UNIT-3

REACT FOR FRONT-END DEVLOPMENT

3.1 LISTS AND KEYS

3.1.1 Lists

Lists are an important aspect within your app. Every application is bound to make use of lists in some form or the other. You could have a list of tasks like a calendar app, list of pictures like Instagram, list of items to shop in a shopping cart and so on. The use-cases are numerous. Lists within an application can be performance heavy. Imagine an app with a huge list of videos or pictures and you keep getting thousands more, as you scroll. That could take a toll on the app's performance.

Because performance is an important aspect, when you are using lists you need to make sure they are designed for optimal performance.

In JavaScript, the .map() method iterates through the parent array, calling a function on each element. Then it creates a new array containing the values that have been changed. It does not affect the parent array.

Lists are very useful when it comes to developing the UI of any website. Lists are mainly used for displaying menus on a website, for example, the navbar menu. In regular JavaScript, we can use arrays for creating lists.

Creating and traversing a list:

We can create lists in React in a similar manner as we do in regular JavaScript i.e. by storing the list in an array. In order to traverse a list we will use the map() function.

Step 1: Create a list of elements in React in the form of an array and store it in a variable. We will render this list as an unordered list element in the browser.

Step 2: We will then traverse the list using the JavaScript map() function and updates elements to be enclosed between elements.

Step 3: Finally we will wrap this new list within elements and render it to the DOM.

Write the below code in the index.js file of your react application:

```
import React from 'react';
import ReactDOM from 'react-dom';
const numbers = [1,2,3,4,5];
constupdatedNums = numbers.map((number)=>{
   return <li>{number}</li>;
});

ReactDOM.render(
<ul>
      {updatedNums}
</ul>,
document.getElementById('root')
);
```

Output: The above code will render an unordered list as shown below

- 1
- 2
- 3
- 4
- 5

Example

```
function ListComponent(props) {
constlistItems = myList.map((item) =>
<li>{item}</li>
  );
  return (
<ul>{listItems}</ul>
  );
}
constmyList = ["apple", "orange", "strawberry", "blueberry", "avocado"];
ReactDOM.render(
<ListComponentmyList={myList} />,
document.getElementById('root')
);
```

Output:

- apple
- orange
- strawberry
- avocado

3.1.2 Keys

What is a key in React?

A “key” is a special string attribute you need to include when creating lists of elements in React. Keys are used in React to identify which items in the list are changed, updated, or deleted.

Keys are used to give an identity to the elements in the lists. It is recommended to use a string as a key that uniquely identifies the items in the list. Assigning keys to the list.

You can assign the array indexes as keys to the list items. The below example assigns array indexes as keys to the elements.

The element's key is a specific property that must be included in the element, and it must be a string. Each list's key should be distinct, which means you shouldn't use values that are identical to the key. Each item within an array must have a unique key, but the key need not be globally unique. The same key can be used across various unrelated components and lists. To put it differently, keys should be unique among siblings rather than globally.

The element's key serves as a form of identifier for React, allowing it to figure out which element was updated, added, or removed.

It's a good idea to pick a value that serves as a unique identifier for each item in the array, which is usually the ID.

Syntax:

```
const numbers = [1, 2, 3, 4, 5];
constupdatedNums = numbers.map((number, index) =>
<li key={index}>
      {number}
</li>
);
```

Issue with assigning index as keys:

Assigning indexes as keys are highly discouraged because if the elements of the arrays get reordered in the future then it will get confusing for the developer as the keys for the elements will also change.

Difference between keys and props in React:

Keys are not the same as props, only the method of assigning “key” to a component is the same as that of props. Keys are internal to React and can not be accessed from inside of the component like props. Therefore, we can use the same value we have assigned to the Key for any other prop we are passing to the Component.

Using Keys with Components:

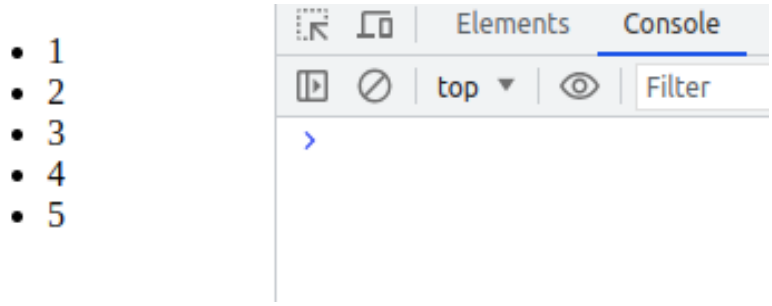
Consider a situation where you have created a separate component for list items and you are extracting list items from that component. In that case, you will have to assign keys to the component you are returning from the iterator and not to the list items. That is you should assign keys to <Component /> and not to <li> A good practice to avoid mistakes is to keep in mind that anything you are returning from inside of the map() function is needed to be assigned a key.

