**Karan Patel**

MERN Stack Assignment

|  |
| --- |
| Module 7 – Mernstack – React JS |

Lists and Keys

Question 1: How do you render a list of items in React? Why is it important to use keys when rendering lists?

Answer:

const fruits = ['Apple', 'Banana', 'Cherry']

function FruitList() {

return (

<ul>

{fruits.map((fruit, index) => (

<li key={index}>{fruit}</li>

))}

</ul>

);

}

### **Why is it important to use keys?**

* **Keys help React identify which items have changed, are added, or removed.**
* This allows **efficient updating** of the UI by avoiding unnecessary re-renders.
* Without keys, or with incorrect keys, React might **reorder or incorrectly reuse DOM elements**, leading to bugs or performance issues.

Question 2: What are keys in React, and what happens if you do not provide a unique key?

Answer:

**Keys** are **special string attributes** you need to include when creating lists of elements in React. Each key **must be unique** among siblings.

### What happens if you don’t use a unique key?

* React will **struggle to keep track** of which item corresponds to which DOM element.
* It may lead to:  
  + Incorrect item updates
  + Loss of component state (like input fields resetting)
  + Poor performance
  + React warnings in the console

Forms in React

Question 1: How do you handle forms in React? Explain the concept of controlled components.

Answer:

In React, **forms are handled using state** to keep track of input values. This approach is known as **controlled components**.

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

**function MyForm() {**

**const [name, setName] = useState('');**

**const handleSubmit = (e) => {**

**e.preventDefault();**

**alert(`Submitted name: ${name}`);**

**};**

**return (**

**<form onSubmit={handleSubmit}>**

**<input**

**type="text"**

**value={name}**

**onChange={(e) => setName(e.target.value)}**

**/>**

**<button type="submit">Submit</button>**

**</form>**

**);**

**}**

**What is a Controlled Component?**

**A controlled component is an input form element (like <input>, <textarea>, etc.) whose value is controlled by React state. The input's value is set by state, and any changes are handled by an onChange event handler that updates the state.**

Question 2: What is the difference between controlled and uncontrolled components in React?

Answer:

|  |  |  |
| --- | --- | --- |
| **Feature** | **Controlled Component** | **Uncontrolled Component** |
| **Data source** | React state | DOM itself (via refs) |
| **Updates via** | onChange handler that sets state | Direct DOM access (using ref.current.value) |
| **Use case** | When you want full control over form behavior and validation | When minimal interaction or performance matters |
| **Example input binding** | value={stateValue} | No value, just a ref |
| **Form submission** | Handled through state | Access value via ref on submit |

Uncontrolled Component Example:

import { useRef } from 'react';

function MyForm() {

const inputRef = useRef();

const handleSubmit = (e) => {

e.preventDefault();

alert(`Submitted name: ${inputRef.current.value}`);

};

return (

<form onSubmit={handleSubmit}>

<input type="text" ref={inputRef} />

<button type="submit">Submit</button>

</form>

);

}

Lifecycle Methods (Class Components)

Question 1: What are lifecycle methods in React class components? Describe the phases of a component’s lifecycle.

Answer:

#### **What are Lifecycle Methods?**

Lifecycle methods are special methods in **React class components** that are automatically called at different stages of a component's life in the DOM. They allow developers to run custom code at specific points in the **component’s lifecycle**, such as when it is created, updated, or destroyed.

### **Phases of a Component’s Lifecycle**

React class component lifecycle has **three main phases**:

#### **1. Mounting (Component is being created and inserted into the DOM)**

* constructor()
* static getDerivedStateFromProps()
* render()
* componentDidMount()

#### **2. Updating (Component is re-rendered due to changes in state or props)**

* static getDerivedStateFromProps()
* shouldComponentUpdate()
* render()
* getSnapshotBeforeUpdate()
* componentDidUpdate()

#### **3. Unmounting (Component is removed from the DOM)**

* componentWillUnmount()

Question 2: Explain the purpose of componentDidMount(), componentDidUpdate(), and componentWillUnmount()

Answer:

#### **componentDidMount()**

* **When it runs:** After the component is mounted (added to the DOM).
* **Purpose:** Ideal for:

Fetching data from an API.  
Setting up subscriptions or timers.  
DOM manipulations.

