Props and State :-

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

- **Props (Properties)** are used to pass data from a parent component to a child component.
- They are **read-only** and **immutable**, meaning they cannot be modified by the child component.
- Props allow components to be **reusable** and **customizable** by accepting dynamic values.
- **Difference from State:**
    - **Props:** Passed from parent to child, cannot be modified by the receiving component.
    - **State:** Managed within the component itself, can be updated using setState in class components or the useState hook in functional components.

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

- State is an **internal data storage** mechanism for React components.
- It holds dynamic data that can change over time, influencing how the component renders.
- State is **mutable** and can be updated using setState() in class components or useState() in functional components.
- State changes trigger **re-renders**, ensuring the UI stays in sync with the data.

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

- this.setState() is used to **update the component's state** in class components.
- It **merges** the updated state with the current state, ensuring only the specified properties are modified.
- It is **asynchronous**, meaning the state might not update immediately.
- It triggers a **re-render** of the component to reflect the updated state in the UI.

Handling Events in React :-

Question 1: How are events handled in React compared to vanilla JavaScript? Explain the concept of synthetic events.

- **Event Handling in React:**
  - React uses **camelCase** for event names (e.g., onClick vs. onclick in vanilla JS).
  - Event handlers in React are **functions** passed as props, not string attributes.
  - React **automatically binds** the this context for function components, but not for class components.
- **Synthetic Events:**
  - React wraps native browser events in a **SyntheticEvent** wrapper for **cross-browser compatibility**.
  - It **normalizes** events to ensure consistent behavior across different browsers.
  - Synthetic events are **pooled** for performance, meaning their properties may be **reset** after the event handler completes.

Question 2: What are some common event handlers in React.js? Provide examples of onClick, onChange, and onSubmit.

```
function ClickExample() {
  function handleClick() {
    alert("Button clicked!");
  }
  return <button onClick={handleClick}>Click Me</button>;
}
```

onChange: Triggers when the value of an input field changes.

```
function ChangeExample() {
  function handleChange(event) {
    console.log("Input changed to:", event.target.value);
  }
  return <input type="text" onChange={handleChange} />;
}
```

onSubmit: Triggers when a form is submitted.

```
function SubmitExample() {
  function handleSubmit(event) {
```

```
    event.preventDefault();
    alert("Form submitted!");
  }
  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

Question 3: Why do you need to bind event handlers in class components?

- **Binding is necessary** because in class components, the this keyword is not automatically bound to the instance.
- Without binding, **this** inside the event handler will be **undefined**.
- Binding can be done using:
  - The **constructor** (e.g., this.handleClick = this.handleClick.bind(this);)
  - The **arrow function** syntax (e.g., onClick={() => this.handleClick()})
  - **Class property syntax** with arrow functions (e.g., handleClick = () => {...}).

Conditional Rendering :-

Question 1: What is conditional rendering in React? How can you conditionally render elements in a React component?

- **Conditional Rendering** allows you to **dynamically** decide what to display based on the component's **state** or **props**.
- It works like conditional statements in JavaScript, determining which **elements**, **components**, or **content** to render.
- Common methods for conditional rendering in React include:
  - **if-else statements**
  - **Ternary operators**
  - **Logical AND (&&) operators**
  - **Conditional functions or helper methods**

Question 2: Explain how if-else, ternary operators, and && (logical AND) are used in JSX for conditional rendering.

1. **if-else Statement (Outside JSX)**

```
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h1>Welcome back!</h1>;
  } else {
    return <h1>Please log in.</h1>;
  }
}
```

2. **Ternary Operator (Inline in JSX)**

```
function Greeting({ isLoggedIn }) {
  return (
    <h1>{isLoggedIn ? "Welcome back!" : "Please log in."}</h1>
  );
}
```

3. **Logical AND (&&) Operator (For Simple Checks)**

```
function Notification({ hasMessages }) {
  return (
    <div>
      <h1>Dashboard</h1>
      {hasMessages && <p>You have new messages.</p>}
    </div>
```

```
  );
}.
```

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?

- **Rendering a List:**
  - Use the **.map()** method to iterate over an array of items and return a **React element** for each item.
  - The **.map()** function helps dynamically generate lists based on data.

