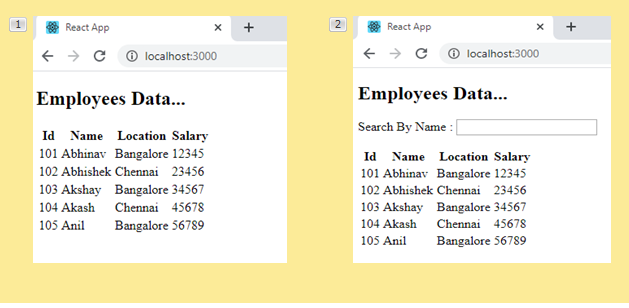
# useEffect Part-2

In our last article, we have developed one Employee Component, which sends a Web API request, gets the list of employees and display them in a table.

Now we will extend that further. Assume that we have to implement one Auto Complete textbox in an application or we have to provide search capabilities to our Employee Data.

There will be a textbox using which we will enter employee name. As we keep entering the employee name, we have to send a request to our Web API, get the list of employees based on the search criteria and display the data in our Employee Component.

It is going to look like this.



This is very common requirement in most of the web applications.

First we will add the required functionality to our Web API.

public List<Employee> GetByName(string name)

{

if (name == null || name == "")

{

return empList;

}

else

{

return empList.Where(e => e.Name.Contains(name)).ToList();

}

}

Now lets open index.js file from our demo-project. We have developed the Employee Component in our last video. We will add the required changes to the same Component.

Lets add one state variable which will be holding search text.

Lets place the input element using which we will enter the search text. Add onChange attribute to the input element and call a function called as onSearchTextChange .

Lets implement the onSearchTextChange function.

When we are calling the Web API, we have to pass the name as well. Lets pass that search-text through the url.

We wanted our code inside useEffect hook to be executed when the search text changes.

But empty [] tells react that we want that effect to be run only once.

This tells React that our effect doesn’t depend on any values from props or state, so it never needs to re-run.

But now we wanted useEffect hook to be executed when the state data changes.

We can pass the state variable as a dependency to our useEffect hook.

Save the changes, navigate to the browser. We can see that the table data changes based on our search text.

Our **useEffect**hook can have dependency on **multiple state variables or properties as well.**

For example, we can create another state variable called as employeeCount.

Lets place a button to add a new Employee. Assuming that on click of this button, we are adding a new employee. When the employee is created successfully, we get the number of employees and update the employeecount state variable. Now when there is a change in the count, we have to re-render our employee data.

That means the employees data should be re-rendered when the search text changes or when the employeecount changes. Now we will add employeesCount data also as a Dependency to our effect.

import ReactDOM from "react-dom";

import React, { Component, useState, useEffect } from "react";

function EmployeeComponent(){

const [employees,setEmployees]=useState([]);

const [searchText, setSearchText]=useState('');

const [employeeCount, setEmployeeCount]=useState(0);

useEffect(()=>{

alert('We are inside useEffect Method');

fetch("https://localhost:44306/api/Employee/"+searchText)

.then(res => res.json())

.then(

(result) => {

setEmployees(result);

}

);

},[searchText,employeeCount]);

function onSearchTextChange(e){

setSearchText(e.target.value);

}

function addNewEmployee(){

setEmployeeCount(employeeCount+1);

}

return(

<div>

<h2>Employees Data...</h2>

<p>

<label>Search By Name : <input type="text" value={searchText}

onChange={onSearchTextChange}></input></label>

</p>

<p>

<button onClick={addNewEmployee}>Add Employee</button>

</p>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

<th>Location</th>

<th>Salary</th>

</tr>

</thead>

<tbody>

{employees.map(emp=>(

<tr key={emp.Id}>

<td>{emp.Id}</td>

<td>{emp.Name}</td>

<td>{emp.Location}</td>

<td>{emp.Salary}</td>

</tr>

))}

</tbody>

</table>

</div>

)

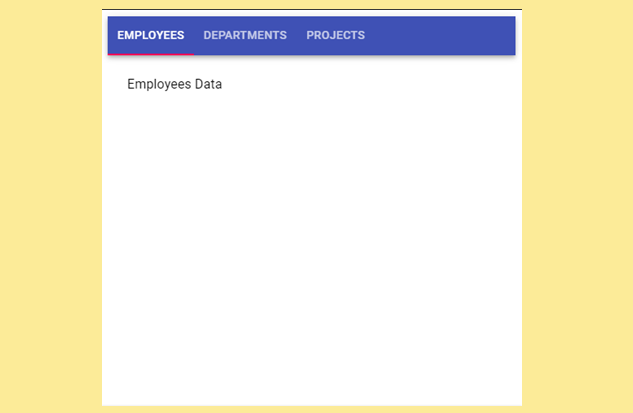
}

const element=<EmployeeComponent></EmployeeComponent>

ReactDOM.render(element,document.getElementById("root"));

# useEffect Part-3

When we develop one application, we develop multiple components as part of it. User will be navigating from one component to another component. Just like going from Employees Tab to Departments and then navigating to Projects tabs.



When we go from one Component to the Other Component, the Previous component will be removed from the DOM and the new Component contents will be displayed in the UI.

This is called Unmounting.

When that happens, we have to Perform any necessary cleanup required, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in that respective component.

We have discussed about similar example in the Case of Class Components. Class Components have one life cycle method called as componentWillUnmount and we write such clean up code inside that method.

Lets understand how do we do that in the Case of function components.

Lets create our Employee Component,

Lets create one state variable named as employeeCount and pass a function to update the same. We will initialize it to 0.

Lets add effect hook, and with in that we will write the code to get the list of employees from our Web API. I have the code handy and pasting it here. With in the getEmployees function, when we get the response we will update our employeeCount value.

We will call that function from our effect.

Lets return one div container. With in this div we will display the employee count.

Now we wanted our getEmployees function to be called for every 5 seconds and show us the updated count. As we have done in our previous video, we will call that getEmployees function using setInterval function and we will give 5 seconds as the interval.

Lets add one alert within our getEmployeesCount function so that we will be notified when that method is called.

Lets create another component called as Departments component and we will return a simple message from this component.

Now we will place a button in the Employee Component and on click of that button, we will render Department Component.

Lets call our Employee Component and render that to our root container.

Save the changes, navigate to the browser. We can see the output.

We can observe that we will get the alert for every five seconds.

Lets Click on the Departments Button. We will see the contents of Departments Component. But one can observe that we will get the Employee Components alert notifications even when we are inside Department Component. Ideally that should not happen.

We have clear the interval which is being set so that we don’t introduce a memory leak.

When we want to do any clean up tasks, we dont need a separate effect to perform the cleanup.

We can do that in the same effect. Lets see how do we do that.

Any Clean up code we wanted to be executed when the Component is Unmounted, we will write it in a function and we will return that function from our effect.

React calls that function when the component unmounts. The function can be a named function or an arrow function as well.

As we have discussed in our Previous articles, we will use clearInterval method of javasript to clear that Interval. But to that clearInterval method, we have to pass the handle. We know that when we call setInterval method, it returns a handle. We will pass that handle to our clearInterval. Save the changes, navigate to the browser.

We can observe that we will get the alert for every five seconds.

Lets Click on the Departments Button. We will see the contents of Departments Component. But one can observe that we dont get the Employee Components alert notifications when we are inside Department Component that is because the cleanUp is done.

