24CA206 Full stack IV Unit React JS

UNIT IV ADVANCED CLIENT SIDE PROGRAMMING

React JS: ReactDOM - JSX - Components - Properties – Fetch API - State and Lifecycle - -JS Localstorage - Events - Lifting State Up - Composition and Inheritance.

React JS

React is a JavaScript library for building user interfaces.
React is used to build single-page applications. React allows us to create reusable UI components.
ReactJS is a simple, feature rich, component based JavaScript UI library. It can be used to develop small applications as well as big, complex applications.
React community provides large set of ready-made components to develop web application in a record time.
It was created by Facebook back in 2013.

Features

The salient features of *React library* are as follows −

- Solid base architecture
- Extensible architecture
- Component based library
- JSX based design architecture
- Declarative UI library

Benefits

Few benefits of using *React library* are as follows −

- Easy to learn
- Easy to adept in modern as well as legacy application
- Faster way to code a functionality
- Availability of large number of ready-made component
- Large and active community

Applications

Few popular websites powered by *React library* are listed below −

- *Facebook*, popular social media application
- *Instagram*, popular photo sharing application
- *Netflix*, popular media streaming application
- *Code Academy*, popular online training application
- *Reddit*, popular content sharing application

React is a tool for building UI components.

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Hello(props) {
  return <h1>Hello World!</h1>;
}

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<Hello />);
```

React DOM

The react-dom package provides DOM-specific methods that can be used at the top level of your app.

```
import * as  from 'react-dom';
```

The react-dom package also provides modules specific to client and server apps:
react-dom/client
react-dom/server

The react-dom package exports these methods:

createPortal()
flushSync()

createPortal(): Creates a portal. Portals provide a way to render children into a DOM node that exists outside the hierarchy of the DOM component.

```
createPortal(child, container)
```

flushSync(): Force React to flush any updates inside the provided callback synchronously. This ensures that the DOM is updated immediately.

```
flushSync(callback)
```

```
// Force this state update to be synchronous.
flushSync(() => {
  setCount(count + 1);
});
```

These react-dom methods are also exported, but are considered legacy:
render()
hydrate()
findDOMNode()
unmountComponentAtNode()

render(): render() controls the contents of the container node you pass in.

```
render(element, container[, callback])
```

If the React element was previously rendered into container, this will perform an update on it and only mutate the DOM as necessary to reflect the latest React element.

hydrate():React expects that the rendered content is identical between the server and the client. It can patch up differences in text content, but you should treat mismatches as bugs and fix them.

```
hydrate(element, container[, callback])
```

findDOMNode(): findDOMNode is an escape hatch used to access the underlying DOM node.
unmountComponentAtNode() : Remove a mounted React component from the DOM and clean up its event handlers and state. If no component was mounted in the container, calling this function does nothing.

React- JSX

JSX stands for JavaScript XML.
JSX allows us to write HTML in React.
JSX makes it easier to write and add HTML in React.

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement() and/or appendChild() methods.

JSX converts HTML tags into react elements.

Example:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = <h1>I Learn JSX!</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Output:
I Learn JSX!

Expressions in JSX
With JSX you can write expressions inside curly braces { }.

The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:

Execute the expression 5 + 5:

```
const myElement = <h1>React is {5 + 5} times better with JSX</h1>;
```

Inserting a Large Block of HTML
To write HTML on multiple lines, put the HTML inside parentheses:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = (
  <ul>
    <li>Apples</li>
    <li>Bananas</li>
    <li>Cherries</li>
  </ul>
);
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Output

localhost:3000

- Apples
- Bananas
- Cherries

Elements Must be Closed

JSX follows XML rules, and therefore HTML elements must be properly closed.

```
const myElement = <input type="text" />;
```

Attribute class = className

The class attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the class keyword is a reserved word in JavaScript, you are not allowed to use it in JSX.

```
const myElement = <h1 className="myclass">Hello World</h1>;
```

Conditions - if statements

React supports if statements, but not *inside* JSX.

To be able to use conditional statements in JSX, you should put the if statements outside of the JSX, or you could use a ternary expression instead:

Example

Write "Hello" if x is less than 10, otherwise "Goodbye":

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const x = 5;
let text = "Goodbye";
if (x < 10) {
  text = "Hello";
}

const myElement = <h1>{text}</h1>;
onst root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Output

Hello

Functions

JSX supports user defined JavaScript function. Function usage is similar to expression. Let us create a simple function and use it inside JSX.

```
<script type="text/babel">
  var cTime = new Date().toTimeString();
  ReactDOM.render(
    <div><p>The current time is {cTime}</p></div>,
    document.getElementById('react-app')
  );
</script>
```

Output

Here, *getCurrentTime()* is used get the current time.

