React JS Introduction

Features of ReactJS

1. Fast
2. Composable
3. Easily nest components, support for props
4. Pluggable
5. Easily integrate with other technologies
6. Isomorphic friendly
7. Simple
8. API is small
9. Battle proven

React is for View and Controller in MVC pattern
To configure React you need to install ract and react-dom packages.

Npm install –save react react-dom

ReactDOM

If you load React from a <script> tag, these top-level APIs are available on the ReactDOM global. If you use ES6 with npm, you can write **import ReactDOM from 'react-dom'**. If you use ES5 with npm, you can write **var ReactDOM = require('react-dom').**

ReactDOM.render(element, container, [callback])

ReactDOM.render() controls the contents of the container node you pass in. Any existing DOM elements inside are replaced when first called. Later calls use React's DOM diffing algorithm for efficient updates.

React.createClass VS React.Component

Two ways to do the same thing. Almost. React traditionally provided the React.createClass method to create component classes, and released a small syntax sugar update to allow for better use with ES6 modules by extends React.Component, which extends the Component class instead of calling createClass.

React.createClass demo

```
import React from 'react';
const Contacts = React.createClass({
        render() {
                return (
                        <div></div>
```

```
                );
        }
});
export default Contacts;
```

React.Component demo

```
import React from 'react';

class Contacts extends React.Component {
        constructor(props) {
                super(props);
        }
        render() {
                return (
                        <div></div>
                );
        }
}

export default Contacts;
```

propTypes and getDefaultProps

There are important changes in how we use and declare default props, their types and setting initial states.

React.createClass

In the React.createClass version, the propTypes property is an Object in which we can declare the type for each prop. The getDefaultProps property is a function that returns an Object to create initial props.

```
import React from 'react';

const Contacts = React.createClass({
        propTypes: {
        },
        getDefaultProps() {
                return {
                };
        },
        render() {
                return (
                        <div></div>
                );
        }
```

```
});
export default Contacts;
```

React.Component

This uses propTypes as a property on the actual class instead of a property as part of the createClass definition Object. The getDefaultProps has now changed to just an Object property on the class called defaultProps, as it's no longer a "get" function, it's just an Object.

```
import React from 'react';

class Contacts extends React.Component {
        constructor(props) {
                super(props);
         }
        render() {
                return (
                        <div></div>
                );
        }
}
Contacts.propTypes = {

};
Contacts.defaultProps = {

};

export default Contacts;
```

State differences

React.createClass

We have a getInitialState function, which simply returns an Object of initial states.

```
import React from 'react';

const Contacts = React.createClass({
        getInitialState () {
                return {

                };
        },
         render() {
                 return (
```

```
                <div></div>
            );
        }
});

export default Contacts;
```

React.Component

The getInitialState function is deceased, and now we declare all state as a simple initialisation property in the constructor

```
import React from 'react';

class Contacts extends React.Component {
        constructor(props) {
            super(props);
            this.state = {
            };
        }
        render() {
            return (
                <div></div>
            );
        }
}

export default Contacts;
```

“this” differences

React.createClass

Note the onClick declaration with this.handleClick bound. When this method gets called React will apply the right execution context to handleClick.

```
import React from 'react';

const Contacts = React.createClass({
        handleClick() {
            console.log(this); // React Component instance
        },
        render() {
            return (
                <div onClick={this.handleClick}></div>
```

```
                );
        }
});

export default Contacts;
```

React.Component

With ES6 classes this is slightly different, properties of the class do not automatically bind to the React class instance.

```
import React from 'react';

class Contacts extends React.Component {
        constructor(props) {
                super(props);
        }
        handleClick() {
                console.log(this); // null
        }
        render() {
                return (
                        <div onClick={this.handleClick.bind(this)}></div>
                );
        }
}

export default Contacts;
```

Another way to do the same is

```
import React from 'react';

class Contacts extends React.Component {
        constructor(props) {
                super(props);
                this.handleClick = this.handleClick.bind(this);
        }
        handleClick() {
                console.log(this); // React Component instance
        }
        render() {
                return (
                        <div onClick={this.handleClick}></div>
                );
```

```
        }
}

export default Contacts;
```

React using ES6 classes

You can create React applications by using ES6 syntax also.

1) First you need to install react libraries for your application.

npm install --save react react-dom

2) Add a new folder ‘components’ in the src folder which is used to keep all the components you create in your application.

3) Create a new component class ‘home.jsx’ inside the components folder. Add the following code inside the home.jsx.

```
import React from 'react';
export class Home extends React.Component{
  render(){
    return(
      <div>
        <p>It is Home Component</p>
      </div>
    );
  }
}
```

4) Also, You can create one more component named header.jsx to contain the navigation menus.

```
import React from 'react';
export class Header extends React.Component {
  render() {
    return (
      <nav className="navbar navbar-inverse">
        <div className="container-fluid">
          <div className="navbar-header">
            <button type="button" className="navbar-toggle collapsed" data-
toggle="collapse" data-target="#bs-example-navbar-collapse-1" aria-expanded="false">
              <span className="sr-only">Toggle navigation</span>
              <span className="icon-bar"></span>
              <span className="icon-bar"></span>
              <span className="icon-bar"></span>
            </button>
```

```
            <a className="navbar-brand" href="#">React Demo</a>
          </div>
          <div className="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
            <ul className="nav navbar-nav">
              <li className="active"><a href="#">Link <span className="sr-
only">(current)</span></a></li>
              <li><a href="#">Link</a></li>
              <li className="dropdown">
                <a href="#" className="dropdown-toggle" data-toggle="dropdown"
role="button" aria-haspopup="true" aria-expanded="false">Dropdown <span
class="caret"></span></a>
                <ul className="dropdown-menu">
                  <li><a href="#">Action</a></li>
                  <li><a href="#">Another action</a></li>
                  <li><a href="#">Something else here</a></li>
                  <li role="separator" className="divider"></li>
                  <li><a href="#">Separated link</a></li>
                  <li role="separator" className="divider"></li>
                  <li><a href="#">One more separated link</a></li>
                </ul>
              </li>
            </ul>

            <ul className="nav navbar-nav navbar-right">
              <li><a href="#">Link</a></li>
              <li className="dropdown">
                <a href="#" class="dropdown-toggle" data-toggle="dropdown"
role="button" aria-haspopup="true" aria-expanded="false">Dropdown <span
class="caret"></span></a>
                <ul className="dropdown-menu">
                  <li><a href="#">Action</a></li>
                  <li><a href="#">Another action</a></li>
                  <li><a href="#">Something else here</a></li>
                  <li role="separator" className="divider"></li>
                  <li><a href="#">Separated link</a></li>
                </ul>
              </li>
            </ul>
          </div>
        </nav>
      )
    }
}
```