```
function ItemList() {
  const items = ["Apple", "Banana", "Cherry"];
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
}
```

- **Importance of Keys:**
  - Keys **uniquely identify** each item in the list.
  - They **help React optimize** rendering by identifying which items **changed, added, or removed**.
  - Without keys, React might **re-render** the entire list unnecessarily, reducing performance.

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

- **Keys:**
  - Keys are **unique identifiers** for each item in a list.
  - They **associate** a DOM element with the corresponding **data item**.
- **Without Unique Keys:**
  - React cannot track **reordering** or **deletion** of elements accurately.
  - This can lead to **unexpected behavior**, such as **incorrect updates** or **element reuse**.
  - It may also result in **performance issues** due to inefficient DOM updates.

Forms in React :-

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

- **Handling Forms in React:**
  - Forms in React are managed by using **state** to control the values of form elements (like input, textarea, and select).
  - A form element is considered a **controlled component** if its value is controlled by React state.
- **Controlled Components:**
  - A **controlled component** is a form element where the value of the input is bound to the component's state.
  - Any changes to the input value trigger the **state update**, and the UI re-renders based on the new state.
  - You must define **event handlers** (e.g., onChange) to update the state when the input changes.

```
function FormExample() {

  const [inputValue, setInputValue] = useState("");



  const handleChange = (event) => {

    setInputValue(event.target.value); // Update state on input change

  };



  const handleSubmit = (event) => {
```

```
    event.preventDefault();
    alert("Input Submitted: " + inputValue);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={inputValue} // Controlled by state
        onChange={handleChange} // Update state on change
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

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

- **Controlled Components:**
  - The **value** of the input field is controlled by the **state** of the component.
  - The input's value is passed through the value attribute, and updates are handled through **state changes**.
  - React is in full control of the form data.

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

- **Controlled Components:**
  - The **value** of the input field is controlled by the **state** of the component.
  - The input's value is passed through the value attribute, and updates are handled through **state changes**.

- React is in full control of the form data.

```
<input type="text" value={inputValue} onChange={handleChange} />
```

Uncontrolled Components:

- The input's value is managed by the **DOM** itself, not by React's state.
- You access the input value using a **ref** (short for reference) instead of the value attribute.
- React has less control over the form data, which can be useful in some cases where you don’t need full control over the input.

```
function FormExample() {
  const inputRef = useRef();

  const handleSubmit = (event) => {
    event.preventDefault();
    alert("Input Submitted: " + inputRef.current.value); // Access value via ref
  };

  return (
    <form onSubmit={handleSubmit}>
      <input ref={inputRef} type="text" />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Key Differences:

- **State Management:** Controlled components use React state for the value; uncontrolled components use the DOM.

- **Ref Usage:** Uncontrolled components require the ref API to access input values.
- **Rendering:** Controlled components cause re-renders with state updates; uncontrolled components do not trigger re-renders on input value changes.

Lifecycle Methods (Class Components) :-

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

- **Lifecycle Methods** in React class components allow you to hook into different stages of a component's life — from creation to removal. These methods are useful for performing tasks like data fetching, subscriptions, and clean-up.
- **Phases of a Component's Lifecycle:**
    1. **Mounting (Creation Phase):**
        - This phase occurs when the component is being created and inserted into the DOM.
        - Methods:
            - constructor(): Initializes the component and its state.
            - static getDerivedStateFromProps(): Used to update state based on props.
            - render(): Renders the component's JSX.
            - componentDidMount(): Called after the component is mounted (useful for data fetching, subscriptions).
    2. **Updating (State or Props Change Phase):**
        - This phase occurs when a component's state or props change.
        - Methods:
            - static getDerivedStateFromProps(): Called before every render, allows state to be updated based on props.
            - shouldComponentUpdate(): Determines whether the component should re-render.
            - render(): Re-renders the component based on the updated state or props.
            - getSnapshotBeforeUpdate(): Captures information about the DOM before it's updated.
            - componentDidUpdate(): Called after the component updates (useful for side effects or updating DOM).
    3. **Unmounting (Removal Phase):**
        - This phase occurs when the component is being removed from the DOM.
        - Methods:
            - componentWillUnmount(): Used for cleanup tasks like canceling subscriptions or clearing timers.

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

1. **componentDidMount():**
    - **Purpose:** This method is called **once** immediately after a component is **mounted** (added to the DOM).
    - It's commonly used for tasks like **data fetching**, **setting up subscriptions**, or performing any **one-time setup**.
    - componentDidMount() {
    - fetchData().then(data => this.setState({ data }));
    - }

componentDidUpdate():

- **Purpose:** This method is called **after** a component is **updated** due to changes in **state** or **props**. It provides the previous props and state as arguments, allowing you to compare the current and previous states.
- It is useful for **performing side effects** based on prop or state changes, like sending network requests or updating the DOM.

