**Install these extensions prior**

**ES7 React/Redux/GraphQL**

rfc – react function component

rcc - react class component

**Thunder Client** - Thunder Client is a lightweight Rest API Client Extension for Visual Studio Code

**NODE.JS**

Since Javascript runs in browser , to help it to run outside the browser NODE.JS was created.

**NPM**

NPM helps in creating packages inside application

**JSX**

JSX stands for JavaScript XML. It is simply a syntax extension of JavaScript. It allows us to directly write HTML in React (within JavaScript code). It is easy to create a template using JSX in React, but it is not a simple template language instead it comes with the full power of JavaScript.

Inside a Function returning JSX. Only one element should be returned.

function App() {

  return ( <div></div> ); }

You can add a JSX fragment if you want multiple elements.{<> <h1></h1> <div></div> </>}

**Babel** is a JavaScript compiler that includes the ability to compile JSX into regular JavaScript

npm install = to install all the packages in node\_modules

npx create-react-app my-app = create new app

Entry point is in Public folder in index.html. There will be a root component html which will house a component.

**COMPONENTS & PROPS**

Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen. Props should be mostly readonly.

You can pass values from other components through by passing values as props.

e.g. <Navbar title="TextUtils" aboutText="About TextUtils"></Navbar>

import React from 'react'

export default function Navbar(props) {

    return (    {props.title}     )}

To run typechecking on the props for a component, you can assign the special propTypes property. This proptype must be placed inside the component where you are using the props and not in the component where you have passed the props. These basically provide typechecking on the values that are being passed. AND PROVIDE VALIDATION CHECKS

Navbar.propTypes =

{title :  PropTypes.string.isRequired, aboutText: PropTypes.string}

You can also provide default values.

Navbar.defaultProps =

{title :  "Set Title here", aboutText: "Set About Title here"}

**STATE & LIFECYCLE**

State is similar to props, but it is private and fully controlled by the component. More specifically, the state of a component is an object that holds information that may change over the lifetime of the component. State is public property.

* Props are immutable. Once set they can’t be changed
* State is observable. It can hold data that may change over time
* Props can be used in either function or class components
* State is limited to class components
* Props are set by the parent component
* State is updated by event handlers

import React, { useState } from 'react'

export default function TextForm(props) {

    const handleUpClick = ()=>{

*//console.log("Uppercase was clicked" + text);*

        let newText = text.toUpperCase();

        setText(newText);

    }

    const handleOnChange = (event)=>{ **//2)** Specify the event method

*//console.log("On change");*

        setText(event.target.value);

    }

    const [text, setText] = useState("Enter Text here"); **// 1)** declare const state(anyname) in the function but not in return

*// text = "some value"; //wrong way to change values* **//**the setText is updated value or value after changed state

*//setText("new text"); //Correct way*

    return (

        <div>

            <h1>{props.heading}</h1>

            <div className="mb-3">

                <textarea className="form-control" value={text} id="myBox" rows="8" onChange={handleOnChange} ></textarea>

**//3)** For change in value in Textbox area event is used

            </div>

            <button className="btn btn-primary" onClick={handleUpClick}>Convert to UpperCase</button> **//4)** For new setText value

        </div>

    )

}

You can pass multiple values as a single object e.g.

<Student data={{name:'shivam',age:24}} ></Student>

You cannot change values of props in receiving component.

**CLASS COMPONENT WITH STATE**

import React, { Component } from 'react'

export default class asd extends Component {

    constructor(){

        super();

        this.state={

            data:"shivam"

        }

    }

    apple(){

        this.setState({data:"bagi"})

    }

    render() {

        return (

            <div>

                <h1>{this.state.data}</h1>

                <button onClick={apple}></button>

            </div>

        )

    }

}

**React Router DOM**

In angular you have built in routing, but in react you have to install react router DOM package externally.

npm install react-router-dom

You need to import below

import {

BrowserRouter as Router,

Switch,

Route,

Link

} from "react-router-dom";

{/\* A <Switch> looks through its children <Route>s and renders the first one that matches the current URL. \*/}

<Router>

<Switch>

<Route path="/about">

<About />

</Route>

<Route path="/users">

<Users />

</Route>

<Route exact path="/"> **//**Use exact for full matching otherwise react will do partial matching

<Home />

</Route>

</Switch>

</Router>

In react you have to use <Link> instead of <a> and to=”” instead of href=””

**DEPLOYMENT**

npm run build

It will create a build for deployment which will contain static files.