5) Update the main.js file with the following code.

```
import React from "react";
import { render } from "react-dom";
import { Header } from './components/Header';
import { Home } from './components/Home';

class AppComponent extends React.Component {
  render() {
    return (
      <div className="container-fluid">
        <Header />
        <Home />
      </div>
    )
  }
};
render(<AppComponent />, document.getElementById("app"));
```

Multiple components in React

You have already created a root component called App in react application. You can add only one root component in a react application. But the root component can container multiple child components. Let's see how to do it…

1) You need to create a new folder 'components' in the 'app' folder. Inside the components, you can create two components 'Header' and 'Home'.
2) Create Header.js inside the components folder and add the following code in the file.

```
import React from 'react';

export class Header extends React.Component{
    render(){
        return (
            <nav className="navbar navbar-default">
                <div className="container">
                    <div className="navbar-header">
                        <ul className="nav navbar-nav">
                            <li><a href="#">Home</a></li>
                        </ul>
                    </div>
                </div>
            </nav>
        )
    }
}
```

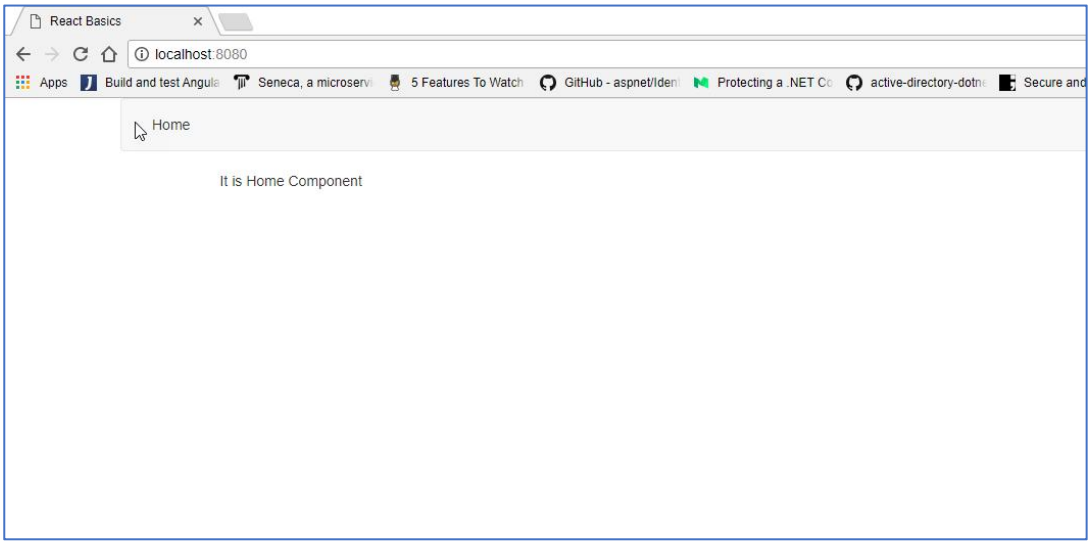
3) Create another component Home in the Home.js file and add the following code.

```
import React from 'react';

export class Home extends React.Component{
    render(){
        return(
            <div>
                <p>It is Home Component</p>
            </div>
        );
    }
}
```

4) Update the index.js file with the following code.

```
import React from "react";
import { render } from "react-dom";

import {Header } from './components/Header';
import {Home } from './components/Home';

class AppComponent extends React.Component {
    render() {
        return (
            <div className="container">
                <div className="row">
                    <div className="col-xs-12">
                        <Header/>
                    </div>
                </div>

                <div className="row">
                    <div className="col-xs-10 col-xs-offset-1">
                        <Home/>
                    </div>
                </div>
            </div>
        )
    }
}

render(<AppComponent />, document.getElementById("app"));
```

5) Run the application using the 'npm start' command. Navigate to http://localhost:8080.



Showing Dynamic Data

You also can show dynamic data on your React application. To show some dynamic value on the page you can use {} symbol, which is an expression that displays the value of an expression or variable.

Eg:

```
export class Home extends React.Component{

  constructor(){
    super();
    this.name="Sonu Sathyadas";
  }

  render(){
    return(
      <div>
        <p>It is Home Component</p>
        <p>{ 2 +2 }</p> {/*javascript expression*/}
        <p>{this.name}</p> {/* variable*/}
      </div>
    );
  }
}
```

Whatever you write inside the curly bracket ({ }) should be one line. Multiline statements are not allowed inside it.

Props in React JS

Props helps us to pass data to a component in React. You can use html like attributes to pass information to the component. You can read those property information/values using the **props** keyword in the component object.

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”.

```
function Welcome(props) {
        return <h1>Hello, {props.name}</h1>;
}
const element = <Welcome name="Sara" />;
ReactDOM.render(element,document.getElementById('root'));
```

Props are Read-Only

Whether you declare a component as a function or a class, it must never modify its own props. Consider this sum function:

```
function sum(a, b) {
  return a + b;
}
```

Such functions are called “**pure**” because they do not attempt to change their inputs, and always return the same result for the same inputs.

In contrast, this function is impure because it changes its own input:

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

React is pretty flexible but it has a single strict rule:

All React components must act like pure functions with respect to their props.

Default Props

You can also set default property values directly on the component constructor instead of adding it to the reactDom.render() element.

PropTypes

You can define the property types by using the PropTypes. React offers many features to assist developers, including a great suite of validators for checking the props set for a component are as expected. You can validate whether a prop:

- is required

- contains a primitive type
- contains something renderable (a node)
- is a React Element
- contains one of several defined types
- is an array containing only items of a specified type
- contains an instanceof a class
- contains an object that has a specific shape

From React 15.5 onwards PropTypes is coming as a separate node package ‘prop-types’. You need to install it.

npm install --save prop-types

You can define the **defaultProps** and **propTypes** as mentioned below.