```
The Current time is 21:19:56 GMT+0530(India Standard Time)
```

Components

React component is the building block of a React application.

A React component represents a small chunk of user interface in a webpage. The primary job of a React component is to render its user interface and update it whenever its internal state is changed.

React component provides below functionalities.

- Initial rendering of the user interface.
- Management and handling of events.
- Updating the user interface whenever the internal state is changed.

React component accomplish these feature using three concepts −

- Properties − Enables the component to receive input.
- Events − Enable the component to manage DOM events and end-user interaction.
- State − Enable the component to stay stateful. Stateful component updates its UI with respect to its state.

Components come in two types, Class components and Function components.
In older React code bases, you may find Class components primarily used.

Class Component

A class component must include the extends React.Component statement. This statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.

The component also requires a render() method, this method returns HTML.

Example

Create a Class component called Car.

```
class Car extends React.Component {
render() {
   return <h2>Hi, I am a Car!</h2>;
  }
}
```

Function Component

A Function component also returns HTML, and behaves much the same way as a Class component, but Function components can be written using much less code, are easier to understand.

Example

Create a Function component called Car.

React component can also be created using plain JavaScript function but with limited features.

```
function Car() {
  return <h2>Hi, I am a Car!</h2>;
}
```

Rendering a Component

Now your React application has a component called Car, which returns an <h2> element.

To use this component in your application, use similar syntax as normal HTML: <Car />

Example

Display the Car component in the "root" element:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

Props

Components can be passed as props, which stands for properties.

Props are like function arguments, and you send them into the component as attributes.

Example

Use an attribute to pass a color to the Car component, and use it in the render() function:

```
function Car(props) {
  return <h2>I am a {props.color} Car!</h2>;
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car color="red"/>);
```

Components in Components

We can refer to components inside other components:

Use the Car component inside the Garage component:

```
function Car() {
  return <h2>I am a Car!</h2>;
}

function Garage() {
  return (
    <>
      <h1>Who lives in my Garage?</h1>
      <Car />
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

Components in Files

React is all about re-using code, and it is recommended to split your components into separate files.

To do that, create a new file with a .js file extension and put the code inside it:

```
function Car() {
  return <h2>Hi, I am a Car!</h2>;
}

export default Car;
```

React Props

props stands for properties.
Props are arguments passed into React components.
Props are passed to components via HTML attributes.
React Props are like function arguments in JavaScript and attributes in HTML.

To send props into a component, use the same syntax as HTML attributes:

Example
Add a "brand" attribute to the Car element:

```
const myElement = <Car brand="Ford" />;
```

The component receives the argument as a props object:
Example
Use the brand attribute in the component:

```
function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}
```

Pass Data
Props are also how you pass data from one component to another, as parameters.

Example
Send the "brand" property from the Garage component to the Car component:

```
function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

function Garage() {
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand="Ford" />
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

React properties supports attribute's value of different types. They are as follows,

- String
- Number
- Datetime
- Array
- List
- Objects

Fetch 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.
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.

App.js

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

	// Constructor
	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> {
                    items.map((item) => (
                    <ol key = { item.id } >
                        User_Name: { item.username },
                        Full_Name: { item.name },
                        User_Email: { item.email }
                        </ol>
                    ))
                }
            </div>
        );
}
}

export default App;
```

State management (State)

State management is one of the important and unavoidable features of any dynamic application. React provides a simple and flexible API to support state management in a React component. .
State represents the value of a dynamic properties of a React component at a given instance. React provides a dynamic data store for each component. The internal data represents the state of a React component and can be accessed using this.state member variable of the component. Whenever the state of the component is changed, the component will re-render itself by calling the render() method along with the new state.

A simple example to better understand the state management is to analyse a real-time clock component. The clock component primary job is to show the date and time of a location at the given instance. As the current time will change every second, the clock component should maintain the current date and time in it's state. As the state of the clock component changes every second, the clock's render() method will be called every second and the render() method show the current time using it's current state.

The simple representation of the state is as follows −

```
{
  date: '2020-10-10 10:10:10'
}
```

The React useState Hook allows us to track state in a function component.

Import useState
To use the useState Hook, we first need to import it into our component.

Example:
At the top of your component, import the useState Hook.

```
import { useState } from "react";
```

Initialize useState
We initialize our state by calling useState in our function component.

useState accepts an initial state and returns two values:

The current state.
A function that updates the state.

```
import { useState } from "react";

function FavoriteColor() {
  const [color, setColor] = useState("");
}
```

Read State

We can now include our state anywhere in our component.

```
function FavoriteColor() {
  const [color, setColor] = useState("red");

  return <h1>My favorite color is {color}!</h1>
}
```

Update State

To update our state, we use our state updater function.