[**GitHub Pages**](https://pages.github.com/)[**#**](https://create-react-app.dev/docs/deployment/#github-pages)

Note: this feature is available with react-scripts@0.2.0 and higher.

**Step 1: Add homepage to package.json**[**#**](https://create-react-app.dev/docs/deployment/#step-1-add-homepage-to-packagejson)

**The step below is important!**

**If you skip it, your app will not deploy correctly.**

\*1) Open your package.json and add a homepage field for your project above name field:

"homepage": "https://myusername.github.io/my-app", *myusername = github name/ my-app = Repository*

or for a GitHub user page:

"homepage": "https://myusername.github.io",

or for a custom domain page:

"homepage": "https://mywebsite.com",

Create React App uses the homepage field to determine the root URL in the built HTML file.

**Step 2: Install gh-pages and add deploy to scripts in package.json**[**#**](https://create-react-app.dev/docs/deployment/#step-2-install-gh-pages-and-add-deploy-to-scripts-in-packagejson)

Now, whenever you run npm run build, you will see a cheat sheet with instructions on how to deploy to GitHub Pages.

\*1) To publish it at <https://myusername.github.io/my-app>, run:

npm install --save gh-pages

npm uninstall gh-pages

Alternatively you may use yarn:

yarn add gh-pages

\*2) Add the following scripts in your package.json:

"scripts": {

"predeploy": "npm run build",

"deploy": "gh-pages -d build",

The predeploy script will run automatically before deploy is run.

**Step 3: Deploy the site by running npm run deploy**

npm run deploy

You may face error like –

Failed to get remote.origin.url (task must either be run in a git repository with a configured origin remote or must be configured with the "repo" option).

You will need to add the origin by below command --

git remote add origin https://github.com/shivambagi/TextUtils-React.git

**Step 4: Setting default page in GitHub Pages**

Goto Repository Settings 🡪 Pages 🡪 Under source Branch should be *gh-pages.*

You cannot add Router React on Github pages directly without changing some settings. It will not work. This is because when there is a fresh page load for a url like *http://user.github.io/todomvc/todos/42*, where */todos/42* is a frontend route, the GitHub Pages server returns 404 because it knows nothing of */todos/42.*

### The Component Lifecycle

Mounting

These methods are called in the following order when an instance of a component is being created and inserted into the DOM:

* [**constructor()**](https://reactjs.org/docs/react-component.html#constructor) - If you don’t initialize state and you don’t bind methods, you don’t need to implement a constructor for your React component. The constructor for a React component is called before it is mounted. When implementing the constructor for a React.Component subclass, you should call super(props) before any other statement. Otherwise, this.props will be undefined in the constructor, which can lead to bugs.

constructor(props) {

super(props);

// Don't call this.setState() here!

this.state = { counter: 0 };

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

}

* [static getDerivedStateFromProps()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromprops)
* [**render()**](https://reactjs.org/docs/react-component.html#render) - render() is used to render HTML of the component in react. This method is required for a class based component to render the DOM. It runs during mounting and updating of component. It should be pure i.e. state should not be modified inside it.
* [**componentDidMount()**](https://reactjs.org/docs/react-component.html#componentdidmount) - componentDidMount() is invoked immediately after a component is mounted and rendered (inserted into the tree). Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request.

