Module -18 : React-Js for Full Stack

THEORY ASSIGNMENT

---

1. Introduction to React.js

Question 1: What is React.js? How is it different from other JavaScript frameworks and libraries?

React.js is an open-source **JavaScript library** developed by Facebook for building **user interfaces**, especially **single-page applications**. It uses a **component-based** architecture and a **virtual DOM** for fast rendering.

How it's different from others:

- **Library**, not a full framework (unlike Angular/Vue).
- Uses **virtual DOM** (faster than real DOM).
- Has **one-way data binding** (easier to debug).
- Requires extra libraries for routing or state (e.g., React Router, Redux).
- Focuses only on the **view layer** (UI part) of apps.

Question 2: Explain the core principles of React such as the virtual DOM and component- based architecture.

Core Principles of React:

1. **Virtual DOM**:
   React creates a lightweight copy of the real DOM in memory. It updates this virtual DOM first, then efficiently updates only the changed parts in the real DOM, improving performance.

2. **Component-Based Architecture**:
   UI is broken into **reusable components**. Each component manages its own state and logic, making code modular, maintainable, and easier to debug.
3. **Unidirectional Data Flow**:
   Data flows in one direction (from parent to child), which makes the app's data flow predictable and easier to manage.
4. **Declarative UI**:
   Developers describe **what the UI should look like**, and React takes care of rendering it correctly based on state and props.

Question 3: What are the advantages of using React.js in web development?

Advantages of Using React.js in Web Development:

1. **Reusable Components**: Promotes modular and maintainable code.
2. **Fast Rendering**: Virtual DOM boosts performance by minimizing real DOM updates.
3. **Declarative Syntax**: Makes UI easier to understand and debug.
4. **Unidirectional Data Flow**: Predictable data handling improves reliability.
5. **Strong Community & Ecosystem**: Lots of libraries, tools, and support.
6. **SEO-Friendly**: With tools like Next.js, React apps can be optimized for search engines.
7. **Cross-Platform Development**: Can be used for mobile apps with React Native.

2. JSX (JavaScript XML)

Question 1: What is JSX in React.js? Why is it used?

JSX (JavaScript XML) is a **syntax extension** for JavaScript used in **React.js**. It allows you to write **HTML-like code inside JavaScript**.

Why JSX is used:

1. **Simplifies UI Code**: Makes it easier to write and understand component structure.
2. **Combines Markup and Logic**: You can embed JavaScript expressions inside JSX using {}.
3. **Improves Readability**: Looks similar to HTML, so it's more intuitive for frontend developers.
4. **Better Tooling**: Works well with IDEs for syntax highlighting, auto-completion, and error checking.

JSX isn't required in React, but it makes code cleaner and more expressive.

Question 2: How is JSX different from regular JavaScript? Can you write JavaScript inside JSX?

JSX (JavaScript XML) is a syntax extension for JavaScript that is commonly used with React. It allows developers to write HTML-like code within JavaScript. This makes it easier to visualize the component structure and improves code readability.

Key Differences Between JSX and Regular JavaScript:

1. **Syntax**:
    - JSX looks like HTML, but it's not HTML. It's syntactic sugar for React.createElement() calls.
    - JavaScript does not natively support HTML-like syntax, but JSX is transpiled into standard JavaScript.
2. **Transpilation**:
    - Browsers cannot understand JSX directly. It needs to be compiled (typically by Babel) into regular JavaScript before being run.
3. **Embedding HTML and Logic**:
    - In regular JavaScript, HTML-like structures must be created manually using methods like document.createElement().
    - JSX allows for a more declarative way to describe UI components.

```
const name = "John";

const element = <h1>Hello, {name}!</h1>;
```

Question 3: Discuss the importance of using curly braces {} in JSX expressions.

Curly braces {} in JSX are **essential for embedding JavaScript expressions** inside the JSX markup. They allow you to dynamically insert values, evaluate expressions, and render content conditionally within your component's UI.

1. **Embedding Dynamic Data:**
    - Curly braces are used to insert variables or expressions.

```
const name = "Alice";

return <h1>Hello, {name}!</h1>; // Outputs: Hello, Alice!
```

2. **Using JavaScript Expressions:**
    - You can insert logic like string concatenation, mathematical calculations, and ternary operators.

```
<p>{5 + 10}</p>          // Outputs: 15

<p>{isLoggedIn ? "Welcome" : "Please log in"}</p>
```

3. **Calling Functions:**
    - You can call JavaScript functions inside curly braces.
    - <h2>{getGreetingMessage()}</h2>
4. **Rendering Lists:**
    - Often used with .map() to render lists of elements.

```
const items = [1, 2, 3];

return <ul>{items.map(i => <li key={i}>{i}</li>)}</ul>;
```

3.Components (Functional & Class Components)

Question 1: What are components in React? Explain the difference between functional components and class components.

In React, **components** are the **building blocks of a user interface**. A component is a **reusable piece of UI** that can contain its own structure (HTML), styling (CSS), and behavior (JavaScript logic). Components can be combined together to build complex interfaces.