#### **componentDidUpdate(prevProps, prevState)**

* **When it runs:** After the component is updated (re-rendered).
* **Purpose:** Used to:  
  + Respond to prop or state changes.
  + Perform side-effects based on changes.

**componentWillUnmount()**

**When it runs:** Just before the component is unmounted and destroyed.

**Purpose:** Used for:

* Cleaning up timers or subscriptions.
* Removing event listeners.

Hooks (useState, useEffect, useReducer, useMemo, useRef, useCallback)

Question 1: What are React Hooks? How do useState() and useEffect() work in functional components?

Answer:

#### **✅ useState()**

* It adds **state** to functional components.

syntax : const [count, setCount] = useState(0);

#### **✅ useEffect()**

* It lets you perform **side effects** (e.g., API calls, subscriptions).
* Runs **after the component renders**.
* Syntax:

useEffect(() => {

console.log("Component mounted or updated");

}, [count]);

Question 2: What problems did Hooks solve in React development? Why are Hooks important?

Answer:

**Problems solved:**

1. **Code Reusability:** Logic was hard to reuse across components (like fetching data).
2. **Complex Components:** Class components became hard to manage with lifecycle methods.
3. **Confusing this:** Managing this in class components was error-prone.
4. **Cleaner Functional Components:** Developers preferred functional style, but needed state/lifecycle which only class components had before.

**Why Hooks are important:**

* They bring **state and side effects** to functional components.
* Enable **logic reuse** via **custom hooks**.
* Make components **simpler and cleaner**.

Question 3: What is useReducer? How do we use it in a React app?

Answer:

useReducer is an alternative to useState, useful for **complex state logic**.

const [state, dispatch] = useReducer(reducer, initialState);

const reducer = (state, action) => {

switch (action.type) {

case "increment":

return { count: state.count + 1 };

case "decrement":

return { count: state.count - 1 };

default:

return state;

}

};

const App = () => {

const [state, dispatch] = useReducer(reducer, { count: 0 });

return (

<>

<p>{state.count}</p>

<button onClick={() => dispatch({ type: "increment" })}>+</button>

<button onClick={() => dispatch({ type: "decrement" })}>-</button>

</>

);

};

Question 4: What is the purpose of useCallback & useMemo Hooks?

Answer:

|  |  |
| --- | --- |
| **Hook** | **Purpose** |
| useCallback | Memoizes a **function** to avoid re-creating it on every render |
| useMemo | Memoizes a **value/result** to avoid re-computing on every render |

Question 5: What’s the difference between useCallback & useMemo?

Answer:

|  |  |  |
| --- | --- | --- |
| **Feature** | **useCallback** | **useMemo** |
| Returns | A **memoized function** | A **memoized value/result** |
| Used for | Avoiding function recreation | Avoiding expensive calculations on every render |
| Syntax | useCallback(fn, deps) | useMemo(() => compute, deps) |

|  |  |  |
| --- | --- | --- |
| Example Use Case | Preventing unnecessary renders in child components | Caching filtered data or computed values |

Question 6: What is useRef? How does it work in a React app?

Answer:

* useRef gives you a **mutable reference** that persists across renders.
* It’s used to:  
  + **Access DOM elements**
  + **Store previous values**
  + **Avoid re-renders**

const myRef = useRef(null);

useEffect(() => {

myRef.current.focus();

}, []);

return <input ref={myRef} />;

Routing in React (React Router)

Question 1: What is React Router? How does it handle routing in single-page applications (SPAs)?