As we have seen the useEffect Hook is the combination of componentDidMount, componentDidUpdate, and componentWillUnmount lifecycle methods.

Currently we are display the number of Employees in our Employee Component. Lets say we have to display another message which shows when did the last employee added. Like Last Employee is added one day ago, two days ago something like that.

Lets create another state variable noOfDaysCount. We have to get this value also from our WebAPI. Assuming that we have TestApi which gives the noOfDays value.

With in our effect, lets write the code to call the Web API and assign it to our noOfDays state variable.

If we observe the our effect hook code, we are dumping everything in one place. Instead of that, like how we use the State Hook more than once, we can also use several effects. This lets us separate unrelated logic into different effects.

React will apply every effect used by the component, in the order they were specified.

I am sure by this time, we are very clear on what is effect hook, how to use that in our React Projects.

import ReactDOM from "react-dom";

import React, { Component, useState, useEffect } from "react";

function Employee(){

const [employeeCount, setEmployeeCount]=useState(0);

const [noOfDays,setNoOfDays]=useState(0);

useEffect(()=>{

var handle=setInterval(getEmployeesCount,5000);

return ()=>{

clearInterval(handle);

}

});

useEffect(()=>{

fetch("https://localhost:44306/api/Test")

.then(res => res.json())

.then(

(result) => {

setNoOfDays(result);

}

);

})

function getEmployeesCount(){

alert('Getting the Employees Count');

fetch("https://localhost:44306/api/Employee")

.then(res => res.json())

.then(

(result) => {

setEmployeeCount(result.length);

}

);

}

function navigateToDepartment(){

ReactDOM.render(<Departments></Departments>,document.getElementById("root"));

}

return (

<div>

<h2>Welcome to Employee Component...</h2>

<p>

<label>Employee Count : <b>{employeeCount}</b></label>

</p>

<p>

<label>Last Employee Added : <b>{noOfDays} days ago...</b></label>

</p>

<button onClick={navigateToDepartment}>Departments</button>

</div>

)

}

function Departments(){

return(

<div>

<h2>Welcome to Departments Component...</h2>

</div>

)

}

const element=<Employee></Employee>

ReactDOM.render(element,document.getElementById("root"));

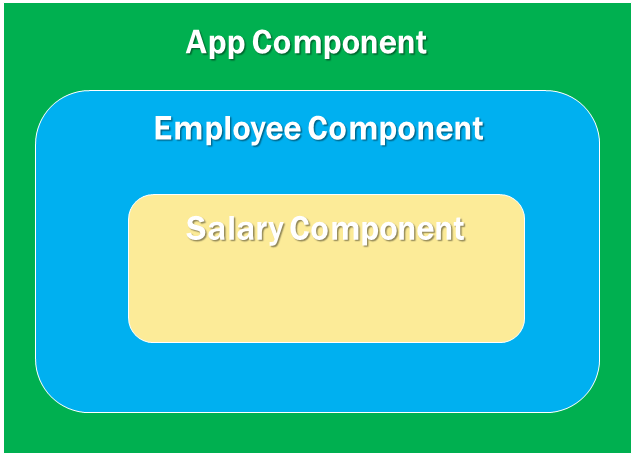
# useContext Hook

In a typical React application, data is passed top-down (parent to child) via props, but this can be difficult for certain types of props (e.g. locale preference, UI theme) that are required by many components which are Nested at different levels within an application.

In this article, we will understand how we use Context to pass the data between components at different nesting levels.

Lets take a look at one example. When an Employee is Logged into the React application, we have a Nesting of Components which are making our UI.

They are App Component, Employee Component and Salary Component. App Component has an Employee Object and this data is needed by Employee Component and Salary Component in Order to function.



Context provides a way to pass data through the component tree without having to pass props down manually at every level.

Context provides a way to share values between components without having to explicitly pass a prop through every level of the tree.

Context is primarily used when some data needs to be accessible by many components at different nesting levels.

We have discussed this in the case of Class Components. We will understand how do we use the context in the case of Function Components.

We will create three functional components. Named App Component, Employee Component and Salary Component.

Lets Call the Salary Component from Employee Component and Call the Employee Component from App Component.

Call the App Component and render it to our DOM.

In App Component, lets create one employee state variable and a function to update the employee data using useState hook and we will initialize the state.

This employee object is needed by employee Component and by Salary Component.

Lets see how do we do that using Context in React.

Lets create context object using React.createContext Method.

const EmployeeContext = React.createContext();

using the EmployeeContext object, we will pass the data from AppComponent to the Employee Component and then from the Employee Component to the Salary Component.

Lets go to App Component, we modify the way how Employee Component is being Called from App Component.

So that Employee Component can receive the data from App Component and pass that to the Child Components of Employee Component implicitly.

Every Context object comes with a Provider React component that allows consuming components to subscribe to context changes.

Context Provider Accepts a value property to be passed to consuming components that are descendants of this Provider.

<EmpContext.Provider value={empData}>

<Employee/>

</EmpContext.Provider>

Now this empData can be accessed in both Employee Component and Salary Component using useContext hook in React.

Lets go to Employee Component, get the Employee Context using useContext hook.

We can display the Employee details by reading from the context.

We can do the Same in Salary Component as well.

Save the Changes, navigate to the browser. We can see the Output.

We can see that our employee data from App Component is accessed by the

Components which are placed at different Nesting Levels.

One Level is from App Component to Employee Component and the Second one is from Employee to Salary Component.

import ReactDOM from "react-dom";

import React, { Component, useState, useContext } from "react";

const employeeContext=React.createContext();

function App(){

const [employee,setEmployee]=useState({Id:101,Name:'SpringPeople',

Location:'Bangalore',Salary:12345});

return(

<div>

<h2>Welcome to App Component...</h2>

<employeeContext.Provider value={employee}>

<Employee></Employee>

</employeeContext.Provider>

</div>

);

}

function Employee(){

let context=useContext(employeeContext);

return(

<div>

<h2>Welcome to Employee Component...</h2>

<p>

<label>Employee ID : <b>{context.Id}</b></label>

</p>

<p>

<label>Employee Name : <b>{context.Name}</b></label>

</p>

<Salary></Salary>

</div>

);

}

function Salary(){

let context=useContext(employeeContext);

return(

<div>

<h2>Welcome to Salary Component...</h2>

<p>

<label>Employee ID : <b>{context.Id}</b></label>

</p>

<p>

<label>Employee Salary : <b>{context.Salary}</b></label>

</p>

</div>

);

}

const element=<App></App>

ReactDOM.render(element,document.getElementById("root"));

# useContext Part-2

In this article, we will continue our discussion about useContext hook in React.

In the last article, we have discussed about how do we use Context in React to Pass the Employee data from App Component to its descendants.

Resulting we were able to access the Employee Data in both Employee Component and Salary Component.

Now what if if we want to update the Employee Salary in the Salary Component or what if if we want to update the Employee Data from any of the Child Components.

Lets see how do we do that.

Lets display the Employee salary value in App Component as well as in Employee Component so that we can visualize the Salary Change Value in all the Components.

Now in App Component we have one function called as setEmployee using which we can update the employee data.

We can pass even this function to the Child Components along with the employee data using the value attribute.

Lets create one object which will hold both employee data and the setEmployees function.

<employeeContext.Provider value={{data:employee, updateFunction:setEmployee}}>

<Employee></Employee>

</employeeContext.Provider>

Now in Employee Component, we get the Employee details using context.data property as that is the Property through which we have passed the data. Lets make the changes accordingly.

