Assignment 7 - Let's get Classy

1. How do you create Nested Routes react-router-dom configuration?

- There are different types of Routers available in react-router-dom. Eg: createBrowserRouter
- createMemoryRouter
- createHashRouter
- createStaticRouter
- Here, we are using createBrowserRouter:

```
const AppLayout = () => {
  return (
    <div className="app">
      <Header />
      {/** Below will load different components based on path */}
      <Outlet />
    </div>
  );
};

const appRouter = createBrowserRouter([
  {
    path: "/",
    element: <AppLayout />,
    children: [
      {
        path: "/",
        element: <Body />,
      },
      {
        path: "/about",
        element: <About />,
      },
      {
        path: "/contactus",
        element: <Contact />,
      },
      {
        path: "/cart",
        element: <Cart />,
      },
      {
        path: "/restaurants/:resId",
        element: <RestaurantMenu />,
      },
    ],
    errorElement: <Error />,
  },
]);

const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(<RouterProvider router={appRouter} />);
```

2. **What is the order of life cycle method calls in Class Based Components ?**

In a React Class Component, lifecycle methods are called in the following order:

Mounting Phase:
constructor()
render()
componentDidMount()

Updating Phase:
setState(), New props, forceUpdate()
render()
componentDidUpdate()

Unmounting Phase:
componentWillUnmount()

Error Handling Phase:

static getDerivedStateFromError()
componentDidCatch()

3. **Why do we use componentDidMount?**

The componentDidMount lifecycle method in React is used for actions that need to be performed after a component has been added to the DOM (Document Object Model). It is called once, immediately after a component and all its children have been rendered to the DOM.

Here are common use cases for componentDidMount:

- **Data Fetching:**

We can initiate API calls or fetch data from a server after the component has mounted. This is a common scenario for populating a component with dynamic data.

- **Third-Party Library Integration:**

If we are integrating with third-party libraries that need access to the DOM, we can do so in componentDidMount. Examples include initializing charts, maps, or other UI components.

- **Event Listeners:**

We might want to set up event listeners after the component has mounted, especially if those events are related to the DOM.

componentDidMount is a good place to perform actions that require the component to be in the DOM, ensuring that the necessary elements are available for manipulation or interaction.

4. **Why do we use componentWillUnmount? Show with example**

The componentWillUnmount lifecycle method in React is used for cleanup or resource release before a component is removed from the DOM (Document Object Model). It is called just before a component is unmounted and destroyed.

Eg:

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

class MyComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {
      intervalId: null,
    };
  }

  componentDidMount() {
    // Start an interval when the component mounts
    const intervalId = setInterval(() => {
      console.log('Interval tick');
    }, 1000);

    this.setState({ intervalId });
  }

  componentWillUnmount() {
    // Clear the interval when the component is about to be unmounted
    clearInterval(this.state.intervalId);
  }

  render() {
    return <div>My Component</div>;
  }
}

export default MyComponent;
```

5. **Why do we use super(props) in constructor?**

In JavaScript classes, the super() method is used to call the constructor of the parent class. In React, when we define a constructor in a class component, it's necessary to call super(props) within the constructor before accessing this or this.props. This is because our component is extending the React.Component class, and calling super(props) ensures that the base class's constructor is executed, setting up the component properly.

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

class MyComponent extends Component {
  constructor(props) {
    super(props); // Call the constructor of the parent class (React.Component)
    // Other constructor logic
  }

  render() {
    // Render logic
    return <div>{this.props.someProp}</div>;
  }
}

export default MyComponent;
```

super(props) calls the constructor of the React.Component class, passing the props to it.
This is necessary because MyComponent extends React.Component, and we want to make sure the base class is properly initialized before setting up any additional logic in the derived class's constructor.
After calling super(props), we can safely access this.props and perform any other necessary setup in your component's constructor.

It's worth noting that starting with React version 16.3, we can omit the super(props) call if we don't need to access this.props in the constructor. React automatically sets this.props in the constructor if we don't explicitly call super(props). However, it's a good practice to include it for clarity and consistency.

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

- **Synchronization with Rendering:** React expects the effects and their cleanup functions to be synchronous to ensure that they don't interfere with the rendering cycle. If an effect callback is asynchronous, it might not finish before the component re-renders, potentially causing inconsistencies in the application state.

- **Cleanup Function Execution:** The cleanup function returned by useEffect should be synchronous and should handle the cleanup logic immediately. If the cleanup function is asynchronous, React may not wait for it to complete, and this can lead to problems when the component is unmounted or when the dependency array changes.

Eg:

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

function MyComponent() {
  useEffect(async () => {
    // This will lead to issues
    const result = await fetchData();
    console.log(result);
    return () => {
      // Cleanup logic (asynchronous) - may not work as expected
      cleanupAsync();
    };
  }, [/* dependency array */]);

  return <div>Component content</div>;
}

export default MyComponent;
```

In the above example, the cleanup function (cleanupAsync) is asynchronous, and it might not complete before the component is unmounted or before the next effect is executed.

If you need to perform asynchronous operations within useEffect, the recommended approach is to define an inner asynchronous function and invoke it immediately:

```jsx
import React, { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    const fetchDataAndCleanup = async () => {
      const result = await fetchData();
      console.log(result);
      // Cleanup logic (synchronous)
      cleanup();
    };

    fetchDataAndCleanup();

    // Cleanup function (synchronous) for unmounting or when dependencies change
    return () => {
      cleanup();
    };
  }, [/* dependency array */]);

  return <div>Component content</div>;
}

export default MyComponent;
```

By calling the inner asynchronous function immediately and handling cleanup synchronously, we can avoid issues related to the asynchronous nature of the cleanup function.

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

createHashRouter:
This router is useful if you are unable to configure your web server to direct all traffic to your React Router application. Instead of using normal URLs, it will use the hash (#) portion of the URL to manage the "application URL".

createMemoryRouter:
Instead of using the browser's history, a memory router manages its own history stack in memory. It's primarily useful for testing and component development tools like Storybook, but can also be used for running React Router in any non-browser environment.