```
componentDidUpdate(prevProps, prevState) {

  if (this.state.count !== prevState.count) {

    console.log("Count updated!");

  }

}
```

componentWillUnmount():

- **Purpose:** This method is called **right before** a component is **removed** from the DOM. It is primarily used for cleanup tasks like **clearing timers**, **canceling network requests**, or **unsubscribing** from events or external services.

```
componentWillUnmount() {

  clearInterval(this.timerID); // Stop any ongoing timer

}
```

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

- **React Hooks:**
  - React hooks are **functions** that let you "hook into" React state and lifecycle features from **functional components**, without needing to convert them to class components.
- **useState() Hook:**
  - It lets you add state to **functional components**. You can declare a state variable and a function to update it.

```
const [state, setState] = useState(initialValue);

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

setCount(count + 1); // Update state
```

useEffect() Hook:

- It allows you to perform side effects (like fetching data or subscribing to events) in a **functional component**. It's similar to lifecycle methods in class components (e.g., componentDidMount, componentDidUpdate).

```
useEffect(() => {

  // Code to run on component mount or state/prop changes

}, [dependencies]);

useEffect(() => {

  document.title = `You clicked ${count} times`;

}, [count]);
```

Question 2: What problems did hooks solve in React development? Why are hooks considered an important addition to React?

- **Problems Solved:**
  - **State in Functional Components:** Before hooks, only class components could manage state. useState() solved this by allowing functional components to use state.

    - **Handling Side Effects:** useEffect() allowed functional components to handle side effects, replacing lifecycle methods (componentDidMount, componentDidUpdate, etc.) in class components.
    - **Code Reusability:** Hooks provide **custom hooks** that allow you to extract and reuse logic, making code more modular and reusable.
    - **Cleaner Code:** Hooks eliminate the need for **complex class structures**, leading to simpler and more concise components.
- **Importance of Hooks:**
    - They enable **functional components** to manage state and side effects, making them more powerful and reducing the boilerplate code of class components.
    - They improve **code readability** and **composability**, making React development more intuitive and flexible.

    - **Handling Side Effects:** useEffect() allowed functional components to handle side effects, replacing lifecycle methods (componentDidMount, componentDidUpdate, etc.) in class components.
    - **Code Reusability:** Hooks provide **custom hooks** that allow you to extract and reuse logic, making code more modular and reusable.
    - **Cleaner Code:** Hooks eliminate the need for **complex class structures**, leading to simpler and more concise components.
- **Importance of Hooks:**
    - They enable **functional components** to manage state and side effects, making them more powerful and reducing the boilerplate code of class components.
    - They improve **code readability** and **composability**, making React development more intuitive and flexible.

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

- **useReducer**:
    - It is a hook used for managing more complex state logic in a component. It's an alternative to useState and is preferred when you need to manage multiple state values or perform more complex state transitions (similar to Redux).

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

const initialState = { count: 0 };

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

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

- **useCallback**:
  - The useCallback hook **memoizes** a function, preventing it from being recreated on every render. It's useful when passing callbacks to child components to avoid unnecessary re-renders.

    ```
    const memoizedCallback = useCallback(() => { /* function body */ }, [dependencies]);
    ```

- **useMemo**:

  The useMemo hook **memoizes** a **computed value** (like a result of a calculation), so that it is recomputed only when the dependencies change. It optimizes performance by avoiding expensive recalculations.

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Question 5: What's the Difference between useCallback & useMemo Hooks?

- **useCallback:**
  - It **memoizes** a **function**, so the same function is used across renders unless dependencies change.
- **useMemo:**
  - It **memoizes** the **result of a computation**, and recomputes the value only when the dependencies change.
- **Key Difference:**

- useCallback is used to **memoize functions**, while useMemo is used to **memoize values** or **results of computations**.

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

- **useRef**:
  - The useRef hook returns a **mutable object** (ref.current) which persists across renders. It is commonly used to access **DOM elements** directly or to store values that do not trigger a re-render when changed.
- **Usage in React app:**
  - To reference a DOM element:

