CHAPTER 04
TALK IS CHEAP, SHOW ME THE CODE

1. **Is JSX mandatory for React?**

   No, JSX is not mandatory for React. We can use React without JSX. Using React without JSX can be preferred if you're not doing any compilation in build environment.

   Moreover, JSX is the syntactical sugar for React.createElement(component, props, children). So, anything which you're doing with JSX can be done with just plain JavaScript.

   If you want to prefer React without JSX, then go on to add a shorthand like

   const k = React.createElement and process with the code. This serves better than to keep on repeating the same.

   👉 https://reactjs.org/docs/react-without-jsx.html#:~:text=JSX%20is%20not%20a%20requirement,syntactic%20sugar%20for%20calling%20React.

2. **Is ES6 mandatory for React?**

   No, ES6 is not mandatory for React. We can use React without ES6 too.

   TO use react without ES6, we need to install a node module create-react-class.
   In your React application, open the terminal and run the command

   npm i create-react-class

   Now we're ready for implementation without ES6

   👉 https://www.codingninjas.com/codestudio/library/react-without-es6
   👉 https://medium.com/@soni.dumitru/react-without-es6-794ed21e12ed
   👉 https://reactjs.org/docs/react-without-es6.html

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

   {} play major role in the JSX. For calling Component, to use React Element inside React Component, to use functional component inside React Component etc.

   {TitleComponent}

   This syntax is used to include a JSX variable or expression or element in the React Component. The given element in {} will be evaluated in JSX Element.

Along with that we can use {} to write plain/any JavaScript inside JSX, and to call functions in it.

{< TitleComponent/>}

We can use {<TitleComp/>} when Functional component is used inside a React component. To use the functional component in React Component we use either **<Title/> or {<Title />}.**

At the end functional component is a function. So, **{Title ()} too shall work.**

Whenever we're referring to functional component/Class Based/Any Component we shall use the syntax of JSX <JSX/>.

Direct Functions are not executed as React Child. So, we've to use {Title ()} / <Title/> / {<Title/>}.

{< TitleComponent> </TitleComponent>}

This syntax is used to create a JSX element that has as starting and ending tag. The content between the starting and ending tags will be included in the JSX element.

4. **How can I write comments in JSX?**
   Comments can be written in JSX just like we write them in JavaScript but wrapped inside the curly braces {}.

   In JS - /* This is a comment */
   In JSX- {/* This is a comment*/

   Not only comments but we can write any JavaScript code inside the {} in JSX.

5. **What is <React. Fragment></React. Fragment> and <> </>?**

   In JSX it isn't allowed to write two parent/root Elements. Any piece of JSX component we write, there can only be one parent element i.e JSX can have only one Root Element.

   Ex- Const Fan = <h1>JSX</h1> <h1>SecondJSX</h1>

   This is **invalid** JSX. Bz there can only be one parent.

   We can write this with div ..

   Const Fan = (

   ** **

   <h1>JSX</h1>

   <h1>SecondJSX</h1>

   ** **

   );

Const root = ReactDOM.createRoot(document.getElementbyId("root"));

root.render(Fan);

This is valid JSX. But the only problem is along with our root id div, this JSX div too displayed inside the DOM which makes it lengthy and complex with 2 div's inside the DOM.

So, to overcome this and not to make our DOM cache we can use **React.Fragment.**

React.Fragmet is a component which is Exported by React. (which is in JS file import react from "react").

We can use React.Fragment as 👇 👇

Const Fan = (

<React.Fragment>

<h1>JSX</h1>

<h1>Second JSX</h1>

</React.Fragment>

);

Now we'll get not extra adds other than the value in our DOM.

But why this <React.Fragment> …….. </React.Fragment> is not reflecting in DOM?

Because this <React.Fragment**>** is an EMPTY TAG. Anything written in between this tag is displayed but not the tag itself. Rather than always writing <React.Fragment>, we can just write **<>… … … </>. This is the shorthand for <React.Fragment>.**

Const Fan = (

< >

<h1>JSX</h1>

<h1>Second JSX</h1>

</>

);

So, rather than using div which is extra bit of code. We can use <React.Fragment></React.Fragment> /// <></>. This means it won't show up in Root/DOM and acts as one parent inside JSX.

6. **What is Virtual DOM?**
   It is Popularly stated that React uses Virtual DOM. And most of us believe that react is faster just because it has virtual DOM which is not true as we've seen earlier that many dependencies and packages and libraries and virtual DOM collectively make REACT most powerful and faster.

Virtual DOM is not only the concept in React. It's used in multiple places in Software Engineering. React is just one of the multiples.
React uses Virtual DOM. Virtual DOM is a representation of our Actual DOM. Virtual DOM is not the ACTUAL DOM. It is just a representation of our Actual DOM. We need Virtual DOM for something known as RECONCILIATION in react.

7. **What is Reconciliation in React?**

Reconciliation in React is a process where React uses Diffing Algorithm to find the difference between the updated Tree and Actual Tree. When the difference or update is found in the Application, that portion in the application which was wanted to be get update only that portion going to re-render without disturbing the other elements by Diffing Algorithm. This whole process is called RECONCILIATION.
And this Diffing Algorithm happens on Browser. This is just like Diffing in GIT.