We will do the same changes even in Salary Component.

Now lets place a button in the Salary Component using which we will update the Employee Salary. Lets call a function on Click of this button.

Now we will implement the updateSalary function. With in this updateSalary function, we will call our update function and to that function we will pass the employee object and the updated Salary.

Save the changes. Navigate to the browser. We can see that Salary is being displayed in all the three components.

Now click on Update button, we can see that the Salary gets updated in all the three Components.

Lets add two more components to our code. One Component contents will be displayed if the Employee is Permanent and we will show the other component contents if the Employee is Contract.

I have the code handy for these components and I am pasting it here.

Now in the Employee Component, we have to display one of this Component based on the Employment Type.

Lets add one new Property to our employee object in the App Component called as Type and we will initialize it to Contract.

If that is the case our Employee Component should render Contract Component and whenever we change the Type to Permanent Employee Component should render Permanent Component Contents.

To accomplish this we use **context.consumer**.

It is A React component that subscribes to context changes. This Component Requires a function as a child. The function receives the current context value and returns a React node.

If the employee Type is Permanent we Call Permanent Component else we call the other Component.

Now lets place a button in Employee Component using which we can change the Employment type. We will handle onClick event.

With in the function, lets change the Type.

Save the Changes, navigate to the browser.

We can see that by default we get the Contents of Contract Component. Change the Employment type by clicking on the button. We can see that we get the Contents of Permanent Component.

With this I hope we are very clear on how to use useContext hook in React.

import ReactDOM from "react-dom";

import React, { Component, useState, useContext } from "react";

const employeeContext=React.createContext();

function App(){

const [employee,setEmployee]=useState({Id:101,Name:'SpringPeople',Type:'Contract',

Location:'Bangalore',Salary:12345, EmploymentType:'Contract'});

return(

<div>

<h2>Welcome to App Component...</h2>

<p>

<label>Employee Salary : <b>{employee.Salary}</b></label>

</p>

<employeeContext.Provider value={{data:employee,updateEmployee:setEmployee}}>

<Employee></Employee>

</employeeContext.Provider>

</div>

);

}

function Employee(){

let context=useContext(employeeContext);

function changeEmploymentType(){

context.updateEmployee({...context.data,EmploymentType:'Permanent'});

}

return(

<div>

<h2>Welcome to Employee Component...</h2>

<p>

<label>Employee ID : <b>{context.data.Id}</b></label>

</p>

<p>

<label>Employee Name : <b>{context.data.Name}</b></label>

</p>

<p>

<label>Employee Salary : <b>{context.data.Salary}</b></label>

</p>

<employeeContext.Consumer>

{value=>value.data.EmploymentType==='Permanent'?

<Permanent></Permanent>:<Contract></Contract>}

</employeeContext.Consumer>

<button onClick={changeEmploymentType}>Make Permanent</button>

<Salary></Salary>

</div>

);

}

function Salary(){

let context=useContext(employeeContext);

function updateSalary(){

context.updateEmployee({...context.data,Salary:15000});

}

return(

<div>

<h2>Welcome to Salary Component...</h2>

<p>

<label>Employee ID : <b>{context.data.Id}</b></label>

</p>

<p>

<label>Employee Salary : <b>{context.data.Salary}</b></label>

</p>

<button onClick={updateSalary}>Update</button>

</div>

);

}

function Permanent(){

return (

<div>

<h2>Permanent Component Contents...</h2>

</div>

);

}

function Contract(){

return(

<div>

<h2>Contract Component Contents...</h2>

</div>

)

}

const element=<App></App>

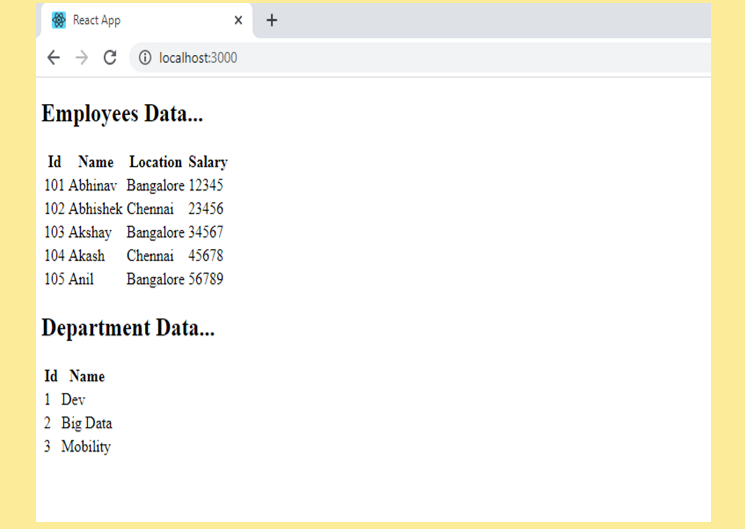
ReactDOM.render(element,document.getElementById("root"));

# Custom Hooks

In this article, we will discuss about Building **Custom hooks** in React.

There might be instances where we have been using the same repetitive and redundant stateful logic inside multiple components. Lets take a look at one example. We have two components, one is Employee Component and the other one is Department Component. Employee Component connects to a Web API, fetch the employee data when the Component is mounted and display the data.

Similarly Department Component connects to a Web API, fetch the departments data when the Component is mounted and display the data.



We might write the same code in both the Components to achieve this. We were able to handle this by creating Render props and Higher Order Components in the case of Class Components.

What if if we want to reuse the code between function components.

Lets see how easily we can do that in the case of function components.

We will create three different components. One is Employee Component, second one is Department Component and the last one is App Component.

We have developed these components previously, lets paste the components one by one.

We can create a Javascript function, write the reusable code in that function and this function can be used in any component where we want to reuse that logic.

Lets create function called as useList and this function accepts Web API url as a parameter,

Now lets create a state variable in which we will store the list data and we will initialize it to empty array.

Now we will call the API with in useEffect hook. I have the code handy and pasting it here.

Now we will return the list from this function. Remember that it is a Javascript function and we can return anything as we want.

With in that Javascript function, as we are using other React hooks then that function will be referred as Custom Hook.

Now we can call this hook from both Employee Component and Department Component.

Save the Changes, navigate to the browser. We can see that both Employees data and departments data being displayed.

Building our own Hooks lets us extract component logic into reusable functions.

**A custom Hook is a JavaScript function whose name starts with ”use” and that may call other Hooks.** It is a convention that we will start the hook name with **use**else we will be violating the rules of Hooks. We will discuss later about rules of Hooks.

A custom Hook doesn’t need to have a specific signature. We can decide what it takes as arguments, and what, to return.

One hook can be used by multiple components as we have seen here, and every time we use a custom Hook, all state and effects inside of it are fully independent from one component to the other component.

import React, { useState, useEffect } from 'react'

import ReactDOM from 'react-dom'

function useList(url){

const [data,setData]=useState([]);

useEffect(()=>{

fetch(url)

.then(res => res.json())

.then(

(result) => {

setData(result);

}

);

});

return data;

}

function Employee(){

const employees=useList("https://localhost:44306/api/Employee");

return (

<div>

<h2>Employees Data...</h2>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

<th>Location</th>

<th>Salary</th>

</tr>

</thead>

<tbody>

{employees.map(emp => (

<tr key={emp.Id}>

<td>{emp.Id}</td>

<td>{emp.Name}</td>

<td>{emp.Location}</td>

<td>{emp.Salary}</td>

</tr>

))}

</tbody>

</table>

</div>

);

}

function Department(){

const departments=useList("https://localhost:44306/api/Dept");

return (

<div>

<h2>Department Data...</h2>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

</tr>

</thead>

<tbody>

{departments.map(emp => (

<tr key={emp.Id}>

<td>{emp.Id}</td>

<td>{emp.Name}</td>

</tr>

))}

</tbody>

</table>

</div>

);

}

function App(){

return(

<div>

<Employee></Employee>

<Department></Department>

</div>

)

}

const element=<App></App>

ReactDOM.render(element,document.getElementById("root"));

# Custom Hooks Part-2

In this article, we will continue our discussion about Custom hooks in React.