React applications are typically made up of **multiple components** that interact with each other through **props** and **state**.

Types of Components in React:

There are two main types:

1. **Functional Components**
2. **Class Components**

🔹 1. Functional Components

- These are **JavaScript functions** that return JSX.
- Introduced as "stateless" components, but with React Hooks, they can now manage state and side effects.
- Simpler and more concise.

Example:

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

With Hooks:

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

function Counter() {
  const [count, setCount] = useState(0);
```

```
  return <button onClick={() => setCount(count + 1)}>Count: {count}</button>;
}
```

- **2. Class Components**
  - Use ES6 class syntax.
  - Must extend React.Component.
  - Have a render() method that returns JSX.
  - Support lifecycle methods like componentDidMount, componentDidUpdate, etc.

Example:

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

class Greeting extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

Question 2: How do you pass data to a component using props?

Props (short for **properties**) are a way to **pass data from a parent component to a child component** in React. Props are **read-only** and help make components **reusable and dynamic**.

How to Pass Data Using Props

1. **Pass Props from Parent:**
   You provide props as attributes when using the child component in JSX.

```
<Greeting name="Alice" age={25} />
```

2. **Access Props in Child Component:**

- **Functional Component:**

```
function Greeting(props) {
  return <h1>Hello, {props.name}. You are {props.age} years old.</h1>;
}
```

- **Using Destructuring:**

```
function Greeting({ name, age }) {
  return <h1>Hello, {name}. You are {age} years old.</h1>;
}
```

- **Class Component:**

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Question 3: What is the role of render() in class components?

In React **class components**, the render() method is **required** and plays a central role. It is responsible for **returning the JSX** that defines the **UI output** of the component.

Key Responsibilities of render():

1. **Returns JSX**:
    - It must return a single parent element (or a fragment).
    - The JSX returned by render() is what React uses to update the DOM.

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

```
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

2. **Re-renders on State/Prop Change**:
    - Whenever state or props change, React automatically calls the render() method again to update the UI.
3. **Pure Function**:
    - It should be a **pure function** of props and state — meaning no side effects (like API calls or timers) should happen here.
    - Side effects should go in lifecycle methods like componentDidMount().

Example:

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
```

```
      <button onClick={this.increment}>Increment</button>
    </div>
  );
 }
}
```

4. Props and State

Question 1: What are props in React.js? How are props different from state?

- **Props** (short for **properties**) are a way to **pass data from a parent component to a child component**.
- Props are **read-only** inside the child component — they cannot be changed there.
- They help make components **dynamic and reusable** by allowing different data to be passed in.

Example:

```
function Welcome(props) {
 return <h1>Hello, {props.name}!</h1>;
}

// Usage
<Welcome name="Alice" />
```

What is State in React.js?

- **State** is a set of data that **belongs to and is managed within a component**.
- State is **mutable** — it can be changed by the component using setState (class) or useState (functional).

- Changes to state trigger **re-rendering** of the component.

Example:

```
function Counter() {
  const [count, setCount] = React.useState(0);

  return <button onClick={() => setCount(count + 1)}>Count: {count}</button>;
}
```

Question 2: Explain the concept of state in React and how it is used to manage component data.

- **State** is a **built-in object** in React components that holds data or information about the component.
- It represents the **dynamic parts of a component** — data that can change over time or in response to user actions.
- When state changes, React **re-renders the component** to update the UI automatically.

How State Manages Component Data:

- State allows components to **remember information** between renders.
- It controls **what the user sees** based on the current state.
- Examples of state data include form inputs, toggles, counters, or data fetched from an API.

🔹 Using State in Functional Components (with Hooks):

- React provides the useState hook to add state to functional components.

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

```
function Counter() {
  const [count, setCount] = useState(0); // Initialize state

  const increment = () => {
    setCount(count + 1); // Update state
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increase</button>
    </div>
  );
}
```

- useState(0) initializes the state with 0.
- Calling setCount updates the state and triggers a re-render.

- **Using State in Class Components:**
  - State is managed via the this.state object and updated using this.setState().

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 }; // Initialize state
  }
```

```
  increment = () => {
    this.setState({ count: this.state.count + 1 }); // Update state
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increase</button>
      </div>
    );
  }
}
```

Question 3: Why is this.setState() used in class components, and how does it work?

- In **React class components**, this.setState() is the **correct way to update the component's state**.
- You **should never modify this.state directly** because:
    - Direct mutation won't trigger React's **re-rendering**.
    - It can cause unpredictable behavior and bugs.
- Using this.setState() ensures React knows the state has changed and the component needs to update its UI.

How Does this.setState() Work?

- this.setState() **merges the new state object with the current state**.
- It **schedules an update** to the component's state and tells React to **re-render** the component with the new state.

- The update is **asynchronous**, meaning React may batch multiple setState() calls for performance optimization.

Example:

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  increment = () => {
    // Correct way to update state
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```