Updating

An update can be caused by changes to props or state. These methods are called in the following order when a component is being re-rendered:

* [static getDerivedStateFromProps()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromprops)
* [shouldComponentUpdate()](https://reactjs.org/docs/react-component.html#shouldcomponentupdate)
* [**render()**](https://reactjs.org/docs/react-component.html#render)
* [getSnapshotBeforeUpdate()](https://reactjs.org/docs/react-component.html#getsnapshotbeforeupdate)
* [**componentDidUpdate()**](https://reactjs.org/docs/react-component.html#componentdidupdate) - componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render.

Unmounting

This method is called when a component is being removed from the DOM:

* [**componentWillUnmount()**](https://reactjs.org/docs/react-component.html#componentwillunmount)

Error Handling

These methods are called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.

* [static getDerivedStateFromError()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromerror)
* [componentDidCatch()](https://reactjs.org/docs/react-component.html#componentdidcatch)

**Class Properties**

* [defaultProps](https://reactjs.org/docs/react-component.html#defaultprops) - defaultProps can be defined as a property on the component class itself, to set the default props for the class. This is used for undefined props, but not for null props.

class CustomButton extends React.Component {

// ...

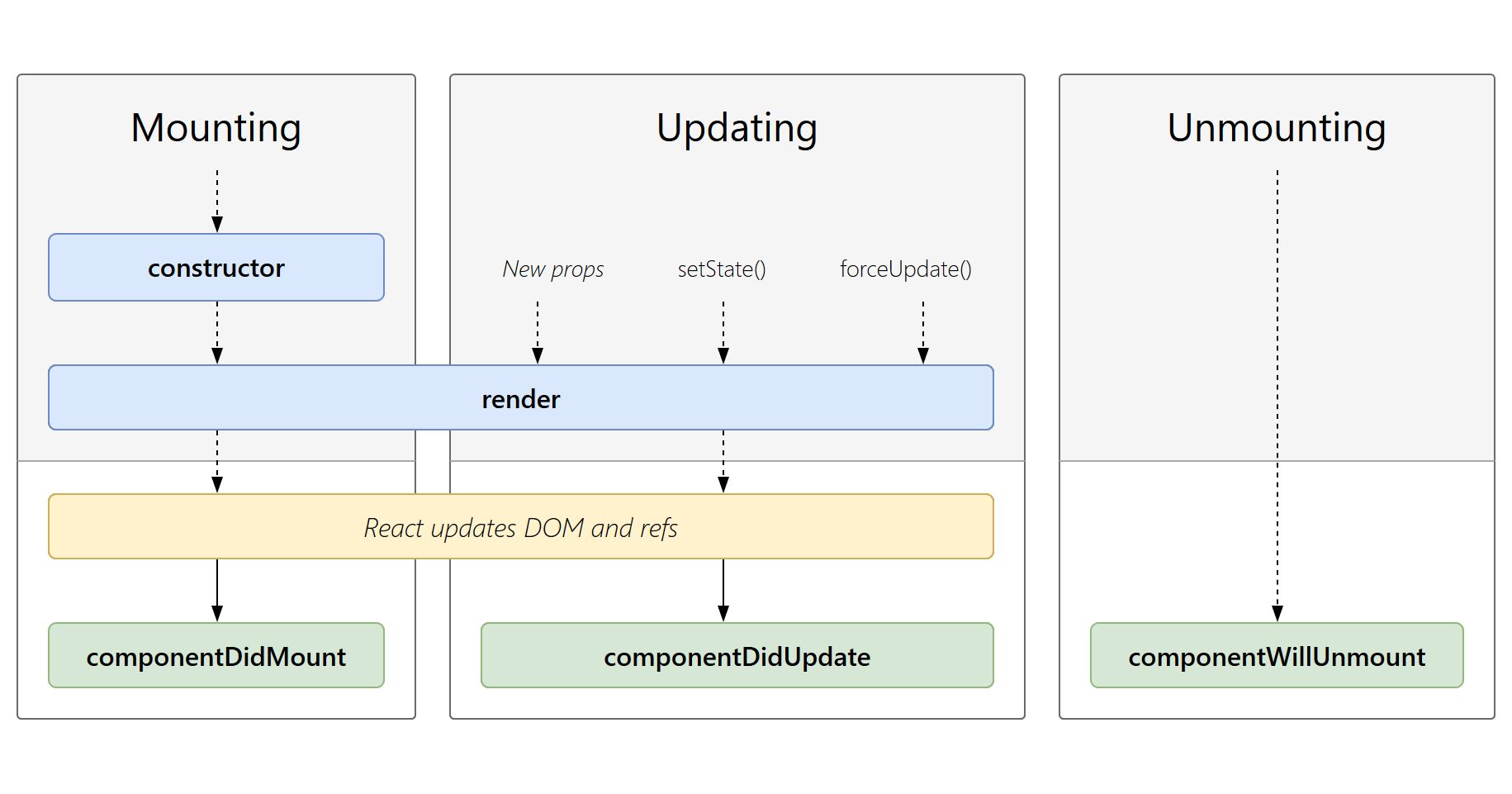
}

CustomButton.defaultProps = {

color: 'blue'

};

* [displayName](https://reactjs.org/docs/react-component.html#displayname)



**HOOKS** They let you use state and other React features without writing a class.

useState - useState is a Hook (we’ll talk about what this means in a moment). We call it inside a function component to add some local state to it. React will preserve this state between re-renders. useState returns a pair: the current state value and a function that lets you update it. You can call this function from an event handler or somewhere else. It’s similar to this.setState in a class, except it doesn’t merge the old and new state together.