In the last article, we have discussed about How to Create Custom Hooks in react and how to Reuse the Code.

There will be many common scenarios which have to be implemented across multiple React Projects. Like for example, assume that you have opened a mobile app from your phone and trying to do some activity. Your mobile network is very Poor or you have lost the Internet connection. Then the app shows us that message you don’t have internet connection. Please check and try again.

We see this functionality in multiple Mobile Apps. Rather than writing this functionality in every Project, we might want to reuse the same across multiple Projects.

There are hundreds of such common examples we come across when we are developing multiple Projects.

 Lets say we have been asked to create a custom React hook using which we will be able to perform speech-to-text conversion.

After the Hooks have been introduced in React, code reusability has reached altogether a new level.

so before we start writing your own custom hooks there is a VERY high possibility that someone has already written it and put it on npm.

There are many Custom hooks available on npm which can be used in our Projects rather than implementing our own one.

We can visit npmjs.com, we can search for packages like @rehooks/online-status or , react-speech-kit.

We can install that respective package into our Project and use that hook in our Project straightaway.

It’s a great way for us and for our team to maximize code reuse and speed up development.

Now lets install one such custom hook from npm into our demo-project, using which we can do speech-to-text conversion.

npm i react-speech-kit

Now lets Open index.js file from our demo-project.

Import useSpeechRecognition from React-speech-kit.

Lets create one function component and we will name it as App.

Lets create one state variable in which we will store the text.

Now we will call **useSpeechRecognition**hook.

The useSpeechRecognition returns an object that contains:

* **listen**: a function that tells the browser to listen for audio coming from the mic.
* **stop**: a function that cancels listening for input coming from the mic.

To this useSpeechRecognition  hook, we can pass one object. To that object, we can pass a function to **onResult** property. This function takes the search to text conversion result as its input, and we can update the text to our state variable.

Lets return a div container. We will place a text area element in which we will display the converted text.

Lets add two buttons, one will be used to listen for the speech and the other one will be used to stop listening to the speech.

Lets call this Component and will render that element to the root container.

Save the changes, navigate to the browser.

One can see that as we keep speaking that will be displayed as text in the textarea.

There are many such hooks are available over npm one can make use of.

Custom Hooks allows us to reuse the stateful logic to the greatest extent.

import React, { useState } from 'react'

import ReactDOM from 'react-dom'

import {useSpeechRecognition} from 'react-speech-kit'

function App(){

const [text,setText]=useState();

const {listen,stop}=useSpeechRecognition({

onResult:result=>setText(result)

});

return(

<div>

<h2>Converting the Speech to Text...</h2>

<textarea value={text}></textarea>

<p>

<button onClick={listen}>Listen</button>

<button onClick={stop}>Stop</button>

</p>

</div>

)

}

const element=<App></App>

ReactDOM.render(element,document.getElementById("root"));

# Routing

In this article, we will introduce ourselves to a Concept Called as Routing and we will also understand how to use Routing in our React Projects.

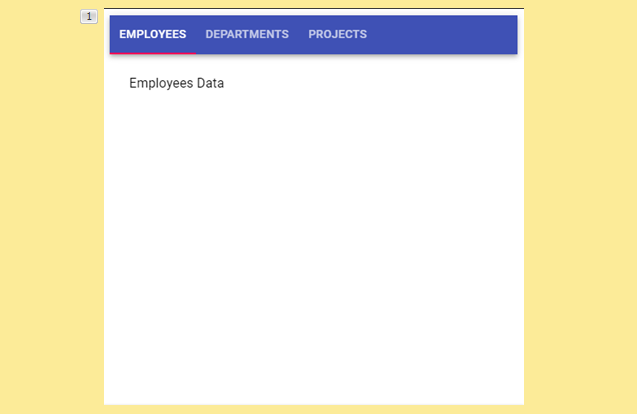
What is Routing?

we know that Components are the basic building blocks of our react applications. We will be developing multiple components as Part of working on any Project, like how we have created Employee Component, Department Component and Project Components.

When we create multiple components, we don’t show all these components together to the User.

 we change what the user sees by showing or hiding portions of the display that correspond to particular components based on User Activities.

As like in this Example, we will show Employee Component when we are in Employee Tab, When we Click on Department Tab, we show the Contents of Department Component and When we Click on Project Tab, we will display the contents of Project Component.



To implement this kind of navigation within the single page of our application, we use the concept of Routing.

Routing enables navigation from one view to the next view (within the same page) as users perform different tasks.

Lets Open index.js file from our demo-project to understand how we can implement Routing in our demo-Project.

Till now we were creating all our Components in our index.js file for our ease of access.

We will create three different Components called as Employee, Department and Project. But now we will create them in separate javascript file.

Lets start with the first One that is Employee Component. Lets create a new javascript file in our src folder and we will name it as employee.js.

I have the Employee Component Code handy and Pasting it here.

We have to export this function from this script file, so that this function can be imported in the other script files where it is needed.

So we write **export default Employee**

We will repeat the Same for rest of the two Components as Well.

As we see here, we have created three different components in three different js files.

Now lets go to index.js file.

Lets create one Component Called as App. We will render this Component to our root container.

Save the changes, navigate to the browser.

We can see the output.

Now here, we wanted to Place three different links using which user should be able to Navigate to the respective Components. It can be Employee or Department or Project.

This is where we will make use of a concept Called as routing.

In Order to use Routing in our demo-project, we have to install Routing related Packages into our Project.

We use **React-Router** which is a Popular Library for implementing Routing for our Project.

Based on whether we are developing a Web App or Mobile App, we will install either react-router-dom or react-router-native.

For a web app, we’ll use **react-router-dom**

For a Mobile app, we’ll use react-router-native.

Since ours is a Web App, lets install react-router-dom from our node js command Prompt.

**npm install react-router-dom**

Now the Installation is Completed Successfully.

Lets run our demo-project using npm start.

Lets switch back to Visual Studio Code.

Lets import the required classes from react-router-dom in order to implement routing.

We are importing BrowserRouter, Switch, Route and Link. In a while we will understand the importance and usage of each one of this.

We will import our Employee, Department and Project Components as well.

Lets go to return method of our App Component.

Now we wanted the li elements to be shown as anchor tags so that user can navigate to different components by clicking on these anchor tags.

React Router provides a <**Link**> component to create links in your application. Wherever you render a <Link>, an anchor (<a>) will be rendered in your HTML document.

We wanted Employee Component to be displayed by default so, we just use / .

Now we will do the same for Department and for Project. But we change the value of to attribute. We access Departments Component by using /departments and Project component using /projects

Now we wanted someone to understand the url and display the respective component contents.

To each **Route**we specify the path and the component.

In order to enable the Routing for our App Component, typically we wrap our App Component in a Router.