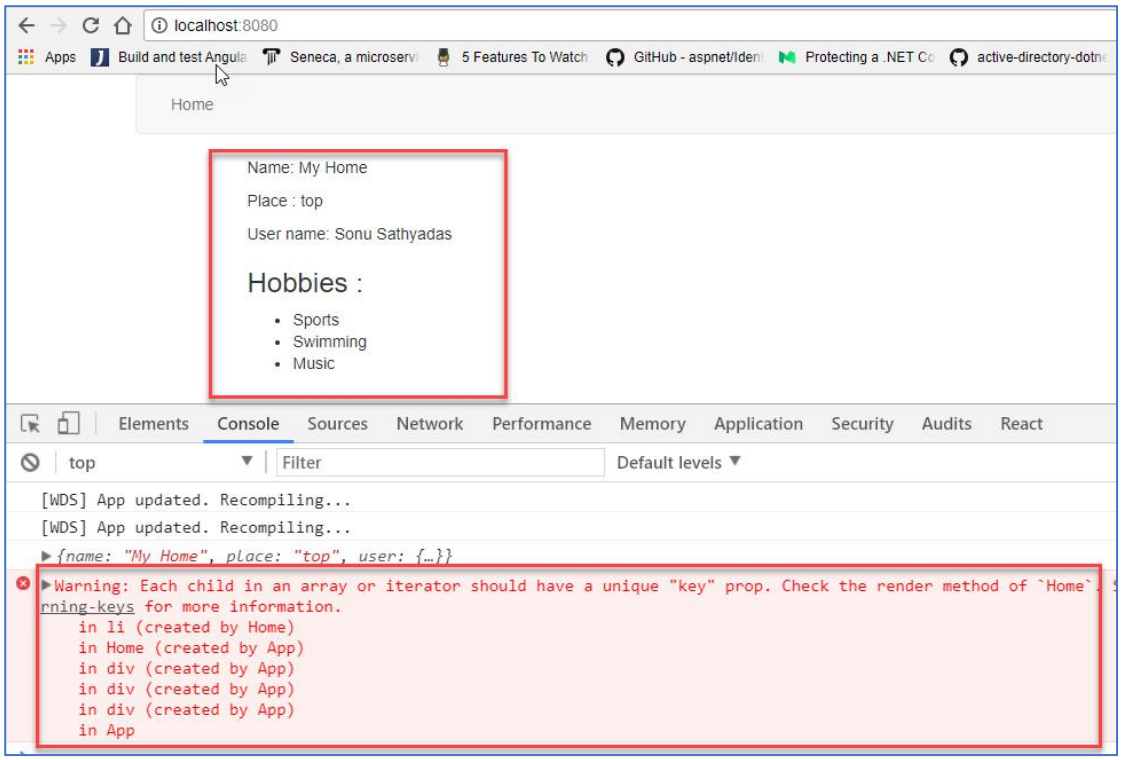
```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
export default class Message extends Component {
  render() {
    return (
      <div>
        <p className="text-success">
Welcome {this.props.username}
</p>
        <p>Created by {this.props.createdBy}</p>
      </div>
    );
  }
}
Message.defaultProps = {
  username: 'Michel Holding',
  createdBy: 'Sonu'
}
Message.propTypes={
  username:PropTypes.string,
  createdBy:PropTypes.string
}
```

```
class App extends React.Component {
    render() {
        let user={
            name:"Sonu Sathyadas",
            hobbies:["Sports", "Swimming", "Music"],
        };

        return (
            <div className="container">
                <div className="row">
                    <div className="col-xs-12">
                        <Header/>
                    </div>
                </div>

                <div className="row">
                    <div className="col-xs-10 col-xs-offset-1">
                        <Home name={"My Home"} place={"top"} user={user}/>
                    </div>
                </div>
            </div>
        )
    }
}
```

You can read this values in the Home component. To read the values use the props property of the object.

```
export class Home extends React.Component{
    render(){
        console.log(this.props);
        return(
            <div>
                <p>Name: {this.props.name}</p>
                <p>Place : {this.props.place}</p>
                <p>User name: {this.props.user.name} </p>
                <div> <h3>Hobbies :</h3>
                    <ul>
                        {this.props.user.hobbies.map((hobby)=><li>{hobby}</li>)}
                    </ul>
                </div>
            </div>
        );
    }
}
```

It provides the output in the following format. But it also gives an error in the console, because every element in React must have a unique key to identify it.



To avoid the error we must provide unique key for each <li> element. You can pass a second argument to the arrow method which represent the index of the element that can be used as the key for the list element.

```
export class Home extends React.Component{
    render(){
        console.log(this.props);
        return(
            <div>
                <p>Name: {this.props.name}</p>
                <p>Place : {this.props.place}</p>
                <p>User name: {this.props.user.name} </p>
                <div> <h3>Hobbies :</h3>
                    <ul>
                        {this.props.user.hobbies.map((hobby, i)=><li key={i}>{hobby}</li>)}
                    </ul>
                </div>
            </div>
        );
    }
}
```

It is also a good practice to specify the type of **props** used inside the component. You can use the module **React.PropTypes** to define the types of **props** used inside your React component.

```
export class Home extends React.Component{
    render(){
        console.log(this.props);
        return(
            <div>
                <p>Name: {this.props.name}</p>
                <p>age : {this.props.age}</p>
                <p>User name: {this.props.user.name} </p>
                <div> <h3>Hobbies :</h3>
                    <ul>
                        {this.props.user.hobbies.map((hobby, i)=><li key={i}>{hobby}</li>)}
                    </ul>
                </div>
            </div>
        );
    }
}

Home.propTypes={
    name:React.PropTypes.string,
    age:React.PropTypes.number,
    user:React.PropTypes.object
};
```

It is also possible that you can include child elements to your react component tags. The child elements can also be accessed using **props.children** in the component. Assume that your Home component contains some child elements.

```
class App extends React.Component {
    render() {
        let user={
            name:"Sonu Sathyadas",
            hobbies:["Sports", "Swimming", "Music"],
        };

        return (
            <div className="container">
                <div className="row">
                    <div className="col-xs-12">
                        <Header/>
                    </div>
                </div>

                <div className="row">
                    <div className="col-xs-10 col-xs-offset-1">
                        <Home name={"My Home"} age={30} user={user}>
                            <div>
                                <h2>Welcome to React</h2>
                                <p>This is a simple React application that describes the use of
                                props in React components</p>
                            </div>
                        </Home>

                    </div>
                </div>
            </div>
        )
    }
}
```