Example

```
import React from "react";
import ReactDOM from "react-dom";
// Component to be extracted
function MenuItems(props) {
const item = props.item;
   return <li>{item}</li>;
}
// Component that will return an
// unordered list
function Navmenu(props) {
const list = props.menuitems;
constupdatedList = list.map((listItems) => {
      return <MenuItems key={listItems.toString()} item={listItems} />;
   });
   return <ul>{updatedList}</ul>;
}
```

```
constmenuItems = [1, 2, 3, 4, 5];
ReactDOM.render(
<Navmenumenuitems={menuItems} />,
document.getElementById("root")
);
```

Output: In the below-given output you can clearly see that this time the output is rendered but without any type of warning in the console, it is the correct way to use React Keys.



Example

```
function ListComponent(props) {
   constlistItems = myList.map((item) =>
   <li key={item.id}>
      {item.value}
   </li>
   );
   return (
   <ul>{listItems}</ul>
   );}
constmyList = [{id: 'a', value: ‘Lotus’},
            {id: 'b', value: ‘Rose’},
            {id: 'c', value: ‘Sunflower’},
            {id: 'd', value: ‘Marigold’},
            {id: 'e', value: ‘Lily’}];
ReactDOM.render(
   <ListComponentmyList={myList} />,
   document.getElementById('root')
);
```

3.2 React Lifecycle

Lifecycle of Components

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.

The three phases are: **Mounting, Updating, and Unmounting.**

3.2.1 Mounting

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

- constructor()
- getDerivedStateFromProps()
- render()
- componentDidMount()

The render() method is required and will always be called, the others are optional and will be called if you define them.

3.2.1.1constructor

The constructor() method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state and other initial values.

The constructor() method is called with the props, as arguments, and you should always start by calling the super(props) before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (React.Component).

Example:

The constructor method is called, by React, every time you make a component:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
this.state = {favoritecolor: "red"};
  }
render() {
    return (
<h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}
ReactDOM.render(<Header />, document.getElementById('root'));
```

3.2.1.2 getDerivedStateFromProps

The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM.

This is the natural place to set the state object based on the initial props.

It takes state as an argument, and returns an object with changes to the state.

The example below starts with the favorite color being "red", but the getDerivedStateFromProps() method updates the favorite color based on the favcol attribute:

Example:

The getDerivedStateFromProps method is called right before the render method:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
render() {
    return (
<h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));
```

Render:

The render() method is required, and is the method that actually outputs the HTML to the DOM.

Example:

A simple component with a simple render() method:

```
class Header extends React.Component {
render() {
    return (
<h1>This is the content of the Header component</h1>
    );
  }
}
ReactDOM.render(<Header />, document.getElementById('root'));
```

3.2.1.3 componentDidMount

The componentDidMount() method is called after the component is rendered.

This is where you run statements that requires that the component is already placed in the DOM.

Example:

At first my favorite color is red, but give me a second, and it is yellow instead:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
this.state = {favoritecolor: "red"};
  }
componentDidMount() {
setTimeout(() => {
```

```
this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
render() {
    return (
<h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

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

3.2.2 Updating

The next phase in the lifecycle is when a component is updated.

A component is updated whenever there is a change in the component's state or props.

React has five built-in methods that gets called, in this order, when a component is updated:

getDerivedStateFromProps()

shouldComponentUpdate()

getSnapshotBeforeUpdate()

render()

componentDidUpdate()

The render() method is required and will always be called, the others are optional and will be called if you define them.

3.2.2.1 getDerivedStateFromProps

Also, at updates the getDerivedStateFromProps method is called. This is the first method that is called when a component gets updated.

This is still the natural place to set the state object based on the initial props.

The example below has a button that changes the favorite color to blue, but since the getDerivedStateFromProps() method is called, which updates the state with the color from the favcol attribute, the favorite color is still rendered as yellow:

Example:

If the component gets updated, the getDerivedStateFromProps() method is called:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
```

```
    return {favoritecolor: props.favcol };
  }
changeColor = () => {
this.setState({favoritecolor: "blue"});
  }
render() {
    return (
<div>
<h1>My Favorite Color is {this.state.favoritecolor}</h1>
<button type="button" onClick={this.changeColor}>Change color</button>
</div>
    );
  }
}
ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));
```

3.2.2.2 shouldComponentUpdate

In the shouldComponentUpdate() method you can return a Boolean value that specifies whether React should continue with the rendering or not.

The default value is true.

The example below shows what happens when the shouldComponentUpdate() method returns false:

Example:

Stop the component from rendering at any update:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
this.state = {favoritecolor: "red"};
  }
shouldComponentUpdate() {
    return false;
  }
changeColor = () => {
this.setState({favoritecolor: "blue"});
  }
render() {
    return (
<div>
<h1>My Favorite Color is {this.state.favoritecolor}</h1>
<button type="button" onClick={this.changeColor}>Change color</button>
</div>
    );
  }
}
```