React-Router-dom Provides BrowserRouter Component as the Router Component.

Save the Changes, navigate to the browser. We can click on different links and we see that we get the Contents of different Components.

As we click on each link, the url changes. We have configured that using Link Component.

**employee.js:**

import React from 'react';

import { useState, useEffect } from "react";

function Employee() {

const [employees,setEmployees]=useState([]);

useEffect(()=> {

fetch("https://localhost:44306/api/Employee")

.then(res => res.json())

.then(

(result) => {

setEmployees(result);

}

);

});

return (

<div>

<h2>Employees Data...</h2>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

<th>Location</th>

<th>Salary</th>

</tr>

</thead>

<tbody>

{employees.map(emp => (

<tr key={emp.Id}>

<td>{emp.Id}</td>

<td>{emp.Name}</td>

<td>{emp.Location}</td>

<td>{emp.Salary}</td>

</tr>

))}

</tbody>

</table>

</div>

);

}

export default Employee

**Department.js:**

import React from 'react';

import { useState, useEffect } from "react";

function Department() {

const [departments,setDepartments]=useState([]);

useEffect(()=> {

fetch("https://localhost:44306/api/Dept")

.then(res => res.json())

.then(

(result) => {

setDepartments(result);

}

);

});

return (

<div>

<h2>Departments Data...</h2>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

</tr>

</thead>

<tbody>

{departments.map(d => (

<tr key={d.Id}>

<td>{d.Id}</td>

<td>{d.Name}</td>

</tr>

))}

</tbody>

</table>

</div>

);

}

export default Department

**App.js**

import React from 'react';

import ReactDOM from 'react-dom';

import {BrowserRouter, Link, Switch, Route} from 'react-router-dom';

import Employee from './employee';

import Department from './department';

import Project from './project';

function App(){

return(

<div>

<h2>Welcome to App Component...</h2>

<ul>

<li><Link to="/employees">Employees</Link></li>

<li><Link to="/departments">Departments</Link></li>

<li><Link to="/projects">Projects</Link></li>

</ul>

<Route path="/employees" component={Employee}></Route>

<Route path="/departments" component={Department}></Route>

<Route path="/projects" component={Project}></Route>

</div>

)

}

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

# Routing Part-2

In this article, we will continue discussing about Routing. This is Continuation to our previous article.

In the Code we have developed during our Previous article. We have created multiple components, placed multiple link components using which we have specified the Navigation URL’s and we have added multiple Route Components who will display the contents of a Component based on the URL Match.

If we look at the output, we can observe that by default we don’t get contents of any Component. We wanted Employee Component Contents should be displayed by default and the rest of the Navigation should remain same.

We will change the Navigation URL of the Link Component of Employees to**/**.

Save the Changes, navigate to the browser. We can see that we get the Employees Data by default.

But it has a Problem, lets click on departments link. We see that both Employees Data and Department data are being displayed.

That is because the Path of departments is /department. It has forward slash in it. So it is matching two routes.

The same thing happens if we click on Projects Link as well.

But that should not happen.

We can avoid that by adding "**exact**paths" to routes.

Save the changes,  navigate to the browser. We can see that we get the Employees Data by default.

lets click on departments link. We see only Department data. Same with the Case of Projects link as well.

When we click on a link, based on the Navigation URL, React Router tries to identify the Route which is having the matching Path.

As we see here, if it is Departments, we get Department Component contents, if it is Project, we get Project Component Contents.

We have another Problem to address.

Though Router finds a matching Route, it continues to search all the remaining routes as well to find the matching Route.

That means if we have 100 routes added, search will happen on all 100 routes irrespective of the match is found or not.

To understand it better, lets add another route which will be pointing to a Employee Component but lets keep the Path as departments.

Save the changes,  navigate to the browser. Click on Departments, we will get the contents of two components.

Though the second matches the path, still the search is continued further and returning the Contents of other Component as well.

But ideally we don’t want that to happen. If the matching path is found, we want to render that Route Component Contents and ignore the rest.

To make that to happen, we use another component called as **switch**. It works similar to switch case.

Lets place all our route elements with in the boundaries of **switch** component.

When a **<Switch>** is rendered, it searches through its children **<Route>** elements to find one whose path matches the current URL. When it finds one, it renders that <Route> and ignores all others.

Save the changes,  navigate to the browser. Click on Departments, we will get the contents of only Department components.

I hope we are very clear on What is Link, Switch and Route Components. We will continue this discussion in our next article.

function App(){

return(

<div>

<h2>Welcome to App Component...</h2>

<ul>

<li><Link to="/">Employees</Link></li>

<li><Link to="/departments">Departments</Link></li>

<li><Link to="/projects">Projects</Link></li>

</ul>

<Switch>

<Route exact path="/" component={Employee}></Route>

<Route path="/departments" component={Department}></Route>

<Route path="/projects" component={Project}></Route>

<Route path="/departments" component={Employee}></Route>

</Switch>

</div>

)

}

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

# Routing Part-3

In this article, we will continue discussing about Routing. This is Continuation to our Previous.

If we take a look at the Output of Program, we have three different links and we are able to Navigate to different Components.

Lets say we want to add some styling attributes to the Particular link when it is active, so it becomes easy for a user to identify which link he has Clicked.

React Router Provides another Component Called as NavLink which is a  special type of <Link> that can style itself as “active” when its to property value matches the current location.

Lets open index.html and add a new css class.

<style>

.testClass{

font-weight:bold;

color: red;

}

</style>

Lets go back to index.js, we will import NavLink Component.

For Departments and Projects, instead of using Link Component, lets use NavLink and we will use a New Property called as activeClassName. To this we will pass the css class we created.

<ul>

<li><Link to="/">Employees</Link></li>

<li><NavLink to="/departments" activeClassName="testClass">Departments</NavLink></li>

<li><NavLink to="/projects" activeClassName="testClass">Projects</NavLink></li>

</ul>

Save the Changes, Navigate to the browser. When we click on Departments link, we can see that that link is styled.

There are some possibilities that at times our application might generate some broken links or if user go and directly enter the Navigation URL, if that component doesnot exist, we can redirect users to specific component and display the message that this component doesnot exist.

Lets go ahead and Create a Component which is used for Displaying the Page Not Found message.

Now to the List of Routes, add one more Route Element at the end and set the Component value as Error Component. It will not have any specific path as such.

Remember to keep this at the end as this Route doesnot have any path. Else for any Navigation, our router shows up this Error Component.

Order of the Routes do play important role.

<Switch>

<Route exact path="/" component={Employee}></Route>

<Route path="/departments" component={Department}></Route>

<Route path="/projects" component={Project}></Route>

<Route component={InvalidPath}></Route>

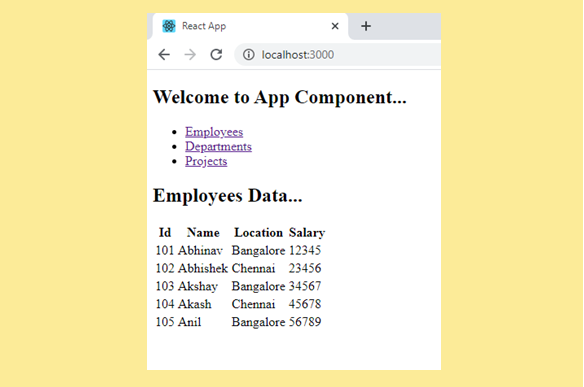
</Switch>

Save the Changes, navigate to the browser. Now if we try to enter any other Navigation URL, Error Component Contents will be displayed.