You can add the child element to the Home component. Note that you can use only one child

inside the component tag. For example, you have a paragraph and heading to display inside the component. You can put both the elements into a div tag and place inside the home element. You can also set the PropTypes for your children.

```
export class Home extends React.Component{
    render(){
        console.log(this.props);
        return(
            <div>
                <p>Name: {this.props.name}</p>
                <p>Age : {this.props.age}</p>
                <p>User name: {this.props.user.name} </p>
                <div> <h3>Hobbies :</h3>
                    <ul>
                        {this.props.user.hobbies.map((hobby, i)=><li key={i}>{hobby}</li>)}
                    </ul>
                </div>
                <hr/>
                {this.props.children}
            </div>
        );
    }
}

Home.propTypes={
    name:React.PropTypes.string,
    age:React.PropTypes.number,
    user:React.PropTypes.object,
    children:React.PropTypes.element.isRequired
};
```

Custom PropTypes

Sometimes you might need to go beyond the built-in validation functions and because we're working with functions, it's actually remarkably easy to create our own.

```
function tweetLength(props, propName, componentName) {
componentName = comopnentName || 'ANONYMOUS';
if (props[propName]) {
        let value = props[propName];
        if (typeof value === 'string') {
        return value.length <= 140 ? null : new Error(propName + ' in ' + componentName +
" is longer than 140 characters");
        }
        }
//assume all ok
        return null;
}
```

How to use:

```
MyComponent.propTypes = {
                title: tweetLength,
```

content: React.PropTypes.isRequired

}

PropTypes complete List

React.PropTypes has moved into a different package since React v15.5. Please use the prop-types library instead.

More: https://reactjs.org/docs/typechecking-with-proptypes.html

```
import PropTypes from 'prop-types';

MyComponent.propTypes = {
 // You can declare that a prop is a specific JS primitive. By default, these
 // are all optional.
 optionalArray: PropTypes.array,
 optionalBool: PropTypes.bool,
 optionalFunc: PropTypes.func,
 optionalNumber: PropTypes.number,
 optionalObject: PropTypes.object,
 optionalString: PropTypes.string,
 optionalSymbol: PropTypes.symbol,
 // Anything that can be rendered: numbers, strings, elements or an array
 // (or fragment) containing these types.
 optionalNode: PropTypes.node,

 // A React element.
 optionalElement: PropTypes.element,
 // You can also declare that a prop is an instance of a class. This uses
 // JS's instanceof operator.
 optionalMessage: PropTypes.instanceOf(Message),
 // You can ensure that your prop is limited to specific values by treating
 // it as an enum.
```

```
optionalEnum: PropTypes.oneOf(['News', 'Photos']),
// An object that could be one of many types
optionalUnion: PropTypes.oneOfType([
  PropTypes.string,
  PropTypes.number,
  PropTypes.instanceOf(Message)
]),
// An array of a certain type
optionalArrayOf: PropTypes.arrayOf(PropTypes.number),
// An object with property values of a certain type
optionalObjectOf: PropTypes.objectOf(PropTypes.number),
// An object taking on a particular shape
optionalObjectWithShape: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
}),
// You can chain any of the above with `isRequired` to make sure a warning
// is shown if the prop isn't provided.
requiredFunc: PropTypes.func.isRequired,
// A value of any data type
requiredAny: PropTypes.any.isRequired,

// You can also specify a custom validator. It should return an Error
// object if the validation fails. Don't `console.warn` or throw, as this
// won't work inside `oneOfType`.
customProp: function(props, propName, componentName) {
  if (!/matchme/.test(props[propName])) {
    return new Error(
```

```
      'Invalid prop `' + propName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    );
  }
 },

 // You can also supply a custom validator to `arrayOf` and `objectOf`.
 // It should return an Error object if the validation fails. The validator
 // will be called for each key in the array or object. The first two
 // arguments of the validator are the array or object itself, and the
 // current item's key.
 customArrayProp: PropTypes.arrayOf(function(propValue, key, componentName, location,
propFullName) {
  if (!/matchme/.test(propValue[key])) {
   return new Error(
     'Invalid prop `' + propFullName + '` supplied to' +
     ' `' + componentName + '`. Validation failed.'
    );
  }
 })
};
```

Events in React JS

Add a new button to the Home component. When you click the button, you need to update the value of the age variable declared inside the component. For that you can create a constructor and pass the props as an argument to it. Store the value of age to the age variable. Also, you can create an event handler method that can be called on the button click.

```
export class Home extends React.Component{

    constructor(props){
        super();
        this.age=props.age;
    }

    onUpdate(){
        console.log("Before:" + this.age);
        this.age+=5;
        console.log("After:" + this.age);
    }

    render(){
        return(
            <div>
                <p>Name: {this.props.name}</p>
                <p>Age : {this.age}</p>
                <hr/>
                <button className="btn btn-primary" onClick={()=>this.onUpdate()}>Update</button>
            </div>
        );
    }
}
```

It executes the onUpdate method whenever you click the button. But it will not update the view with the updated age value. Because, the state of the React View is not updated.

Component State in React

To update the view of the React component whenever the event executes you can use the state property of the React component. Instead of declaring age as a member of the component, you can declare it as a member of the **state** property.

When you click the button you can update the value of the state by calling the **setState()** method. This method can accept a JSON object that contains the value of only updated members. If some members are not updated it is not necessary to specify inside the **setState()** method.

```
export class Home extends React.Component{

    constructor(props){
        super();
        this.state={
            age:props.age,
            name:props.name
        };
    }

    onUpdate(){
        this.setState({
            age:this.state.age + 5 //no need to update the name because it is not changed
        });
    }

    render(){
        return(
            <div>
                <p>Name: {this.state.name}</p>
                <p>Age : {this.state.age}</p>
                <hr/>
                <button className="btn btn-primary" onClick={()=>this.onUpdate()}>Update</button>
            </div>
        );
    }
}
```

Stateless component

It is also possible to create stateless component to makes your application better, because the state change in React is unpredictable and it makes become more complex to handle such application. You can use Redux to handle all state related operations.