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

Example:

Same example as above, but this time the shouldComponentUpdate() method returns true instead:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
this.state = {favoritecolor: "red"};
  }
shouldComponentUpdate() {
    return true;
  }
changeColor = () => {
this.setState({favoritecolor: "blue"});
  }
render() {
    return (
<div>
<h1>My Favorite Color is {this.state.favoritecolor}</h1>
<button type="button" onClick={this.changeColor}>Change color</button>
</div>
    );
  }
}
ReactDOM.render(<Header />, document.getElementById('root'));
```

3.2.2.3 getSnapshotBeforeUpdate

The getSnapshotBeforeUpdate() method is invoked just before the DOM is being rendered. It is used to store the previous values of the state after the DOM is updated.

Any value returned by getSnapshotBeforeUpdate() method will be used as a parameter for componentDidUpdate() method. This function is always used along with the componentDidUpdate() method but vice-versa isn't true.

Syntax:

```
getSnapshotBeforeUpdate(prevProps, prevState)
```

Parameters: It accepts two parameters, they are *prevProps* and *prevState* which are just the props or state before the component in question is re-rendered.

Example:

```
import React from 'react';
class App extends React.Component {
// Initializing the state
state = {
```

```
        name: 'ETT',
};
componentDidMount() {
        // Changing the state after 1 sec
        setTimeout(() => {
        this.setState({ name: 'Abc' });
        }, 1000);
}
getSnapshotBeforeUpdate(prevProps, prevState) {
        // Displaying the previous value of the state
        document.getElementById('prev').innerHTML ='Previous Name: ' + prevState.name;
}
componentDidUpdate() {
        // Displaying the current value of the state
        document.getElementById('new').innerHTML =
        'Current Name: ' + this.state.name;
}
render() {
        return (
        <div>
                <div id="prev"></div>
                <div id="new"></div>
                </div> );}
}
export default App;
```

3.2.2.4 render

The render() method is of course called when a component gets updated, it has to re-render the HTML to the DOM, with the new changes.

The example below has a button that changes the favorite color to blue:

Example:

Click the button to make a change in the component's state:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
this.state = {favoritecolor: "red"};
  }
changeColor = () => {
this.setState({favoritecolor: "blue"});
  }
render() {
    return (
<div>
<h1>My Favorite Color is {this.state.favoritecolor}</h1>
```

```
<button type="button" onClick={this.changeColor}>Change color</button>
</div>
    );
  }
}

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

3.2.2.5 componentDidUpdate

The componentDidUpdate method is called after the component is updated in the DOM.

The example below might seem complicated, but all it does is this:

When the component is mounting it is rendered with the favorite color "red".

When the component has been mounted, a timer changes the state, and the color becomes "yellow".

This action triggers the update phase, and since this component has a componentDidUpdate method, this method is executed and writes a message in the empty DIV element:

Example:

The componentDidUpdate method is called after the update has been rendered in the DOM:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
this.state = {favoritecolor: "red"};
  }
componentDidMount() {
setTimeout(() => {
this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
componentDidUpdate() {
document.getElementById("mydiv").innerHTML =
    "The updated favorite is " + this.state.favoritecolor;
  }
render() {
    return (
<div>
<h1>My Favorite Color is {this.state.favoritecolor}</h1>
<div id="mydiv"></div>
</div>  );
  }
}
ReactDOM.render(<Header />, document.getElementById('root'));
```

3.2.3 Unmounting

The next phase in the lifecycle is when a component is removed from the DOM, or unmounting as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

componentWillUnmount()

componentWillUnmount

The componentWillUnmount method is called when the component is about to be removed from the DOM.

Example:

Click the button to delete the header:

```
class Container extends React.Component {
  constructor(props) {
    super(props);
this.state = {show: true};
  }
delHeader = () => {
this.setState({show: false});
  }
render() {
    let myheader;
    if (this.state.show) {
myheader = <Child />;
    };
    return (
<div>
      {myheader}
<button type="button" onClick={this.delHeader}>Delete Header</button>
</div>
    );
}
}

class Child extends React.Component {
componentWillUnmount() {
alert("The component named Header is about to be unmounted.");
  }
render() {
    return (
<h1>Hello World!</h1>
    );
  }
}
ReactDOM.render(<Container />, document.getElementById('root'));
```

3.3Hooks

Hooks are the new feature introduced in the React 16.8 version. It allows you to use state and other React features without writing a class. Hooks are the functions which "hook into" React state and lifecycle features from function components. It does not work inside classes.