Answer:

**React Router** is a **standard library** for routing in **React** applications. It enables navigation **without reloading the page**, which is ideal for **Single Page Applications (SPAs)**.

#### **🔍 How it Works:**

* In SPAs, there is only **one HTML file (index.html)**.
* React Router updates the **URL** and renders different **components** based on the URL, **without refreshing** the page.
* It uses the **History API** to manipulate the browser's URL.

Question 2: Difference between BrowserRouter, Route, Link, and Switch

Answer:

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| **BrowserRouter** | Wraps the whole app and enables routing using browser’s History API |
| **Route** | Defines the mapping between **URL paths** and **React components** |
| **Link** | Replaces <a> tag; navigates to a different route **without page reload** |
| **Switch** (v5) | Renders the **first matched Route** inside it *(Replaced by <Routes> in v6+)* |

React – JSON-server and Firebase Real Time Database

Question 1: What do you mean by RESTful web services?

Answer:

**RESTful web services** are **APIs** that follow the **REST (Representational State Transfer)** architecture, which is:

* **Stateless**: No client context stored on the server between requests.
* **Resource-based**: Everything is treated as a resource (like /users, /products).
* **Uses HTTP methods**:  
  + GET – retrieve data
  + POST – create data
  + PUT/PATCH – update data
  + DELETE – remove data

Question 2: What is json-server? How do we use it in React?

Answer: **json-server** is a fake REST API tool for front-end development. It lets you create a full REST API with **zero coding**, just by using a JSON file.

**Install**:

npm install -g json-server

**Create a db.json** file

{

"users": [

{ "id": 1, "name": "Alice" },

{ "id": 2, "name": "Bob" }

]

}

**Run server**

json-server --watch db.json --port 3001

**In React**, access it using:

http://localhost:3001/users

Question 3: How do you fetch data from a json-server API in React? Role of fetch() or axios()

Answer:

Using Fetch()

useEffect(() => {

fetch('http://localhost:3001/users')

.then(res => res.json())

.then(data => setUsers(data))

.catch(err => console.error(err));

}, []);

Using axios:

import axios from 'axios';

useEffect(() => {

axios.get('http://localhost:3001/users')

.then(res => setUsers(res.data))

.catch(err => console.error(err));

}, []);

Question 4: What is Firebase? What features does Firebase offer?

Answer:

**Firebase** is a **Backend-as-a-Service (BaaS)** platform by Google that provides backend services for mobile and web apps.

#### **🔧 Key Features:**

* **Authentication** (Email/password, Google, Facebook, etc.)
* **Realtime Database** (live sync)
* **Firestore** (NoSQL cloud database)
* **Cloud Storage** (upload files)
* **Firebase Hosting** (deploy websites)
* **Cloud Functions** (serverless backend logic)
* **Firebase Analytics** (track usage)

#### **📦 Why Use Firebase in React?**

* Simplifies backend.
* Great for authentication and database without setting up a server.

Question 5: Importance of Handling Errors and Loading States in React APIs

Answer:

#### **⚠️ Why it's important:**

* **Without loading state**: User won’t know if data is being fetched.
* **Without error handling**: Failures (e.g., no internet) break the UI.

Example

const [data, setData] = useState([]);

const [loading, setLoading] = useState(true);

const [error, setError] = useState(null);

useEffect(() => {

fetch("http://localhost:3001/users")

.then(res => {

if (!res.ok) throw new Error("Failed to fetch");

return res.json();

})

.then(data => setData(data))

.catch(err => setError(err.message))

.finally(() => setLoading(false));

}, []);

Context API

Question 1: What is the Context API in React? How is it used to manage global state across multiple components?

Answer:

The **Context API** in React allows you to **share state or data globally** across multiple components **without using props drilling** (passing props through many levels).