# Routing Part-4

In this article, we will continue discussing about Routing.

If we take a look at the Output of our Previous Session Code, We are displaying the list of employees in a table.



We would like to Provide Edit Option against each row of the table.

When we click on edit, we should be navigated to edit employee Component. To that Edit Employee Component Component, we have to carry the Employee ID so that we can make an API request using that Id and get the Employee Details.

In this session, we will understand how to create dynamic urls and carry the Employee ID through URL Parameter.

Lets Open index.js file from our demo-project.

Lets go to employee.js file, to the table we want to add a new column using which we can edit the employee.

Lets add a new Header Column.

We will add a New anchor tag, assuming that we will add a new Route with path as /employee/ we will pass employee id  as another parameter to the Route.

<a href={'/employee/'+emp.Id}>Edit</a>

Lets add new javascript file,  create a New Component which will receive this id, make a Web API Call by passing the Employee ID, get the Employee details and display for Edit.

This Component receives one URL parameter using which we will get the id.

I have the Contents of this Component available handy and Pasting it here.

Within useEffect hook, we will send a Web API Request.

To the Web API, we pass the Employee ID as Parameter and we get the id  value using

**{props.match.params.id}**

Upon receiving the response from API, we will update our employee state variable.

**EditEmployee.js**

import React from 'react';

import { useState, useEffect } from "react";

function EditEmployee(props) {

const [employee,setEmployee]=useState({});

useEffect(()=> {

fetch("https://localhost:44306/api/Employee/"+props.match.params.id)

.then(res => res.json())

.then(

(result) => {

setEmployee(result);

}

);

});

function changeEmployeeData(e){

}

return (

<div>

<h2>Employee Details...</h2>

<p>

<label>Employee ID : <input type="text" name="Id"

value={employee.Id} onChange={changeEmployeeData}></input></label>

</p>

<p>

<label>Employee Name : <input type="text" name="Name"

value={employee.Name} onChange={changeEmployeeData}></input></label>

</p>

<p>

<label>Employee Location : <input type="text" name="Location"

value={employee.Location} onChange={changeEmployeeData}></input></label>

</p>

<p>

<label>Employee Salary : <input type="text" name="Salary"

value={employee.Salary} onChange={changeEmployeeData}></input></label>

</p>

<button>Update</button>

</div>

);

}

export default EditEmployee

Now lets go to index.js file. Import the EditEmployee Component.

Lets add a New Route, to this Route path, we will pass two segments. First segment is employee and we will pass id as the next segment. We want to make id as dynamic. So we write

<Route path="/employee/:id" component={EditEmployee}></Route>

 Save the Changes, navigate to the browser. We can click on edit link of any employee, we will be displaying the employee details in edit mode.

One can see that id is carried as part of navigation url.

Lets click on Back button. We can click on edit link of any other row, we can see that respective employee details are being displayed in Edit mode.

If required we can pass multiple Parameter values through the URL.

# Routing Part-5

In this article, we will continue discussing about Routing.

If we take a look at the Output of our Previous Session Code, We are displaying the list of employees in a table.

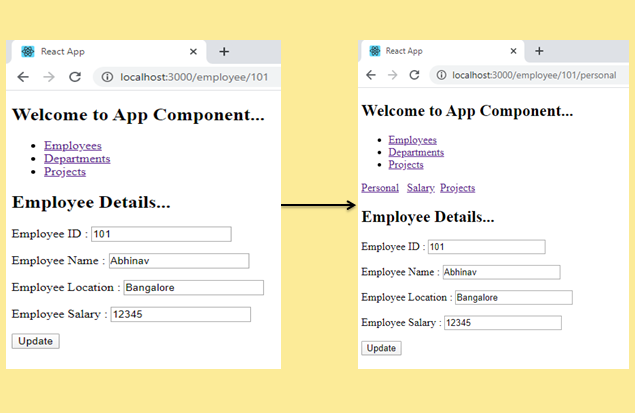
When we click on edit, we are navigated to edit employee Component and we are displaying the Employee Details.

Now what if if we want to display the Employee Data in different tabs.

Employee Personal Information in One tab

Employee Salary Information in One Tab

Employee Projects Information in One Tab.



That means we have to Add route Components to our EditEmployee Component and provide a way for users to navigate between the Tabs.

In this session, we will understand how to create Nested Routes and how do we Navigate.

Let’s Open EditEmployee.js file from our demo-project.

Let’s Create three Components.

The first one is EmployeePersonalInfo, lets cut the code from EditEmployee Component and we will paste it in EmployeePersonalInfo Component.

We will create the remaining two components. They are EmployeeSalaryInfo and EmployeeProjectInfo, I have the code handy and pasting it here.

Now lets go to EditEmployee Component.

Lets return a div container,

Lets add the Navigation using Link Component.

One Link Component is for Personal Details Tab, One is for Salary Details Tab and the other One is for Project Details Tab.

We wanted the Personal Details tab to be displayed by default. So lets add to attribute for each Link Component. So the Navigation url is going to be /employee/ employee id. We get the employee id from the url parameter using props.match.params.id

But For Salary Link Component to attribute value, we add /salary and for Project Link Component to attribute value, we add /projects.

Lets add the Route Components required. I have the Code handy and Pasting it here.

Save these Changes, navigate to the browser.

Lets click on edit of any employee record, we can see that we have three different links again. We can navigate to different Components by Clicking on different links.

**EditEmployee.js**

import React from 'react';

import { useState, useEffect } from "react";

import {Link,NavLink, Switch, Route} from 'react-router-dom';

function EmployeeSalaryInfo() {

return (

<h2>Employee Salary Details...</h2>

);

}

function EmployeeProjectInfo() {

return (

<h2>Employee Project Details...</h2>

);

}

function EmployeePersonalInfo(props){

const [employee,setEmployee]=useState({});

useEffect(()=> {

fetch("https://localhost:44306/api/Employee/"+props.match.params.id)

.then(res => res.json())

.then(

(result) => {

setEmployee(result);

}

);

});

function changeEmployeeData(e){

}

return (

<div>

<h2>Employee Details...</h2>

<p>

<label>Employee ID : <input type="text" name="Id"

value={employee.Id} onChange={changeEmployeeData}></input></label>

</p>

<p>

<label>Employee Name : <input type="text" name="Name"

value={employee.Name} onChange={changeEmployeeData}></input></label>

</p>

<p>

<label>Employee Location : <input type="text" name="Location"

value={employee.Location} onChange={changeEmployeeData}></input></label>

</p>

<p>

<label>Employee Salary : <input type="text" name="Salary"

value={employee.Salary} onChange={changeEmployeeData}></input></label>

</p>

<button>Update</button>

</div>

);

}

function EditEmployee(props) {

return(

<div>

<Link to={"/employee/"+props.match.params.id}>Personal</Link> &nbsp;&nbsp;

<NavLink to={"/employee/"+props.match.params.id+"/salary"} activeClassName="testClass">Salary</NavLink>&nbsp;&nbsp;

<NavLink to={"/employee/"+props.match.params.id+"/projects"} activeClassName="testClass">Projects</NavLink>

<Switch>

<Route exact path="/employee/:id" component={EmployeePersonalInfo}></Route>

<Route path="/employee/:id/salary" component={EmployeeSalaryInfo}></Route>

<Route path="/employee/:id/projects" component={EmployeeProjectInfo}></Route>

</Switch>

</div>

)

}

export default EditEmployee

If we observe our Route Components, for every Route Component Path, we have passed the Hardcoded path as /employee/:id, instead of doing that if we want to read the current url and append the rest of the path here, we can write

path={props.match.url+"/salary"}

The same can be done for projects as well.