A stateless component does not have a state property but have only props. To create a stateless component, you can create a **const** that is a function with a props argument. The function returns the rendered view.

```
export const Header = (props) => {
    return (
        <nav className="navbar navbar-default">
            <div className="container">
                <div className="navbar-header">
                    <ul className="nav navbar-nav">
                        <li><a href="#">Home</a></li>
                    </ul>
                </div>
            </div>
        </nav>
    )
};
```

It is also possible to pass dynamic values to the Stateless component but it will bind the value to the view only once. Hence, there is no state associated with it, it is not possible to update the view with the changes in the property variable.

```
export const Header = (props) => {
    return (
        <nav className="navbar navbar-default">
            <div className="container">
                <div className="navbar-header">
                    <ul className="nav navbar-nav">
                        <li><a href={props.homeLink}>Home</a></li>
                    </ul>
                </div>
            </div>
        </nav>
    )
};
```

You can pass the value from the App Component.

```
class App extends React.Component {
    render() {
        return (
            <div className="container">
                <div className="row">
                    <div className="col-xs-12">
                        <Header homeLink="home.html"/>
                    </div>
                </div>

                <div className="row">
                    <div className="col-xs-10 col-xs-offset-1">
                        <Home name={"My Home"} age={30}/>

                    </div>
                </div>
            </div>
        )
    }
}
```

Communication between Child and Parent component.

In React components can communicate each other. It depends of the relationship between the

components, and it depends on what we prefer.

There are 3 possible relationships:

- parent to child (one-way binding, default in React)
- child to parent (Two-way binding, need to pass methods through props)
- not directly related

You can pass the data from parent component to the child component by using props as we see in the above examples. You can pass any values to the child components by using attributes and that can be read using props in the child component.

But for passing data from child component to the parent component, you can use the props again. This time you need to define a method inside the parent component and pass it to the child component as an attribute. Child component can read it using props and execute with in it. If any value need to be passed you can call the method with arguments.

```
class App extends React.Component {

    doSomething(){
        alert("Hello, I am parent func without args");
    }

    doSomethingMore(data){
        alert("Hello, I am also parent method with args, args is " + data);
    }

    render() {
        return (
            <div className="container">
                <div className="row">
                    <div className="col-xs-12">
                        <Header homeLink="home.html"/>
                    </div>
                </div>

                <div className="row">
                    <div className="col-xs-10 col-xs-offset-1">
                        <Home name={"My Home"} age={30}
                            method1={this.doSomething}
                            method2={this.doSomethingMore}
                        />
                    </div>
                </div>
```

Now you can update the Home component to execute the parent methods by invoking them using props.

```
export class Home extends React.Component{

    constructor(props){ ···
    }

    onUpdate(){ ···
    }

    callWithArgs(){
        this.props.method2("Sonu Sathyadas");
    }

    render(){
        return(
            <div>
                <p>Name: {this.state.name}</p>
                <p>Age : {this.state.age}</p>
                <hr/>
                <button onClick={()=>this.onUpdate()} className="btn btn-primary">Update</button>
                <hr/>
                <button onClick={()=>this.props.method1()} className="btn btn-primary">Call Parent</button>
                <hr/>
                <button onClick={()=>this.props.method2('Synergetics')} className="btn btn-primary">Call Parent</button>
                <hr/>
                <button onClick={()=>this.callWithArgs()} className="btn btn-primary">Call Parent</button>
            </div>
        );
    }
```

Calling using Child method

You can also define the types of the prop types in Home component

```
Home.propTypes={
    name:React.PropTypes.string,
    age:React.PropTypes.number,
    method1:React.PropTypes.func,
    method2:React.PropTypes.func
};
```

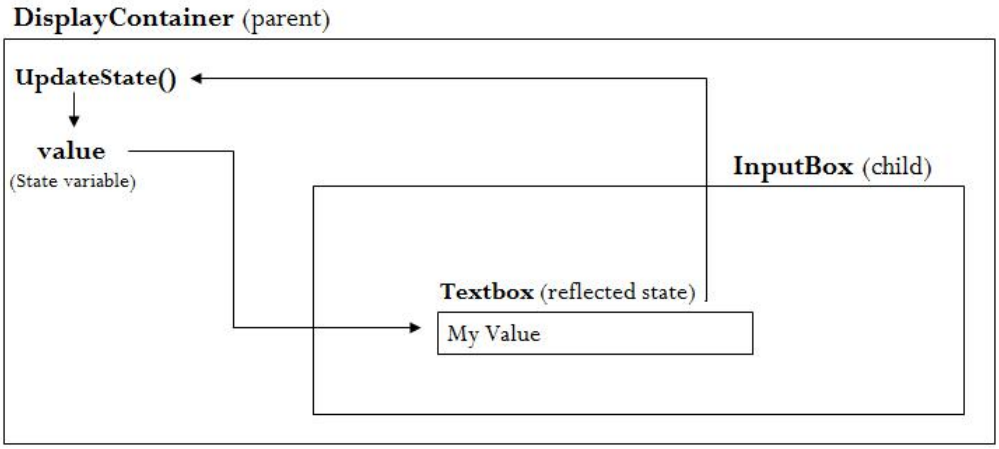
Binding

One-Way binding

React apps are organized as a series of nested components. These components are functional in nature: that is, they receive information through arguments (represented in the props attribute) and pass information via their return values (the return value of the render function). This is called unidirectional data flow.

Two-way binding (Child to Parent communication)

Facebook react was not really made for two way data binding. But by adding a little complexity we can make two way data binding easily. Normally in facebook react data flows only in one direction; that is, parent to child. So when a data is modified in parent, that reflects to the child but to reflect the changed data of child to the parent, you have to notify the parent about the change of value of child and also have to take appropriate action.



The state variable 'value' of the component 'DisplaContainer' in initialized with 'My Value', and is passed through prop to the child. Once you run the program you will see 'My Value' is reflecting to the textbox of 'InputBox' component. This is the default binding facebook react provides; data flowing automatically from parent to child.

Now you can follow any of the following two processes to send the modified data back to parent and once you done that the modified data will again reflect every child from the parent automatically. So what you need to do is send the modified data back to the parent.

Two-way binding using props

```
import React,{Component} from 'react';
import {InputBox} from './InputBox';

export class DisplayContainer extends Component{
    constructor(){
        super();
        this.state={
            data:"default value"
        };
        this.showData=this.showData.bind(this);
    }

    showData(updatedValue){
        this.setState({data:updatedValue});
    }
    render(){
        return(
            <div>
                <p>Data : {this.state.data}</p>
```

```
                <InputBox value={this.state.data} update={this.showData} />
            </div>
        );
    }
}
```

Communication between components are not directly related

When components are not related or are related but too far away in the hierarchy, we can use an external event system to notify anyone that wants to listen.