#### **📌 Why Use Context API?**

* To manage **global state** like:  
  + User authentication
  + Theme (light/dark)
  + Language settings
  + Shopping cart data

How Context Works:

**Create a context**

**Provide** the context value at a high level in the component tree

**Consume** the context value in child components

Example Use Case (Global Theme):

import { createContext } from 'react';

export const ThemeContext = createContext();

// App.js

import { ThemeContext } from './ThemeContext';

import Child from './Child';

function App() {

const theme = "dark"; // Global state

return (

<ThemeContext.Provider value={theme}>

<Child />

</ThemeContext.Provider>

);

}

// Child.js

import { useContext } from 'react';

import { ThemeContext } from './ThemeContext';

function Child() {

const theme = useContext(ThemeContext);

return <h1>Current Theme: {theme}</h1>;

}

Question 2: How do createContext() and useContext() work in React for sharing state. ?

Answer:

|  |  |
| --- | --- |
| **Hook / Function** | **Purpose** |
| createContext() | Creates a Context object |
| useContext() | Lets you read the context value in a component |

**Create Context**

import { createContext } from 'react';

export const MyContext = createContext();

**Provide the State**

<MyContext.Provider value={sharedData}>

<App />

</MyContext.Provider>

**Consume the State**

import { useContext } from 'react';

import { MyContext } from './MyContext';

const Child = () => {

const value = useContext(MyContext);

return <p>{value}</p>;

};

**Full Example: Share a count state**

// CounterContext.js

import { createContext } from 'react';

export const CounterContext = createContext();

**// App.js**

import React, { useState } from 'react';

import { CounterContext } from './CounterContext';

import Child from './Child';

function App() {

const [count, setCount] = useState(0);

return (

<CounterContext.Provider value={{ count, setCount }}>

<Child />

</CounterContext.Provider>

);

}

**// Child.js**

import React, { useContext } from 'react';

import { CounterContext } from './CounterContext';

function Child() {

const { count, setCount } = useContext(CounterContext);

return (

<div>

<h2>Count: {count}</h2>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

**State Management (Redux, Redux-Toolkit or Recoil)**

Question 1: What is Redux, and why is it used in React applications?

Answer:

**Redux** is a **predictable state container** for JavaScript apps. It is commonly used with **React** to manage the global state of an application in a centralized and consistent way.

#### **🚀 Why use Redux in React?**

* To manage **complex state** shared across many components.
* To make state **predictable and testable**.
* To provide a **single source of truth** (the store).
* To enable **debugging and time-travel** with dev tools.

**Core Concepts of Redux**

#### **1. Actions**

* Plain JavaScript objects that describe what happened.
* They must have a type field (string).

const addTodo = {

type: "ADD\_TODO",

payload: { text: "Learn Redux" }

};

**Reducers**

Pure functions that take current state and an action, and return a new state.

They do not mutate the original state.

**Store**

The central container holding the application state.

Created using createStore(reducer).

Question 2: How does Recoil simplify state management in React compared to Redux?

Answer:

**Recoil** is a **state management library** developed by Facebook, designed specifically for **React**. It provides a simpler and more **React-like approach** compared to Redux.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Recoil** | **Redux** |
| Boilerplate | Minimal | Verbose |
| Learning Curve | Easy | Steeper |
| State Access | Directly in components | Through connect or hooks |
| Local & Global State | Both supported | Mostly global |
| Setup | No need for extra files or config | Requires actions, reducers, store |

**Key Concepts in Recoil**

#### **Atoms**

* Units of state.
* Any component can read/write them.

**Selectors**

Derived/computed state.

Like derived values or computed props.

**useRecoilState / useRecoilValue**

const [count, setCount] = useRecoilState(countState);

const double = useRecoilValue(doubleCount);

|  |  |  |
| --- | --- | --- |
| **Feature** | **Redux** | **Recoil** |
| Setup | Complex | Simple |
| Concepts | Actions, Reducers, Store | Atoms, Selectors |
| Code Size | More | Less |
| React Integration | External logic | Fully React-native |