One can observe that the Navigation url’s remain same.

# Protected Routes

In this article, we will continue discussing about Routing.

In our last article, we have discussed about Programmatic navigation in React.

If we look at the Output of our routing Program we have developed in the last session, we have three components named login, home and editprofile. When we enter the valid credentials we are navigating users to home component. But our application is having an issue. home or edit profile components can still be accessed by Users who don’t have the LogIn Credentials.

In this article, we will discuss about how to **Protect our routes** so that only Logged In Users can access Home or Edit Profile Components.

Lets Open index.js file from our demo project.

When user clicks on home link or edit profile, we have to verify if user is already logged in or not. If user is logged in, we will allow user to navigate else we will redirect user to Login Component.

Lets create one object which will manage the user authentication related part.

We will create one property isLoggedIn using which we will track if the User is logged in or not and we will initialize it to false.

Lets add a method called as onAuthenticate and with in this method, we will set isLoggedIn to true.

Lets create another method from which we will return isLoggedIn value.

We will be creating a component called SecuredRoute  which would accept the component to which application has to route if the user is authenticated. Otherwise user will be redirected to login component.

We use Render Properties, we will check if user is logged in or not. If user is Logged in , we will render that respective component else we will redirect to login Component.

Now lets go to App Component, lets secure our home component and edit profile component by using our securedroute component.

Save the changes, navigate to the browser.

Now try to access home or edit profile components, we will not be able to access them.

Now enter valid credentials, we will be redirected to home component. From there we can access edit profile component.

import React, { useState, Component } from 'react';

import ReactDOM from 'react-dom';

import {BrowserRouter, Link, NavLink, Switch, Route, Redirect} from 'react-router-dom';

const authentication={

isLoggedIn:false,

onAuthentication(){

this.isLoggedIn=true;

},

getLogInStatus(){

return this.isLoggedIn;

}

}

function SecuredRoute(props){

return(

<Route path={props.path} render={data=>authentication.getLogInStatus()?(

<props.component {...data}></props.component>):

(<Redirect to={{pathname:'/'}}></Redirect>)}></Route>

)

}

function LogIn(props){

const [loginData,setLoginData]=useState({username:'',password:''});

function changeLogInData(e){

setLoginData({...loginData,[e.target.name]:e.target.value});

}

function onLogIn(){

fetch("https://localhost:44306/api/Test",{

method:'POST',

headers:{'Content-type':'application/json'},

body:JSON.stringify(loginData)

}).then(r=>r.json()).then(result=>{

if(result){

authentication.onAuthentication();

props.history.push('/home');

}

else{

alert('Invalid UserName or PassWord');

}

});

}

return(

<div>

<h2>Welcome to LogIn...</h2>

<p>

<label>UserName : <input type="text" value={loginData.username}

name="username" onChange={changeLogInData}></input></label>

</p>

<p>

<label>PassWord : <input type="text" value={loginData.password}

name="password" onChange={changeLogInData}></input></label>

</p>

<button onClick={onLogIn}>LogIn</button>

</div>

)

}

function Home(props){

function onNext(){

props.history.replace('/editprofile');

}

return(

<div>

<h2>Welcome to Home...</h2>

<button onClick={onNext}>Next</button>

</div>

);

}

function EditProfile(){

return(

<div>

<h2>Welcome to Edit Profile...</h2>

</div>

);

}

function App(){

return(

<div>

<h2>Welcome to App Component...</h2>

<Link to="/">LogIn</Link>&nbsp;&nbsp;

<NavLink to="/home" activeClassName="testClass">Home</NavLink>&nbsp;&nbsp;

<NavLink to="/editprofile" activeClassName="testClass">Edit Profile</NavLink>

<Switch>

<Route exact path="/" component={LogIn}></Route>

<SecuredRoute path="/home" component={Home}></SecuredRoute>

<SecuredRoute path="/editprofile" component={EditProfile}></SecuredRoute>

</Switch>

</div>

)

}

ReactDOM.render(<BrowserRouter><App></App></BrowserRouter>,

document.getElementById("root"));

# Prevent Navigation

In this article, we will continue discussing about Routing.

In our last article, we have discussed about how to Protect our routes so that only Logged In Users can access Home or Edit Profile Components.

If we look at the Output of our routing Program we have developed in the last session, we have three components named login, home and editprofile. When we enter the valid credentials we are navigating users to home component. From home Component, user can navigate to Edit Profile Component as well.

Now assuming that EditProfile Component looks this way, user can change his Profile details and update them by Clicking on Update.

Now imagine that we made some changes to the User Profile details and by mistake we clicked on home tab. Then we get navigated to home component. Now if we go back to edit profile component, we don’t find our changes. Its because the component gets unmounted when we go away from a component. But it would have been good if our application prompts a notification to the user that there are some unsaved changes. Do you really want to go away. So that user can take appropriate action.

In this article, we will understand **prevent the user from navigating** away from a page when there are some **unsaved changes**.

Lets open index.js file from our demo-project.

Lets go to Edit Profile Component, create one state variable which holds user information like first name, last name and email Id.

Lets add the required input elements and let the input elements display the user details by reading from the state variable.

We will add a button to update the user details.

We will implement the onDataChange function and with in that function, we will update the state object.

Lets add one more state variable using which we can track if the data is changed by the user or not and we will initialize it to false.

When there is a change of user data, we will set isDataChanged to true.

We will set this to false again when user clicks on update button.

we use Prompt Component to prompt a notification when the transition happens.

We will import prompt from react router dom.

We show the Prompt component when we have unsaved changes. We can pass the message as well.

Save the changes, navigate to the browser.

Navigate to the edit profile component. Make some changes to the user profile. Now if we click on home link, we get the notification.

We can save the user details. Now if we click on home link, we get navigated without any notification.

import React, { useState, Component } from 'react';

import ReactDOM from 'react-dom';

import {BrowserRouter, Link, NavLink, Switch, Route, Redirect, Prompt} from 'react-router-dom';

const authentication={

isLoggedIn:false,

onAuthentication(){

this.isLoggedIn=true;

},

getLogInStatus(){

return this.isLoggedIn;

}

}

function SecuredRoute(props){

return(

<Route path={props.path} render={data=>authentication.getLogInStatus()?(

<props.component {...data}></props.component>):

(<Redirect to={{pathname:'/'}}></Redirect>)}></Route>

)

}

function LogIn(props){

const [loginData,setLoginData]=useState({username:'',password:''});

function changeLogInData(e){

setLoginData({...loginData,[e.target.name]:e.target.value});

}

function onLogIn(){

fetch("https://localhost:44306/api/Test",{

method:'POST',

headers:{'Content-type':'application/json'},

body:JSON.stringify(loginData)

}).then(r=>r.json()).then(result=>{

if(result){

authentication.onAuthentication();

props.history.push('/home');

}

else{

alert('Invalid UserName or PassWord');

}

});

}

return(

<div>

<h2>Welcome to LogIn...</h2>

<p>

<label>UserName : <input type="text" value={loginData.username}

name="username" onChange={changeLogInData}></input></label>

</p>

<p>

<label>PassWord : <input type="text" value={loginData.password}

name="password" onChange={changeLogInData}></input></label>

</p>

<button onClick={onLogIn}>LogIn</button>

</div>

)