useEffect - The Effect Hook, useEffect, adds the ability to perform side effects from a function component. It serves the same purpose as componentDidMount, componentDidUpdate, and componentWillUnmount in React classes, but unified into a single API. It runs after every render cycle or effect on DOM. Like change in input in textbox even if it is a single letter. You can add dependencies for allowing it to run on specific conditions.

Rules – 1) Can be only used inside functions. 2) Only call hooks at the top level

**CONTAINMENT (COMPOSITION)**

Some components don’t know their children ahead of time. This is especially common for components like Sidebar or Dialog that represent generic “boxes”.

We recommend that such components use the special children prop to pass children elements directly into their output:

*function FancyBorder(props) {*

*return (*

*<div className={'FancyBorder FancyBorder-' + props.color}>*

*{props.children}*

*</div>*

*); }*

This lets other components pass arbitrary children to them by nesting the JSX:

*return (*

*<Card className='expense-item'>*

*<ExpenseDate date={props.date} />*

*<div className='expense-item\_\_description'>*

*<h2>{props.title}</h2>*

*<div className='expense-item\_\_price'>${props.amount}</div>*

*</div>*

*</Card>*

*);*

**PASSING DATA FROM CHILD COMPONENT TO PARENT**

1) We create a function inside parent component which accepts data in form of object or singular value.

const onSaveExpenseDataHandler = (enteredExpenseData) => { *// enteredExpenseData is received from child*

const expenseData = { // *expenseData is of parent component*

...enteredExpenseData,

id: Math.random().toString()

};

console.log(expenseData);

}

2) Similar to passing data to child component we pass this function.

<ExpenseForm onSaveExpenseData={onSaveExpenseDataHandler} />

3) Inside the child component we use props and set the data to the passed data variable name.

const submitHandler =(event) =>{

        const expenseData = {

            title: enteredTitle,

            amount: enteredAmount,

            date: new Date(enteredDate)

        }

        props.onSaveExpenseData(expenseData); *// we assign child data using props onSaveExpenseData is parent data*

    }

**PORTALS**

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

ReactDOM.createPortal(child, container)

The first argument (child) is any renderable React child, such as an element, string, or fragment. The second argument (container) is a DOM element.

render() {

// React does \*not\* create a new div. It renders the children into `domNode`.