Rules of Hooks

Hooks are similar to JavaScript functions, but you need to follow these two rules when using them. Hooks rule ensures that all the stateful logic in a component is visible in its source code. These rules are:

1. Only call Hooks at the top level

Do not call Hooks inside loops, conditions, or nested functions. Hooks should always be used at the top level of the React functions. This rule ensures that Hooks are called in the same order each time a componentrenders.

2. Only call Hooks from React functions

You cannot call Hooks from regular JavaScript functions. Instead, you can call Hooks from React function components. Hooks can also be called from custom Hooks.

Pre-requisites for React Hooks

Node version 6 or above

NPM version 5.2 or above

Create-react-app tool for running the React App

3.3.1 Hooks useState()

Hook state is the new way of declaring a state in React app. Hook uses useState() functional component for setting and retrieving state. Let us understand Hook state with the following example.

App.js

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

function CountApp() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        click me
      </button>
    </div>
  );
}
export default CountApp;
```

OUTPUT

You clicked 3 times

Click me

3.3.2 Hooks useEffect()

The Effect Hook allows us to perform side effects (an action) in the function components. It does not use components lifecycle methods which are available in class components. In other words, Effects Hooks are equivalent to componentDidMount(), componentDidUpdate(), and componentWillUnmount() lifecycle methods.

Side effects have common features which the most web applications need to perform, such as:

Updating the DOM, Fetching and consuming data from a server API, Setting up a subscription, etc.

App.js

```
import React, { useState, useEffect } from 'react';
function CounterExample() {
  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>
  );
}
export default CounterExample;
```

OUTPUT

You clicked 2 times

Click me

3.4 Forms

Just like in HTML, React uses forms to allow users to interact with the web page.

Handling Forms

- Handling forms is about how you handle the data when it changes value or gets submitted.
- In HTML, form data is usually handled by the DOM.
- In React, form data is usually handled by the components.

- When the data is handled by the components, all the data is stored in the component state.
- You can control changes by adding event handlers in the onChange attribute.
- We can use the useState() Hook to keep track of each inputs value and provide a "single source of truth" for the entire application

Example

```
import {useState } from "react";
import ReactDOM from "react-dom/client";

function MyForm() {
const [inputs, setInputs] = useState({});
consthandleChange = (event) => {
const name = event.target.name;
const value = event.target.value;
setInputs(values => ({...values, [name]: value}))
  }
consthandleSubmit = (event) => {
event.preventDefault();
    console.log(inputs);
  }
  return (
<form onSubmit={handleSubmit}>
<label>Enter your name
<input
       type="text"
       name="username"
       value={inputs.username || ""}
onChange={handleChange}
     />
</label>
<label>Enter your age
<input
        type="number"
        name="age"
        value={inputs.age || ""}
onChange={handleChange}
      />
</label>
          <input type="SUBMIT" />
</form>
  )
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```

OUTPUT

SUBMIT

Enter your name Enter your age

3.5 Pulling data from an API

API: API is an abbreviation for Application Programming Interface which is a collection of communication protocols and subroutines used by various programs to communicate between them. A programmer can make use of various API tools to make its program easier and simpler. Also, an API facilitates the programmers with an efficient way to develop their software programs.

Approach: We will know how we fetch the data from API (Application Programming Interface). For the data, we have used the API endpoint from http://jsonplaceholder.typicode.com/users we have created the component in App.js and styling the component in App.css. From the API we have target "id", "name", "username", "email" and fetch the data from API endpoints. Below is the stepwise implementation of how we fetch the data from an API in react. We will use the fetch function to get the data from the API.

Step by step implementation to fetch data from an api in react:

Step 1: Create React Project

npm create-react-app MY-APP

Step 2: Change your directory and enter your main folder charting as

cd MY-APP

Step 3: API endpoint

https://jsonplaceholder.typicode.com/users

Step 4: Write code in App.js to fetch data from API and we are using fetch function.

Example:

```
import React from "react";
class App extends React.Component {

  constructor(props) {
    super(props);
    this.state = { items: [], DataisLoaded: false };
  }
  // ComponentDidMount is used to execute the code
  componentDidMount() {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then((res) => res.json())
      .then((json) => {this.setState({ items: json, DataisLoaded: true });});
  }

  render() {
    const { DataisLoaded, items } = this.state;
    if (!DataisLoaded)
      return (
```

```
      <div>
        <h1> Pleses wait some time.... </h1>
      </div>
    );

  return (
    <div className="App">
      <h1> Fetch data from an api in react </h1>
      <ol>
        {items.map((item) => (
          <li key={item.id}>
            User_Name: {item.username}, Full_Name: {item.name}, User_Email: {item.email}
          </li>
        ))}
      </ol>
    </div>
  );
 }
}

export default App;
```