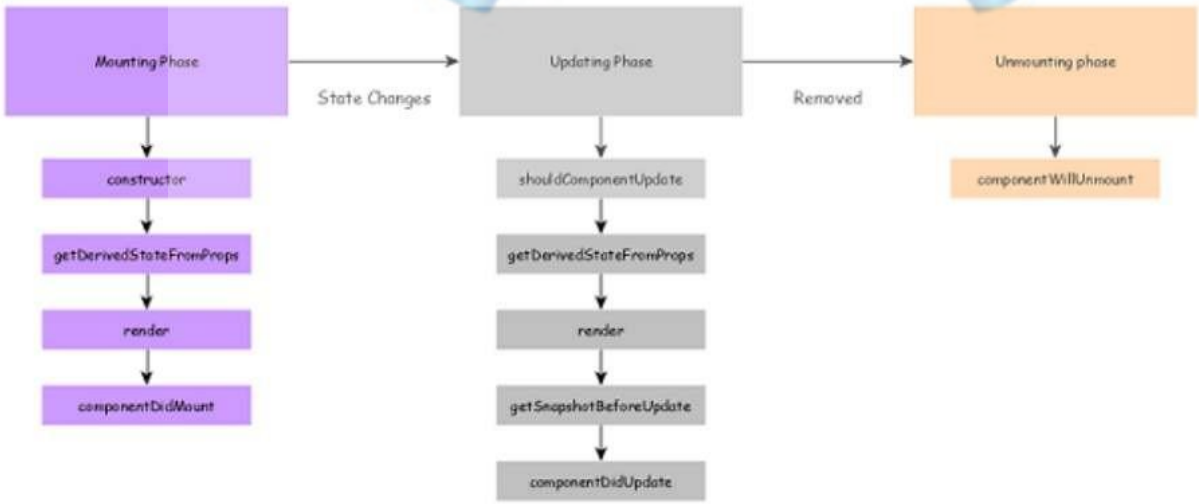
ReactJS - Component Life Cycle

In React, Life cycle of a component represents the different stages of the component during its existence. React provides callback function to attach functionality in each and every stages of the React life cycle.

Each React component has three distinct stages.

Mounting − Mounting represents the rendering of the React component in the given DOM node.
Updating − Updating represents the re-rendering of the React component in the given DOM node during state changes / updates.
Unmounting − Unmounting represents the removal of the React component.

React provides a collection of life cycle events (or callback API) to attach functionality, which will to be executed during the various stages of the component.



constructor() − Invoked during the initial construction phase of the React component. Used to set initial state and properties of the component.

render() − Invoked after the construction of the component is completed. It renders the component in the virtual DOM instance. This is specified as mounting of the component in the DOM tree.

componentDidMount() − Invoked after the initial mounting of the component in the DOM tree. It is the good place to call API endpoints and to do network requests. In our clock component, setInterval function can be set here to update the state (current date and time) for every second.

```
componentDidMount() {
  this.timeFn = setInterval( () => this.setTime(), 1000);
}
```

componentDidUpdate() − Similar to ComponentDidMount() but invoked during the update phase. Network request can be done during this phase but only when there is difference in component's current and previous properties.

The signature of the API is as follows −

componentDidUpdate(prevProps, prevState, snapshot)
prevProps − Previous properties of the component.

prevState − Previous state of the component.

snapshot − Current rendered content.

componentWillUnmount() − Invoked after the component is unmounted from the DOM. This is the good place to clean up the object. In our clock example, we can stop updating the date and time in this phase.

```
componentDidMount() {
  this.timeFn = setInterval( () => this.setTime(), 1000);
}
```

shouldComponentUpdate() − Invoked during the update phase. Used to specify whether the component should update or not. If it returns false, then the update will not happen.

The signature is as follows −

shouldComponentUpdate(nextProps, nextState)
nextProps − Upcoming properties of the component

nextState − Upcoming state of the component

```
import React from 'react';

class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      date: new Date()
    }
  }
  componentDidMount() {
    this.setTimeRef = setInterval(() => this.setTime(), 1000);
}
  componentWillUnmount() {
    clearInterval(this.setTimeRef)
  }
```

```
  setTime() {
    this.setState((state, props) => {
      console.log(state.date);
      return {
        date: new Date()
      }
    })
  }
  render() {
     return (
       <div>
         <p>The current time is {this.state.date.toString()}</p>
       </div>
     );

}
}
export default Clock;
```

ReactJS - Event management : (Events)

Event management is one of the important features in a web application. It enables the user to interact with the application. React support all events available in a web application. React event handling is very similar to DOM events with little changes.
React can perform actions based on user events.
React has the same events as HTML: click, change, mouseover etc.

Adding Events
React events are written in camelCase syntax:
onClick instead of onclick.