It is the basis of any event system:

- we can subscribe/listen to some events and be notify when they happen
- anyone can send/trigger/publish/dispatch an event of this kind to notify the ones who are listening

Component Life Cycle

When a component is initialized and rendered in React it goes through different life cycle methods or life cycle hooks. The following are the list of methods participated in the component creation in React.

1) **componentWillMount()**
   This method executes immediately before the initial rendering happens for the component. It is optional to implement the method. If the method exists in the component class, it executes immediately before the initial rendering of the component. (executes before the render method).
2) **componentDidMount()**
   It is the next method in the life cycle. It executes immediately after the initial rendering of the component.
3) **ComponentWillReceiveProps()**
   This method is executed whenever a component is about to receive a props.
4) **shouldComponentUpdate()**
   It is executed before the re-rendering of the component. Whenever the props or state changes for a component, it decides whether to re-render the view to update the content in the view. This method decides whether any change happened in the Virtual DOM and need to update the actual DOM or not. You can write your custom code before re-rendering the view. This method can return false to prevent re-rendering of the component.
5) **componentWillUpdate()**
   This method is executed before the component is re-rendered. The

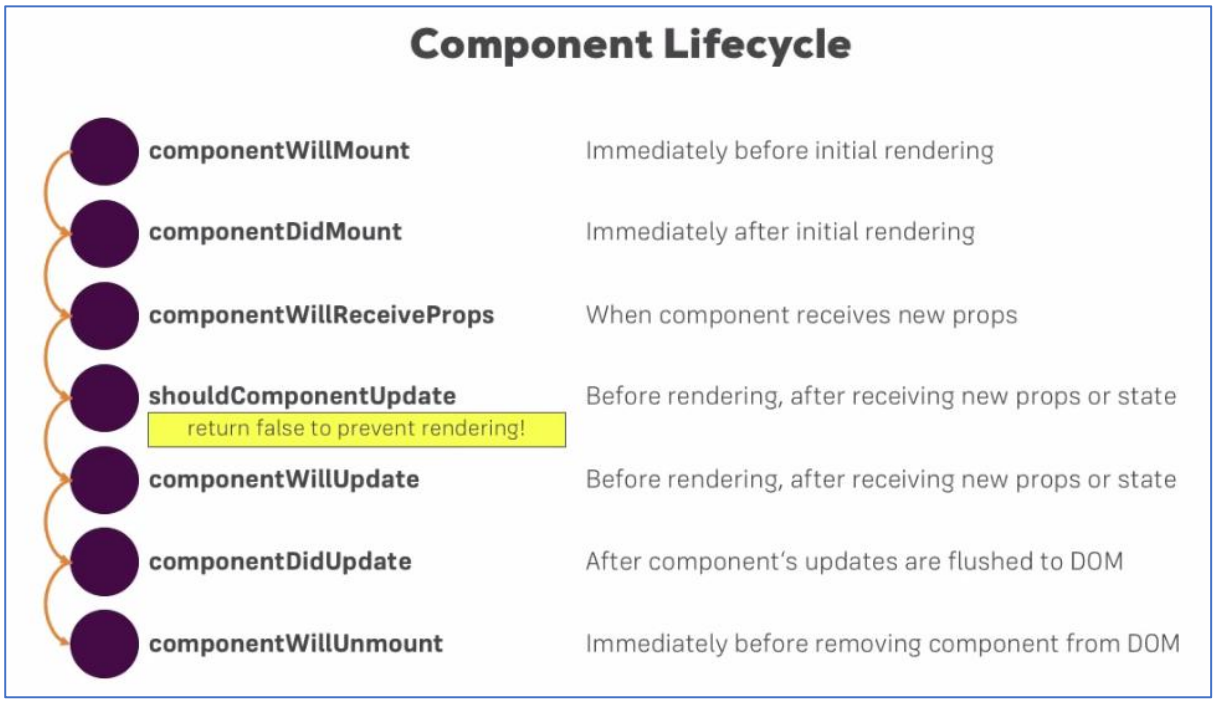
shouldComponentUpdate() decides whether to render the DOM and if decides to render the DOM then the componentWillUpdate() method gets executed.

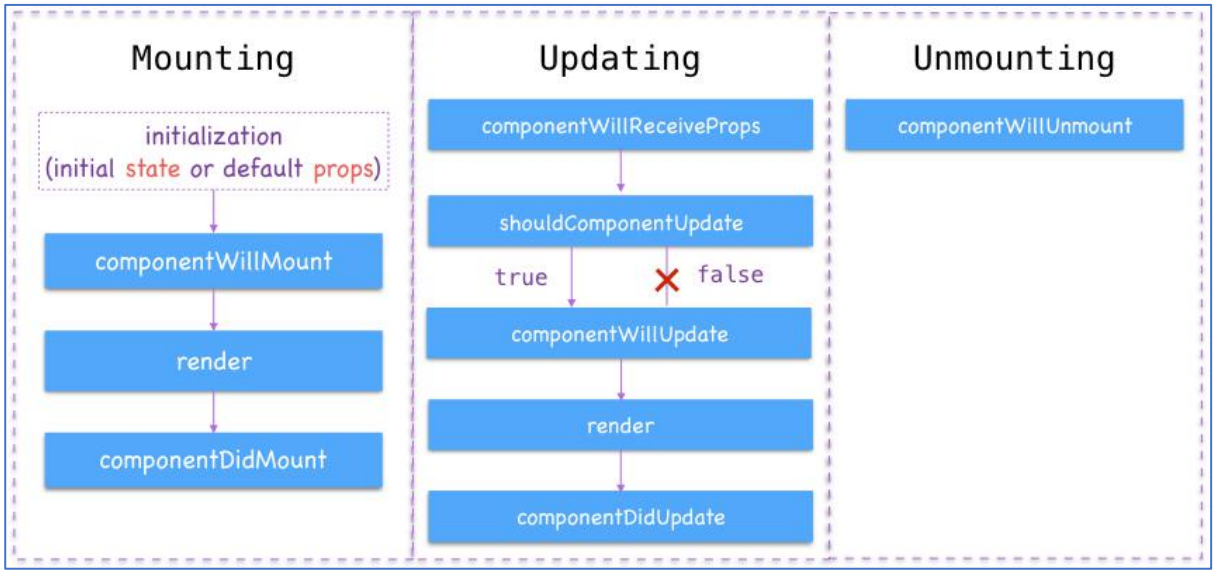
6) **componentDidUpdate()**

This method executes after the component is rendered and flushes to the DOM.