```
const inputRef = useRef();

useEffect(() => {

  inputRef.current.focus();

}, []);

function FocusInput() {

  const inputRef = useRef();

  return <input ref={inputRef} />;

}
```

Routing :-

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

- **React Router** is a library for handling **client-side routing** in React apps, allowing navigation without full page reloads.
- It uses the **HTML5 history API** to manage **URL changes** and render the appropriate **components** based on the current path.

Question 2: Difference between BrowserRouter, Route, Link, and Switch components in React Router.

- **BrowserRouter:** Wraps the app for routing, using the **HTML5 history API**.
- **Route:** Matches a **URL** path to a specific **component**.

- **Link:** Provides **client-side** navigation without full page reloads.
- **Switch (v5) / Routes (v6):** Renders the **first matching** route.

<u>React – JSON-server and Firebase Real Time Database :-</u>

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

- RESTful web services are **APIs** that follow the **REST (Representational State Transfer)** architecture, using **HTTP** methods (GET, POST, PUT, DELETE) for communication and typically return **JSON** or **XML** data.

Question 2: What is Json-Server? How do we use it in React?

- **Json-Server** is a **mock REST API** that allows you to **quickly create** a **backend** for testing and prototyping.
- **Usage in React:**
  - Install with: npm install json-server
  - Create a **db.json** file with mock data.
  - Run the server: json-server --watch db.json --port 3001
  - Use **fetch** or **axios** to make API requests to http://localhost:3001.

Question 3: How do you fetch data from a Json-server API in React?

- Use **fetch()** or **axios()** to make API requests:

```
useEffect(() => {

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

    .then(response => response.json())

    .then(data => setPosts(data))

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

}, []);
```

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

- **Firebase** is a **backend-as-a-service** (BaaS) platform by **Google** for building web and mobile apps.
- **Features:**

- Authentication
- Realtime Database
- Cloud Firestore
- Cloud Storage
- Hosting
- Cloud Functions
- Analytics and Messaging

Question 5: Discuss the importance of handling errors and loading states when working with APIs in React.

- **Importance:**
  - Provides a **better user experience**.
  - Prevents **UI blocking** during **data fetching**.
  - **Error handling** helps show **meaningful messages** to users.

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

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



useEffect(() => {

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

    .then(response => response.json())

    .then(data => setPosts(data))

    .catch(error => setError(error))

    .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?

- Context API is a React feature for managing global state without prop drilling.
- It allows data to be shared across multiple components without explicitly passing props at every level.
- Usage:

- Create a Context: const MyContext = React.createContext();
- Provide the Context: Wrap components in a Provider**.**
- Consume the Context: Use useContext() or Consumer to access the state.

Question 2: How are createContext() and useContext() used in React for sharing state?

- **createContext()**:
  - Creates a Context object for holding state.
  - Provides a Provider component to share the state.

```
const MyContext = createContext();

const value = useContext(MyContext);
```

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

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

- **Redux** is a **state management** library for **JavaScript** applications, commonly used with **React** to **centralize** and **manage** global state.
- **Why Use Redux:**
  - Centralizes **state** management.
  - Simplifies **state** sharing across components.
  - Provides **predictable** state updates.
  - Improves **debugging** with tools like **Redux DevTools**.

Core Concepts:

- **Actions:** Plain **JavaScript objects** that describe **what** happened.

```
{ type: 'INCREMENT', payload: 1 }
```

- **Reducers:** Pure functions that specify **how** the state changes in response to **actions**.

```
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + action.payload;
    case 'DECREMENT':
```

```
            return state - action.payload;

        default:

            return state;

    }

}
```

Store: Holds the **application state**, **dispatches** actions, and **subscribes** to changes.

```
const store = createStore(counter);
```

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

- **Recoil** is a **state management** library for React that simplifies managing **shared** and **derived** state.
- **Simplifies State Management:**
    - **No Boilerplate:** Less setup compared to **Redux**.
    - **Direct State Access:** Uses **atoms** for state and **selectors** for derived state.
    - **Reactivity:** Automatically updates components when the **state** changes.
    - **Concurrency Support:** Handles complex **async** and **synchronous** state logic.