React event handlers are written inside curly braces:
onClick={shoot} instead of onClick="shoot()".

React:
```
<button onClick={shoot}>Take the Shot!</button>
```

Example:
Put the shoot function inside the Football component:

```
function Football() {
  const shoot = () => {
  alert("Great Shot!");
  }

  return (
    <button onClick={shoot}>Take the shot!</button>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football />);
```

Passing Arguments
To pass an argument to an event handler, use an arrow function.

```
function Football() {
```

```
  const shoot = (a) => {
    alert(a);
  }

  return (
    <button onClick={() => shoot("Goal!")}>Take the shot!</button>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football />);
```

React Event Object

Event handlers have access to the React event that triggered the function.

In our example the event is the "click" event.

Arrow Function: Sending the event object manually:

```
function Football() {
  const shoot = (a, b) => {
  alert(b.type);

  }

  return (
    <button onClick={(event) => shoot("Goal!", event)}>Take the shot!</button>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football />);
```

Lifting up the State:

As we know, every component in React has its own state. Because of this sometimes data can be redundant and inconsistent. So, by Lifting up the state we make the state of the parent component as a single source of truth and pass the data of the parent in its children.

Time to use Lift up the State: If the data in “parent and children components” or in “cousin components” is Not in Sync.

Example 1: If we have 2 components in our App. A -> B where, A is parent of B. keeping the same data in both Component A and B might cause inconsistency of data.

Example 2: If we have 3 components in our App.

Where A is the parent of B and C. In this case, If there is some Data only in component B but, component C also wants that data. We know Component C cannot access the data because a component can talk only to its parent or child.

we will Lift the state of component B and component C to component A. Make A.js as our Main Parent by changing the path of App in the index.js file

Before:
import App from './App';
After:
import App from './A';

Filename- A.js:

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

class A extends Component {

constructor(props) {
        super(props);
        this.handleTextChange = this.handleTextChange.bind(this);
        this.state = {text: ''};
}

handleTextChange(newText) {
        this.setState({text: newText});
}

render() {
        return (
                <React.Fragment>
                <B text={this.state.text}
                        handleTextChange={this.handleTextChange}/>
                <C text={this.state.text} />
                </React.Fragment>
        );
}
}

export default A;
```

Filename- B.js:

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

class B extends Component {

 constructor(props) {
         super(props);
        this.handleTextChange = this.handleTextChange.bind(this);
}

handleTextChange(e){
        this.props.handleTextChange(e.target.value);
}

render() {
        return (
                <input value={this.props.text}
                        onChange={this.handleTextChange} />
```

```
        );
}
}

export default B;
```

Filename- C.js:

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

class C extends Component {

render() {
        return (
                <h3>Output: {this.props.text}</h3>
        );
}
}

export default C;
```

Output: Now, component C can Access text in component B through component A.

Composition and inheritance:

Composition and inheritance are the approaches to use multiple components together in React.js . This helps in code reuse. React recommend using composition instead of inheritance as much as possible and inheritance should be used in very specific cases only.

Example to understand it −

Let's say we have a component to input username.

Inheritance

```
class UserNameForm extends React.Component {
  render() {
    return (
      <div>
        <input type="text" />
      </div>
    );
  }
}
ReactDOM.render(
  < UserNameForm />,
  document.getElementById('root'));
```

This sis simple to just input the name. We will have two more components to create and update the username field.

With the use of inheritance we will do it like −

```
class UserNameForm extends React.Component {
  render() {
```

```
    return (
      <div>
        <input type="text" />
      </div>
    );
  }
}
class CreateUserName extends UserNameForm {
  render() {
    const parent = super.render();
    return (
      <div>
{parent}
        <button>Create</button>
      </div>
    )
  }
}
class UpdateUserName extends UserNameForm {
  render() {
    const parent = super.render();
    return (
      <div>
        {parent}
        <button>Update</button>
      </div>
    )
  }
}

ReactDOM.render(
  (<div>
    < CreateUserName />
    < UpdateUserName />
  </div>), document.getElementById('root')
);
```

We extended the UserNameForm component and extracted its method in child component using super.render();

Composition

```
class UserNameForm extends React.Component {
  render() {
    return (
      <div>
        <input type="text" />
      </div>
    );
  }
}



class CreateUserName extends React.Component {
  render() {
    return (
```

```
        <div>
< UserNameForm />
          <button>Create</button>
        </div>
      )
    }
}
class UpdateUserName extends React.Component {
  render() {
    return (
      <div>
        < UserNameForm />
        <button>Update</button>
      </div>
    )
  }
}

ReactDOM.render(
  (<div>
    <CreateUserName />
    <UpdateUserName />
  </div>), document.getElementById('root')
);
```

Use of composition is simpler than inheritance and easy to maintain the complexity.