7) **componentWillUnmount()**

It executes immediately before the component is removed from the DOM.





Conditional Rendering

In React, you can create distinct components that encapsulate behavior you need. Then, you can render only some of them, depending on the state of your application.

Conditional rendering in React works the same way conditions work in JavaScript. Use JavaScript operators like if or the conditional operator to create elements representing the current state, and let React update the UI to match them.

Inline If with Logical && Operator

You may embed any expressions in JSX by wrapping them in curly braces. This includes the JavaScript logical && operator. It can be handy for conditionally including an element.

```
import React, { Component } from 'react';

export class Home extends Component {

    constructor() {
        super();
        this.state={
            loginState:false
        };
        this.loginClick=this.loginClick.bind(this);
    }

    loginClick(){
        this.setState({
            loginState:!this.state.loginState,
            messageCount:5
        });
    }

    render() {
        let buttonText=this.state.loginState?"Logout":"Login";
        let element=null;
        //Using If condition, rendering the content
        if(this.state.loginState){
            element=<p>Welcome user</p>;
        }else{
            element=<p>Welecome guest</p>;
        }
```

```
        return (
            <div className="container-fluid">
                <div className="row">
                    <div className="col-md-4 col-md-offset-4">
                        {element}
                        <p>
                            {/* Displaying usig conditional
                                operator of React (&&) */}
                            {this.state.loginState &&
                                <span>
                                      You have {this.state.messageCount}
                                messages
                                </span>
                            }
                        </p>
                        <button className="btn btn-primary"
                                onClick={this.loginClick}>
                            {buttonText}
                        </button>
                    </div>
                </div>
            </div>
        );
    }
}
```

Inline If-Else with Conditional Operator

Another method for conditionally rendering elements inline is to use the JavaScript conditional operator condition ? true : false.

```
render() {
        let buttonText=this.state.loginState?"Logout":"Login";
        let element=null;
        //Using If condition, rendering the content
        if(this.state.loginState){
            element=<p>Welcome user</p>;
        }else{
            element=<p>Welecome guest</p>;
        }
        return (
            <div className="container-fluid">
                <div className="row">
```

```
                    <div className="col-md-4 col-md-offset-4">
                        {element}
                        <p>
                            {/* Displaying usig conditional
                                 operator of React */}
                            {
                                this.state.loginState ?
                                (<span>You have {this.state.messageCount}
messages</span>):
                                (<span>Click the login button to
login</span>)
                            }
                        </p>
                        <button className="btn btn-primary"
onClick={this.loginClick}>
                            {buttonText}
                        </button>
                    </div>
                </div>
            </div>
        );
    }
```

Preventing Component from Rendering

In rare cases you might want a component to hide itself even though it was rendered by another component. To do this return null instead of its render output.

Parent Component

```
import React, { Component } from 'react';
import { UserInfo } from './UserInfo';

export class Home extends Component {

    constructor() {
        super();
        this.state={
            loginState:false
        };
        this.loginClick=this.loginClick.bind(this);
    }
```

```
    loginClick(){
        this.setState(prevState=>({
            loginState:!this.state.loginState
        }));
    }

    render() {
        return (
            <div className="container-fluid">
                <div className="row">
                    <div className="col-md-4 col-md-offset-4">
                        <UserInfo loggedIn={this.state.loginState} />
                        <button className="btn btn-primary"
onClick={this.loginClick}>
                            {this.state.loginState?"Logout":"Login"}
                        </button>
                    </div>
                </div>
            </div>
        );
    }
}
```

Child Class

```
import React, { Component } from 'react';

export class UserInfo extends Component {
    constructor(){
        super();
    }
    render() {
        if (!this.props.loggedIn)
            return null;

        return (
            <p>This will display only when logged in</p>
        );
    }
}
```

Forms

HTML form elements work a little bit differently from other DOM elements in React, because form elements naturally keep some internal state. For example, this form in plain HTML accepts a single name:

```
<form>
    <label>Name:    <input type="text" name="name" /></label>
    <input type="submit" value="Submit" />
</form>
```

This form has the default HTML form behavior of browsing to a new page when the user submits the form. If you want this behavior in React, it just works. But in most cases, it's convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered into the form. The standard way to achieve this is with a technique called "**controlled components**".

Controlled and uncontrolled Inputs

Uncontrolled inputs are like traditional HTML form inputs:

They remember what you typed. You can then get their value using a ref. For example, in onClick handler of a button:

```
class Form extends Component {
    handleSubmitClick = () => {
    const name = this._name.value;
    // do something with `name`
    }
    render() {
    return (
        <div>
        <input type="text" ref={input => this._name = input} />
        <button onClick={this.handleSubmitClick}>Sign up</button>
        </div>
    );
    }
```

}

In React, mutable state is typically kept in the state property of components, and only updated with setState(). We can combine the two by making the React state be the “single source of truth”. Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a “controlled component”.

A **controlled** input accepts its current value as a prop, as well as a callback to change that value. You could say it’s a more “React way” of approaching this. The value of this input has to live in the state somewhere. Typically, the component that renders the input (aka the form component) saves that in its state:

```
class Form extends Component {
 constructor() {
  super();
  this.state = {
   name: '',
  };
 }

 handleNameChange = (event) => {
  this.setState({ name: event.target.value });
 };

 render() {
  return (
   <div>
    <input
     type="text"
     value={this.state.name}
     onChange={this.handleNameChange}
```

```
      />
    </div>
  );
 }
}
```

This flow kind of 'pushes' the value changes to the form component, so the Form component always has the current value of the input, without needing to ask for it explicitly. This means your data (state) and UI (inputs) are always in sync. The state gives the value to the input, and the input asks the Form to change the current value. This also means that the form component can respond to input changes immediately; for example, by:

- in-place feedback, like validations
- disabling the button unless all fields have valid data
- enforcing a specific input format, like credit card numbers

A form element becomes "controlled" if you set its value via a prop.