Check this too 👉 https://reactjs.org/docs/reconciliation.html

8. **What is React Fiber?**

In the latest version of React i.e. In React 18**,** the creators have introduced a new way of finding Diffing Algorithm and that is called React Fiber. React Fiber is the new Reconciliation Engine.
A fiber generally represents a unit of work. The Primary goal of React Fiber is to take advantage of Scheduling. Scheduling is the process of telling when the work should be done/performed.

React currently doesn't take advantage of scheduling completely. When an update is done, it results in the re-render of entire sub-tree. Remodeling React core algorithm to take advantage of scheduling is the main aspect of React FIBER.

With scheduling React Fiber specifically must be able to do the following 👇 👇

- 👉 Pause Work and Come back to it later
- 👉 Assign priority to different types of work
- 👉 Reuse previously completed work
- 👉 Abort work if it's no longer needed

In order to do any of this, we need a way to break work down into units. In one way, this is what **FIBER** is. FIBER represents units of work…

For more detail check out 👇 👇 👇
- 👉 https://github.com/acdlite/react-fiber-architecture
- 👉 https://www.simplilearn.com/tutorials/reactjs-tutorial/reactjs-state

9. **Why do we need keys in React? When do we need keys in React?**

   Let's check it out with a subtle example.

   Suppose we're having 3 div's (div 1, div2, div3). Now, we've added a new div 4 to the existing div (sub-root). We, think React is very fast and so, it is too proficient in all aspects but in this circumstance React is not a pro chunk. Infact React is somewhat oblivion. React doesn't know where the newly added "div" is present in the sub-root. So, it re-renders the whole node which is not we've opted/expected from react. So, to adjust this and to re-render only the updated/added portion separately without touching/re-rendering everything we use "**KEYS" in REACT.**
   If each div is provided with a unique key, react can easily find the position of the element and re-render only that portion which is always the better way to go with. So, always in case of **multiple** same tags provide keys. But, when there're multiple unique tags/elements we needn't to provide keys as the tag itself represents its position. But when we're using **MULTIPLE SAME TAGS, WE MUST NEED TO GIVE KEYS** which will help react a lot.

   👉 https://stackoverflow.com/questions/59517962/react-using-index-as-key-for-items-in-the-list

10. **Can we use index as keys in React?**

    Yes, we can use index as keys in React. But this is not highly recommended to use. Using index as keys in react can negatively impact performance and may cause issues with component state. If you choose not to assign an explicit key, then react will default using indexes as keys.

    Get into it 👇👇👇
    👉 https://medium.com/geekculture/reactjs-why-index-as-a-key-is-an-anti-pattern-4b9dc6ef0067
    👉 https://reactjs.org/docs/reconciliation.html#elements-of-different-types (Elements of different types)
    👉 https://robinpokorny.com/blog/index-as-a-key-is-an-anti-pattern/

11. **What is props in React?**

    Props are nothing but the **properties**. Props are an **Object**. We know Parameters and Arguments from JavaScript. During the Call, we say we're passing arguments as parameters into the function. In the same way in react, whenever we say passing props to your component that mean you're passing some data into your component. We can pass multiple props to your component. And props can be anything.

    Literally, components are like JavaScript functions. They accept arbitrary inputs (called props) and return react elements describing what should appear on the screen.

    All react components must act like pure functions with respect to their props**. Props are read-only**. They shouldn't get modified whether you declare a component as a function or a class, it must **never modify its own props**. Such functions are called "pure" because they don't attempt to change their inputs and always return the same result for the same inputs.

As application UIs are dynamic and change over time. Here comes the **State** which allows react components to change their output over time in response to user actions, network responses, and anything else, without violating props rule.

👇👇👇

https://reactjs.org/docs/components-and-props.html

https://reactjs.org/docs/state-and-lifecycle.html

12. What is a Config Driven UI?

Breaking the traditional way of building applications there's a different way of development proposed to enlighten the clients.

The seed started with the questions-

👉 What if you build applications that were dynamically composed and based on configuration?
👉What if you built an application that could easily adapt with just a few changes in this configuration?

TRADITIONALLY BUILD APPLICATIONS	WITH CONFIG BUILD APPLICATIONS
• Lead architects design around business requirements • Application is built and deployed • Changes are done through additional components (SOLID principles) or painful refactor.	• Independent components are built first, starting at the atomic level. • An interface (usually JSON) is defined to compose the higher-level UI. • Combination of reusable components and JSON blueprints allows developers to easily build up and out

We can check more of it from 👇 👇 👇 👇 👇 👇

👉 https://medium.com/captech-corner/an-intro-to-configuration-driven-development-cdd-48a1c088baa9

👉 https://github.com/andrewevans0102/sandbar2

👉 https://iamrajatsingh1.medium.com/config-driven-ui-c8e93b730993

By Basic def

When we build a real-world huge application, we want our website to be working in all the states. We can't have separate website for each state. So, we control our website using Configuration. Big companies (Zomato, Swiggy, amazon etc.) use config driven UI. All the UI on the website is driven by a config sent by BACKEND.

Yes, Backend drives this config.

Backend controls what type of website should be in different cities, what offers should be running in different cities etc. Everything is controlled by Backend.

Everything is driven by this Configuration. That's why most of the websites at present are stated as Config Driven UI.