Chapter 08 Assignment Theory

1. **How do you create Nested Routes react-router-dom cofiguration?**
   To create nested routes we should add path and element property to the children property of the object created by using createBrowser.
   For example:

```
const router = createBrowserRouter([
    {
        path: "/",
        element: <Root />,
        children: [
            {
                path: '/team',
                element: <Team />,
                children: [
                    {
                        path: 'individual',
                        element: <Individual />
                    }
                ]
            },
        ]
    }
]);
```

   In the example shown above, we can access the <Individual /> by using the URL /team/individual

2. **Read abt createHashRouter, createMemoryRouter from React Router docs.**

createHashRouter	**createMemoryRouter**
The # part of the URL is not a new thing. We've seen websites using it for a long time. Typical use of # in modern websites is to scroll down to a specific section of an article.	Memory router keeps the URL changes in memory not in the user browsers. It keeps the history of the URL in memory and it does not read or write to the address bar so the user can not use the browser's back button as well as the forward button. It doesn't change the URL in your browser.
When using hash routing, the hash portion of the URL is never sent to a server. This means that changing URLs and navigating between pages won't make any server request. This is why hash routing doesn't require any server configuration. And it makes it a great candidate for single-page applications.	
Example: app.example.com/#/ - Dashboard app.example.com/#/profile - Profile	It is very useful for testing and non-browser environments like React Native.

3. **What is the order of lifecycle method in Class Based component?**
   React class component has three states :
   1. Mounting
   2. Updating
   3. Unmounting

   Each state has three essential phase:
   a. Render
   b. Pre-commit
   c. Commit

Initial Rendering	**Re-render**	**Unmounting**
constructor() -> render() -> componentDidMount ()	componentWillReceiveProps () -> shouldComponentUpdate() -> componentWillUpdate() -> render() ->componentWillreceiveProps ()	componentWillUnmount ()

4. **Why do we use componentDidMount?**
   The componentDidMount() method is the last step in the Mounting phase. This method is called post mounting. When all the children elements and components are mounted in the Document Object Model (DOM) then we call this method. This is where you run statements that requires that the component is already placed in the DOM.

5. **Why do we use componentWillUnmount? Explain with example**
   The componentWillUnmount method is called when the component is about to be removed from the DOM. This is the right place for any cleanup that we would want to do. For example, if you want to remove any setTimeOut or setInterval this is the function to clean it. Similarly, to prevent creation of multiple eventListeners we can clear the eventListeners in componentWillUnmount.
   Example::

```
componentDidMount() {
    window.addEventListener('scroll', this.onScroll, false);
  }

  componentWillUnmount() {
    window.removeEventListener('scroll', this.onScroll, false);
  }
```

6. **Why do we use super(props) in constructor**
   Super() is called to access the parent class's constructor. Consider the following code :

```
class Person {
  constructor(name) {
    this.name = name;
  }
}

class PolitePerson extends Person {
  constructor(name) {
    this.greetColleagues();
    super(name);
  }
  greetColleagues() {
    alert('Good morning folks! I am ',this.name);
  }
}
```

this.greetColleagues() is called before the *super()* call had a chance to set up this.name. So this.name isn’t even defined yet. Code like this can be very difficult to think about.To avoid such pitfalls, JavaScript enforces that if you want to use this in a constructor, you have to call super first.

Passing or not passing props to super has **no effect** on later uses of this.props outside constructor. That is render, shouldComponentUpdate, or event handlers always have access to it. But if you want to access *this.props* in constructor we will have to pass props to super**.**

7. **Why can't we have the callback function of useEffect async?**

You cannot directly make the callback function supplied to the useEffect hook async because:

- async functions implicitly return a promise, and;
- useEffect expects its callback to either return nothing or a clean-up function.

When you attempt to make useEffect's callback async, you will see the following warning:

Warning: Effect callbacks are synchronous to prevent race conditions