Element	Value property	Change callback	New value in the callback
`<input type="text" />`	`value="string"`	`onChange`	`event.target.value`
`<input type="checkbox" />`	`checked={boolean}`	`onChange`	`event.target.checked`
`<input type="radio" />`	`checked={boolean}`	`onChange`	`event.target.checked`
`<textarea />`	`value="string"`	`onChange`	`event.target.value`
`<select />`	`value="option value"`	`onChange`	`event.target.value`

Scenario where controlled and controlled are better

feature	uncontrolled	controlled
one-time value retrieval (e.g. on submit)	✅	✅
validating on submit	✅	✅
instant field validation	❌	✅
conditionally disabling submit button	❌	✅
enforcing input format	❌	✅
several inputs for one piece of data	❌	✅
dynamic inputs	❌	✅

Refs and DOM

In the typical React dataflow, props are the only way that parent components interact with their children. To modify a child, you re-render it with new props. However, there are a few cases where you need to imperatively modify a child outside of the typical dataflow. The child to be modified could be an instance of a React component, or it could be a DOM element.

React supports a special attribute that you can attach to any component. The ref attribute takes a callback function, and the callback will be executed immediately after the component is mounted or unmounted. When the ref attribute is used on an HTML element, the ref callback receives the underlying DOM element as its argument.

The **ref** attribute makes it possible to reference a DOM node in order to access it and to interact with it. The ref attribute definition always follows the same pattern: ref={node => this.input = node}. When the component renders the first time, you want to bind the DOM node to the this object of the component. Then you have access to the DOM node.

When to use React's Ref attribute?

There are a few good use cases for refs:

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

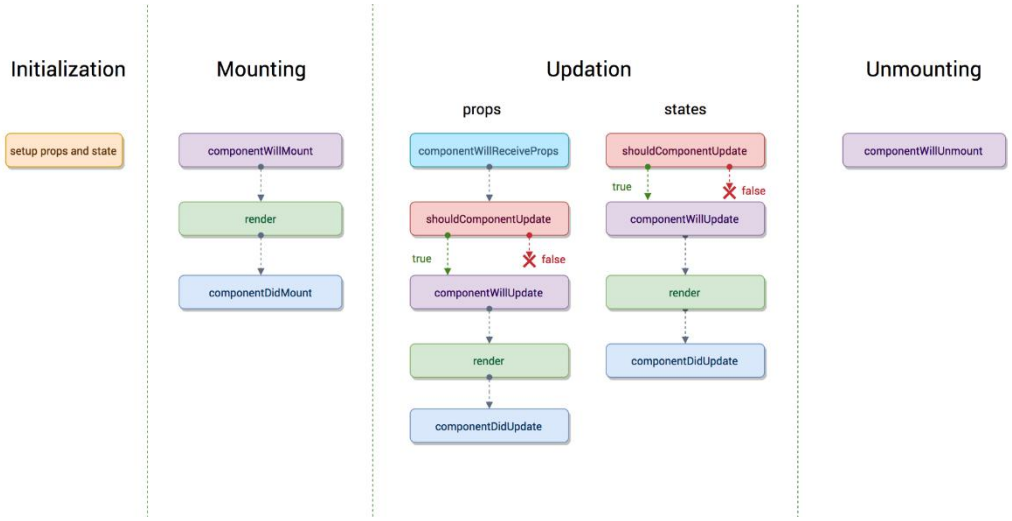
Avoid using refs for anything that can be done declaratively.

React will call the ref callback with the DOM element when the component mounts, and call it with null when it unmounts. **ref callbacks are invoked before componentDidMount or componentDidUpdate lifecycle hooks**.

Component Life Cycle

The lifecycle methods are various methods which are invoked at different phases of the lifecycle of a component. The React component, like anything else in the world, goes through the following phases

- Initialization
- Mounting
- Updation
- Unmounting



1) Initialization

In this phase the React component prepares for the upcoming tough journey, by setting up the initial states and default props. The component is setting up the initial state in the constructor, which can be changed later by using the **setState** method. The **defaultProps** is defined as a property of Component to define all the default value of props, which can be overridden with new prop values.

2) Mounting

After preparing with basic needs, state and props, our React Component is ready to mount in the browser DOM. This phase gives hook methods for before and after mounting of components. The methods which gets called in this phase are:

- **componentWillMount** is executed just before the React Component is about to mount on the DOM. Hence, after this method the component will mount. All the things that you want to do before a component mounts has

to be defined here. This method is executed once in a lifecycle of a component and before first render.
Usage: componentWillMount is used for initializing the states or props, there is a huge debate going on to merge it with the constructor.

- **render** mounts the component onto the browser. This is a pure method, which means it gives the same output every time the same input is provided.
- **componentDidMount** this is the hook method which is executed after the component did mount on the dom. This method is executed once in a lifecycle of a component and after the first render. As, in this method, we can access the DOM, we should initialize JS libraries like D3 or Jquery which needs to access the DOM.

3) Updation

This phase starts when the react component has taken birth on the browser and grows by receiving new updates. The component can be updated by two ways, sending new props or updating the state. Let's see the list of hook methods when the current state is updated by calling setState.

- **componentWillReceiveProps** gets executed when the props have changed and is not first render. Sometimes state depends on the props, hence whenever props changes the state should also be synced. This is the method where it should be done. The similar method for the state doesn't exist before state change because the props are read only within a component and can never be dependent on the state.
- **shouldComponentUpdate** tells the React that when the component receives new props or state is being updated, should React re-render or it can skip rendering? This method is a question, should the Component be Updated? Hence this method should return true or false, and accordingly the component would be re-rendered or skipped. By default, this method return true.
- **componentWillUpdate** is executed only after the shouldComponentUpdate returns true. This method is only used to do the preparation for the upcoming render, similar to componentWillMount or constructor. There can be some use case when there needs some calculation or preparation before rendering some item, this is the place to do so.
- **render** And then the component gets rendered.
- **componentDidUpdate** is executed when the new updated component has been updated in the DOM. This method is used to re trigger the third party libraries used to make sure these libraries also update and reload themselves.

4) Unmounting

In this phase, the component is not needed and the component will get unmounted from the DOM. The method which is called in this phase

- **componentWillUnmount** This method is the last method in the lifecycle. This is executed just before the component gets removed from the DOM.