}

function Home(props){

function onNext(){

props.history.replace('/editprofile');

}

return(

<div>

<h2>Welcome to Home...</h2>

<button onClick={onNext}>Next</button>

</div>

);

}

function EditProfile(){

let [isDataChanged,setDataChanged]=useState(false);

let [userInfo,setUserInfo]=useState({firstName:'SpringPeople',

lastName:'Technologies',

userName:'SpringPeople Tech',

emailId:'SpringPeople@gmail.com',

contactNo:'+91-9945699393'});

function saveChanges(){

setDataChanged(false);

}

function onUserDataChange(e){

setUserInfo({...userInfo,[e.target.name]:e.target.value});

setDataChanged(true);

}

return(

<div>

<Prompt when={isDataChanged}

message={()=>"There are Some Unsaved Changes. Do you want to go Away?"}></Prompt>

<h2>Welcome to Edit Profile...</h2>

<p>

<label>First Name : <input type="text" name="firstName"

onChange={onUserDataChange}

value={userInfo.firstName}></input></label>

</p>

<p>

<label>Last Name : <input type="text" name="lastName"

onChange={onUserDataChange} value={userInfo.lastName}></input></label>

</p>

<p>

<label>User Name : <input type="text" name="userName"

onChange={onUserDataChange}

value={userInfo.userName}></input></label>

</p>

<p>

<label>Email ID : <input type="text" name="emailId"

onChange={onUserDataChange}

value={userInfo.emailId}></input></label>

</p>

<p>

<label>Contact Number : <input type="text" name="contactNo"

onChange={onUserDataChange}

value={userInfo.contactNo}></input></label>

</p>

<button onClick={saveChanges}>Save</button>

</div>

);

}

function App(){

return(

<div>

<h2>Welcome to App Component...</h2>

<Link to="/">LogIn</Link>&nbsp;&nbsp;

<NavLink to="/home" activeClassName="testClass">Home</NavLink>&nbsp;&nbsp;

<NavLink to="/editprofile" activeClassName="testClass">Edit Profile</NavLink>

<Switch>

<Route exact path="/" component={LogIn}></Route>

<SecuredRoute path="/home" component={Home}></SecuredRoute>

<SecuredRoute path="/editprofile" component={EditProfile}></SecuredRoute>

</Switch>

</div>

)

}

ReactDOM.render(<BrowserRouter><App></App></BrowserRouter>,

document.getElementById("root"));

# Redux Part-1

**State management** is absolutely critical in a Web Application Development.

We have discussed about how we manage state in a react application using State and Context.

**State**contains data specific to a given component that may change over time.

Using **Context**, we pass the data from parent component to Child component and from Child to Parent which are placed at different nesting levels.

For low-frequency updates like locale or theme changes or user authentication, the React Context is perfectly fine. But with a more complex state object like products in the shopping cart which has high-frequency updates, the React Context won't be a good solution. Because, the React Context will trigger a re-render on each update, and optimizing it manually can be really tough.

**Redux** provides a solid, stable and mature solution to managing state in your React application.

If we have components that are siblings and need to share data, the way to do that in React is to pull that data up into a parent component and pass it down with props.

That can be cumbersome though. Redux can help by giving you one global “parent” where you can store the data, and then you can connect the sibling components to the data with React-Redux.

In this article, we will start understanding Redux.

Lets Open Index.js file from our demo-project.

Add a New Javascript file in our src folder and we will name it as Employee.

I have the Contents of this Component handy and I Paste it here.

We will create a new Component in our index.js file called AppComponent and we will call our Employee Component from AppComponent.

We will render the AppComponent to our ReactDOM.

Save the Changes, navigate to the browser.

We can see that Employee Salary gets changed as we click on increase or decrease buttons.

To start understanding **Redux**, we will re-write the same example using redux so that one can understand the basics of redux so well before we get into complex examples.

 Lets install redux into our project by running a command

**Npm install –save redux react-redux**

redux gives you a store, and lets you keep state in it, and get state out, and respond when the state changes. But that’s all it does.

It’s actually react-redux that lets you connect data of the state to React components.

The redux library can be used outside of a React app too. It’ll work with Vue or Angular apps as well.

Lets create one Global store using Redux where we are going to keep our data.

Redux comes with a handy function that creates stores, and it’s called **createStore**.

To this createStore, We have to provide a function that will return the state data based on the action user performs. This function takes two parameters, one is the state data and the other one is action. That function is called a **reducer**in Redux terminology.

In short, reducer is a function that takes the current state, and an action, and returns the newState

We can create an employee object and pass that to the state parameter.

**Action** is a JavaScript object. Action will have a type property that indicates the type of action being performed. Types should typically be defined as string constants.

We will write switch case on the action type and accordingly increment or decrement the salary.

And we return the state object.

Now it’s time to hook up Redux to React.

To do that, the react-redux library comes with 2 things: a component called **Provider**, and a function called **connect**.

By wrapping the entire app with the Provider component, every component in the app tree will be able to access the Redux store if it wants to.

After this, Employee Component, and children of Employee, and children of their children, and so on – all of them can now access the Redux store.

But not automatically. We’ll need to use the connect function on our components to access the store.

Lets go to Employee.js file,

Lets remove the state object and setstate calls we are doing.

We can access the salary from the redux store using the props.

When user clicks on increment or decrement button, we initiate the state change trigger by calling dispatch method. To the dispatch method, we pass the type.

We do the same in decrement function as well.

To get the Salaryout of Redux, we first need to import the connect function.

Then we need to “connect” the Employee component to Redux at the bottom.

To the connect function, we pass another function as a parameter and that function takes state object as parameter and we can return the salary.

Connect is a higer order Component to which we pass our Employee Component

Save the Changes, navigate to the browser.

We can see that Employee Salary gets changed as we click on increase or decrease buttons.

I hope we are clear now on what is action, reducer and store in redux.

We will continue our discussion in our next article.

**Index.js**

import React from 'react';

import ReactDOM from 'react-dom';

import Employee from './Employee';

import {createStore} from 'redux';

import {Provider} from 'react-redux';

const employeeData={

salary:15000

};

const reducer=(state=employeeData,action)=>{

switch(action.type){

case 'INCREMENT':

return {salary:state.salary + 1000};

case 'DECREMENT':

return {salary:state.salary - 1000};

default:

return state;

}

}

const store=createStore(reducer);

const App = () => (

<Provider store={store}>

<Employee></Employee>

</Provider>

);

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

**Employee.js**

/\* eslint-disable no-useless-constructor \*/

import React from 'react';

import {connect} from 'react-redux';

class Employee extends React.Component {

constructor(props){

super(props);

}

incrementSalary = () => {

this.props.dispatch({type:'INCREMENT'});

}

decrementSalary = () => {

this.props.dispatch({type:'DECREMENT'});

}

render() {

return (

<div>

<h2>Welcome to Employee Component...</h2>

<div>

<p>

<label>Employee Salary : <b>{this.props.salary}</b></label>

</p>

<button onClick={this.incrementSalary}>Increment</button>

<button onClick={this.decrementSalary}>Decrement</button>

</div>

</div>

)

}

}

function mapStateToProps(state){

return{

salary:state.salary

};

}

export default connect(mapStateToProps)(Employee);