# Understanding ReactJS Component Lifecycle

# ReactJS Component Mounting Methods

* These methods are called when the component is being rendered for the first time.
* Whenever a component is rendered the first method called is “constructor”.
* In the constructor we can provide default/initial data for component state and props.
* After constructor, Mounting Methods will be called. They are
  1. **static getDerivedStateFromProps**:
     + It is called just before render method.
     + It takes two parameters props and state and it should return an object with data or null for empty object to update the state.
  2. **render**:
     + It is called after method componentWillMount. It simply renders the react component onto the DOM by using props and state of the component.
  3. **componentDidMount**:
     + It is called after method render.
     + In this method we can do stuff like calling REST API to update the state, etc.

### ReactJS Component Updating Methods

* We know ReactJS is very quick to user actions. Some times we need to updated DOM based on user actions. To update the React DOM with respect to user actions we use ReactJS component updating methods. They are
  1. **shouldComponentUpdate**:
     + It is called before component re-renders right after change in props of state.
     + It’s a boolean method which tells react to re-render the DOM or not. Default return value for this method is true.
  2. **render**:
     + It will update the DOM with new props and new state.
  3. **componentDidUpdate**:
     + It is called immediately after updating occurs. It is not called for the initial render.
     + It works just like method componentDidMount.

### ReactJS Component Unmounting Methods

* It has only one method **componentWillUnmount** which executed just before ReactJS component removed.
* It is called as “cleanup method”. Because, here we can remove the unused data, unwanted network requests, etc.
* We should not call method setState here because the component will never be re-rendered.

### Example of all above method

import React from "react";

class LifeCycleMethods extends React.Component{

constructor(props){

super(props)

console.log('constructor')

this.state = {"username": ""}

this.handleChange = this.handleChange.bind(this);

}

// component methods

// ReactJS Component Mounting Methods

static getDerivedStateFromProps(props, state){

console.log("getDerivedStateFromProps")

console.log(props, state)

return {}

}

componentDidMount(){

console.log("componentDidMount")

}

// ReactJS Component Updating Methods

shouldComponentUpdate(){

console.log("shouldComponentUpdate")

return true

}

componentDidUpdate(){

console.log("componentDidUpdate")

}

// ReactJS Component Unmounting Methods

componentWillUnmount(){

console.log("componentWillUnmount")

}

// custom methods

handleChange(event) {

this.setState({username: event.target.value});

}

render(){

console.log("render")

return(

<form>

<label>Name: </label>

<input name="username" onChange={this.handleChange}/>

<br/>

Your name is "{this.state.username}"

</form>

);

}

}

module.exports = LifeCycleMethods;

**REACT HOOKS**



## Using Hooks Requires react "next"

## So, use "next" in your package.json to get the latest version of it

*// package.json*

"react": "next",

"react-dom": "next"

**OR**

npm install react@next react-dom@next

## We are covering 4 hooks of react

1. useState
2. useEffect
3. useContext
4. useRef

# useState

An example for increment/decrement value with reset button.

#### Example code:

**import** React, { useState} **from** 'react';

**function** **Example**() {

**const** [count, setCount] = useState(0);

**return** (

<div>

Count: {count}

<button onClick={() => setCount(0)}>Reset</button>

<button onClick={() => setCount(count + 1)}>+</button>

<button onClick={() => setCount(count - 1)}>-</button>

</div>

);

}

**export** **default** Example;

#### Description:

* Start by importing useState from react. Make a functional component and use useState() therein. useState() is used to declare a state variable and can be initialized with any type of value.

const [count, setCount] = useState(0)*;*

* The first value, count in this case, is the current state (like this.state) and  
  The second value setCount is a function used to update the state (first) value.  
  The 0 in useState(0) function is Initial Value of that particular state.

# useEffect

An example for changing webpage title in realtime

#### Example code:

**function** **Example**() {

**const** [count, setCount] = useState(0);

useEffect(() => {

document.title = `You clicked ${count} times`;

});

**return** (

<div>

<p>You clicked {count} times</p>

<button onClick={() => setCount(count + 1)}>Click me</button>

</div>

);

}

#### Description:

* Just like we did in previous example by importing useState. Here same we will import useEffect from react. Make a functional component and use useEffect() therein. **useEffect()** is used to write function statements and as seen above we are changing title of webpage.
* **useEffect** is similar to **componentDidMount** and **componentDidUpdate** of React Lifecycles.
* Hence, whenever you click on the button, the title text will change with (number of) counts. Like.. You clicked (n) times.

# useContext

An example for useContext

**const** TestContext = React.createContext();

**function** **Display**() {

**const** value = useContext(TestContext);

**return** <div>{value}, I am learning react hooks.</div>;

}

**function** **App**() {

**return** (

<TestContext.Provider value={"Raman"}>

<Display />

</TestContext.Provider>

);

}

#### Description:

* In the code above, the context TestContext is created using React.createContext(). I have use the **TestContext.Provider** in our App component and set the value there to "Raman". This means any context-reading object in the tree can now read that value.
* To read this value in the Display() function we call useContext, passing the TestContext an argument.
* We then pass in the context object we got from React.createContext, and it automatically outputs the value. When the value of the provider updates, this Hook will trigger a rerender with the latest context value.

# useRef

Refs provide a way to access the React elements created in the render() method

##### An example for **useRef**

* The useRef() function returns a ref object.

**function** **App**() {

**let** [name, setName] = useState("Nate");

**let** nameRef = useRef();

**const** submitButton = () => {

setName(nameRef.current.value);

};

**return** (

<div className="App">

<p>{name}</p>

<div>

<input ref={nameRef} type="text" />

<button type="button" onClick={submitButton}>

Submit

</button>

</div>

</div>

);

}

#### Description:

* In the example above, we're using the useRef() hook in conjunction with the useState() to render the value of the input tag into a p tag.
* The ref is instantiated into the nameRef variable. The nameRef variable can then be used in the input field by being set as the ref. Essentially, this means the content of the input field will now be accessible through ref.
* The submit button in the code has an onClick event handler called submitButton. The submitButton function calls setName (created via useState).
* As we've done with useState hooks before, setName will be used to set the state name. To extract the name from the input tag, we read the value nameRef.current.value.
* Another thing to note concerning useRef is the fact that it can be used for more than the ref attribute.

## ComponentDidMount (and render props)

Data fetching in React! Do you remember the old days of componentDidMount? You would slap fetch(url) in componentDidMount and call it a day. Here’s how to fetch an array of data from an API for rendering out a nice list:

import React, { Component } from "react";

export default class DataLoader extends Component {

state = { data: [] };

componentDidMount() {

fetch("http://localhost:3001/todos/")

.then(response => response.json())

.then(data =>

**this**.setState(() => {

**return** { data };

})

);

}

render() {

**return** (

<div>

<ul>

{**this**.state.data.map(el => (

<li key={el.id}>{el.title}</li>

))}

</ul>

</div>

);

}}