// `domNode` is any valid DOM node, regardless of its location in the DOM.

return ReactDOM.createPortal(

<SomeElement />,

domNode

);

}

**REFS & THE DOM**

Refs provide a way to access DOM nodes or React elements created in the render method. Refs are a function provided by React to access the DOM element and the React element that you might have created on your own. They are used in cases where we want to change the value of a child component, without making use of props and all. They also provide us with good functionality as we can use callbacks with them.

import { useRef } from 'react'; ***//****import*

const nameInputRef = useRef(); ***//*** *use a variable*

const addUserHandler = (event) => {

event.preventDefault();

const enteredName = nameInputRef.current.value; ***//****getting value from input HTML*

props.onAddUser(enteredName, enteredUserAge); ***//*** *passing to upper element*

nameInputRef.current.value = ''; ***//****setting value of input back to empty*

};

<input id="username" type="text" ref={nameInputRef} /> ***//****you have to use ref in HTML with defined variable*

These are not controlled but react, because state is not controlled.

**EFFECTS, REDUCERS AND CONTEXT**

**REDUCERS**

In React there are two main hooks that are used for state management, and we all know them. These are:

* useState
* useReducer

The useReducer hook is used for complex state manipulations and state transitions. Just like the other React hooks, we can import useReducer from react as shown in the below snippet:

import React, { useReducer } from 'react';

React.useReducer is a React hook function that accepts a reducer function, and an initial state.

const [state, dispatch] = useReducer(reducer, initialState);

This hook function returns an array with 2 values. The first one is the state value, and the second value is the dispatch function which is further used to trigger an action with the help of array destructuring. useReducer is usually preferable to useState when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one.

**CONTEXT API**

The Context API can be used to share data with multiple components, without having to pass data through props manually. For example, some use cases the Context API is ideal for: theming, user language, authentication, etc.

The React Context API is a way for a React app to effectively produce global variables that can be passed around. This is the alternative to "prop drilling" or moving props from grandparent to child to parent, and so on. Context is also touted as an easier, lighter approach to state management using Redux.

**useReducer vs useState**

**GETTING DATA FROM INPUT WITHOUT FORM(onChange & state)**

export default function asd() {

    const [data, setdata] = useState();

    const [print, setprint] = useState(false);

    const getdata=(val)=> {

        console.log(val.target.value); //

        setdata(val.target.value);

        setprint(false)

    }

    const getPrint=(val)=> {

        console.log(val.target.value);

        setdata(val.target.value);

    }

    return (

        <div>

            {

                print?

                <h1>{data}</h1>

                :null

            }

            <input onChange={getdata}></input>

            <h1>{data}</h1>

            <button onClick={}></button>

        </div>

    )

}

**HIDE SHOW ELEMENT OR TOGGLING ELEMENT**

export default function asd() {

const [status, setstatus] = useState(true)

return (

<div>

{status? <h1>Hello World</h1>:null}

<button onClick={setstatus(false)}>Hide</button>

<button onClick={setstatus(true)}>Show</button>

<button onClick={setstatus(!status)}>Toggle</button>

</div>

)

}

**GETTING VALUES FROM FORMS**

export default function asd() {

const [name, setname] = useState("");

const [tnc, settnc] = useState(false);

const [interest, setinterest] = useState("");

getFormData=(event)=>{

event.preventDefault();

console.log(name,tnc,interest); //getting onchange values from controls and DIRECTLY SAVING THEM IN STATE

}

return (

<div>

<form onSubmit={getFormData}>

<input type="text" onChange={(e)=>setname(e.target.value)}></input> <br/> <br/>

<select onChange={(e)=>setinterest(e.target.value)}>

<option>Select Option</option>

<option>Marvel</option>

<option>DC</option>

</select><br/> <br/>

<input type="checkbox" onChange={(e)=>settnc(e.target.checked)}>Accept terms</input><br/> <br/>

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

</form>

</div>

)

}

**CONDITIONAL RENDERING**

Ternary Operator (condition?valuetrue:false)

export default function asd() {

const [loggedIn, setloggedIn] = useState(1)

return (

<div>

{loggedIn==1? <h1>Welcome User</h1> :

loggedIn==2? <h1>Welcome User2</h1> :

<h1>Welcome User3</h1>}

</div>

)

}