1. Is JSX mandatory for React?

No, JSX is not mandatory for React. JSX is a syntactic sugar for the React.createElement() method, which is used to create React elements. While JSX offers a more concise and readable syntax that resembles HTML, it is ultimately compiled into React.createElement() calls under the hood.

JSX is **not part of React itself**, but a convenience provided by build tools like Babel. Both JSX and React.createElement() return the same type of object: a plain JavaScript object with a $$typeof property (indicating it's a React element) and a type that specifies the HTML tag or component being created.

So while you can build React apps without JSX, using JSX can significantly improve code readability and developer experience — especially in complex UIs.

2. is ES6 mandatory for React?

No, ES6 is not mandatory for React. React can be written using plain JavaScript (ES5), but ES6 (ECMAScript 2015) and newer versions provide modern syntax and features — such as classes, arrow functions, destructuring, template literals, and let/const — which make React code more concise, readable, and maintainable.

React itself does not require ES6, but most modern React codebases use it for better structure and developer productivity. When ES6+ syntax is used, build tools and transpilers like Babel convert the code into backward-compatible JavaScript (usually ES5) so that it can run in all browsers. So, while not mandatory, ES6 is strongly recommended for a smoother React development experience.

3. {TitleComponent} vs {< TitleComponent/>} vs {< TitleComponent></ TitleComponent>} in JSX

{TitleComponent}

This refers to the component function itself, not its rendered output. Example usage: passing the component as a prop:

<WrapperComponent Component={TitleComponent} />

You're treating TitleComponent as a reference to a function, not calling it.

{<TitleComponent />}

This is the JSX syntax to render the component. It invokes the component and returns its rendered output (i.e., the React element). Commonly used in JSX:

{<TitleComponent />}

{<TitleComponent></TitleComponent>} This is functionally the same as {<TitleComponent />}.

The difference is syntactic: the first is a self-closing tag, and the second is an explicit open/close tag.

Use the open/close form when you want to pass children to the component:

<TitleComponent>Some child content</TitleComponent>

4. How can I write comments in JSX?

Comments in JSX can be written by including them within {*//*}

5. What is difference between <React.Fragment></React.Fragment> and <></>?

They are functionally the same thing and perform the same function at the end of the day.

6. What is Virtual DOM?

The **Virtual DOM** is a **lightweight copy** of the real DOM used by React to improve performance.

Instead of updating the real DOM directly every time something changes, React:

- **Creates a Virtual DOM** — a JavaScript object representing the current UI.
- **Makes changes** to the Virtual DOM first.
- **Compares** the new Virtual DOM with the previous one using a process called **"diffing."**
- **Updates only the changed parts** in the real DOM — making it fast and efficient.

Why it's useful:

Real DOM updates are slow.

Virtual DOM reduces the number of direct DOM manipulations.

Leads to faster UI rendering and better performance.

Example: Virtual DOM representation of this JSX

jsx Copy Edit

```jsx
const element = (
  <div className="container">
    <h1>Hello, React!</h1>
    <p>This is a virtual DOM example.</p>
  </div>
);
```

How the Virtual DOM might look (simplified)

js Copy Edit

```js
const virtualDom = {
  type: "div",
  props: {
    className: "container",
    children: [
      {
        type: "h1",
        props: {
          children: "Hello, React!"
        }
      },
      {
        type: "p",
        props: {
          children: "This is a virtual DOM example."
        }
      }
    ]
  }
};
```

7. What is Reconciliation in React

Reconciliation is the process React uses to update the user interface efficiently when component state or props change. When a change occurs, React creates a new Virtual DOM tree and compares it with the previous one. This comparison, called diffing, helps React identify which parts of the UI have changed.

Based on this comparison, React calculates the minimum number of changes needed and then updates only those parts in the real DOM. This process ensures that React performs the least amount of manipulation on the actual DOM, resulting in better performance and a smoother user experience.

In summary, reconciliation allows React to update the UI efficiently by applying only necessary changes to the DOM after comparing Virtual DOM trees.

8. Why we need keys in React? When do we need keys in React?

Keys are required in React when rendering a list or group of elements, such as an array of objects. Keys help React uniquely identify each item in the list, enabling it to track which items have changed, been added, or removed. Without keys, React would have to re-render the entire list every time there is a change.

However, when keys are provided, React can efficiently update only the changed elements by comparing the keys during the reconciliation process using the Virtual DOM. This optimization improves rendering performance and ensures a smoother user experience.

Therefore, keys are necessary whenever rendering lists of elements to help React manage updates efficiently.

9. Can we use index as keys in React?

Technically, it is not wrong to use array indices as keys in React. However, it is generally discouraged, especially in cases where the list items can be **reordered, added, or removed**.

When you use indices as keys, and the order of elements changes, the indices also change. This can confuse React during the reconciliation process, causing it to incorrectly associate data with DOM elements. As a result, the UI may display incorrect or inconsistent data.

To avoid such issues, it's better to use a **stable and unique identifier**, such as an id, that doesn't change between renders. This ensures that React can correctly track each element and update the DOM efficiently and accurately.

10. What are props in React ?

Props (short for **properties**) are a way to pass **data from one component to another** in React, typically from a **parent component to a child component**. They allow components to be **reusable and dynamic** by providing input values that a component can use to render customized content.

Props are **read-only**, meaning a component cannot change the props it receives. Instead, any data updates must happen in the parent component, which then passes new props down to its children.

11. What is config drivern UI?

A **Config Driven UI** is a user interface that is **built dynamically using configuration data**, instead of hardcoding each UI element manually. In this approach, a JSON or JavaScript object (the "config") defines what UI components to render, their types, labels, validation rules, styles, etc.

React components then **read this config and generate the UI automatically** based on its structure. This makes the UI highly flexible, reusable, and easier to update or scale without modifying the core codebase.