**Assignment 4 – Talk is cheap, show me the code**

Q1: Is JSX mandatory for React?

The syntax of React.createElement(type, props, …children) is not developer friendly when it comes to nesting tags. JSX or javascript XML is an extension of Javascript that lets you write HTML-like markup inside a Javascript file.

Ex: const element = <p>Learning React – Assignment 3</p>

These elements are rendered as React Elements. Each JSX element is the syntactic sugar for calling React.createElement() itself. So, anything that you do in JSX can be done in plain Javascript as well. Hence, JSX is not mandatory, but it is highly recommended to use it as it helps in writing better and understandable code.

Q2: Is ES6 mandatory for React?

ECMAScript6(or ES6) is not mandatory for React, but it is highly recommended and widely used in React Development. One has to be familiar with the ES6 features like arrow functions, classes, modules, template literals, Spread operator to name a few.

Q3: {TitleComponent} vs {<TitleComponent/>} vs {< TitleComponent >< /TitleComponent >} in JSX

These are all different ways of rendering a component.

• {TitleComponent}: This value describes the TitleComponent as a javascript expression or a variable. The {} can embed a javascript expression or a variable inside it.

• <TitleComponent/> : This value represents a Component that is basically returning Some JSX value. In simple terms TitleComponent is a function that is returning a JSX value. A component is written inside the {< />} expression.

• <TitleComponent></TitleComponent> : <TitleComponent /> and <TitleComponent></TitleComponent> are equivalent only when < TitleComponent /> has no child components. The opening and closing tags are created to include the child components.

<TitleComponent>

    <FirstChildComponent />

    <SecondChildComponent />

    <ThirdChildComponent />

</TitleComponent>

Q4: How can I write comments in JSX?

Enclosing the text in {/\* \*/} for both single and multiline.

{/\* <div className="search">Search Here</div> \*/}

{/\* <div className="search">Search Here</div>

<div className="res-container"> \*/}

Q5: What is <React.Fragment></React.Fragment> and <></>?

const myComponent = () =>{

  return (

<div className="search">Search Here</div>

        <div className="res-container">

  <p>Hello Fragment</p>

        </div>

);

  }

When you want to render a code like shown above – React throws an error “JSX expressions must have one parent”. This implies that we must surround the code with a wrapper node.

Before React 16, developers resolved this by using wrappers like <div> or <span> to group multiple elements returned from a component. This eventually led to the unnecessary addition of nodes in the DOM which could affect the styling and the layout of the document.

This is where <React.Fragment> or <Fragment> , often used as <></> shorthand syntax lets you group elements without a wrapper node. Grouping elements using Fragment has no effect on the resulting DOM; it is the same as if the elements were not grouped.

const myComponent = () =>{

  return (

  <Fragment>

  <div className="search">Search Here</div>

    <div className="res-container">

  <p>Hello Fragment</p>

    </div>

</Fragment>

);

}

The difference between <Frgament> and <></> , is that the shorthand syntax does not support the attribute “key”.

Q6: What is Virtual DOM?

DOM stands for Document Object Model, which is the representation of your application UI(HTML) in the object format. Whenever there is a change to the HTML page, the DOM gets updated, and the user can visualize these changes. Therefore, DOM is an interface that allows scripts to update, modify the content, style, and the structure of the document.

Problem:

DOM manipulation is the heart of the modern, interactive web. Unfortunately, it is a lot slower than most of the Javascript operations. This slowness is made worse by the fact that most Javascript frameworks update the DOM much more than they need to.

Example: Say you have a list of 20 items. You want to check-off the first item. Most Javascript frameworks would rebuild the entire list. Rebuilding the list is not a big deal for a web-browser, but modern websites can use huge amounts of DOM manipulations. Inefficient updating has become a serious problem.

This problem is rectified by **Virtual DOM.**

The Virtual DOM is a light-weight abstraction(clone) of the DOM. You can think of it as a copy of the DOM, that can be updated without affecting the real DOM. It has all the same properties as the real DOM object, but doesn’t have the ability to write to the screen like the real DOM.

Virtual DOM is just like a blueprint of a machine, can do the changes in the blueprint but those changes will not directly apply to the machine.

When you render a JSX element, every single virtual DOM object gets updated. Once the virtual DOM has been updated, React then **compares the virtual DOM with a virtual DOM *snapshot* that was taken right before the update. By comparing the new virtual DOM with a pre-update version, React figures out exactly which virtual DOM objects have changed**. This process is called “**diffing**.”

Once React knows which virtual DOM objects have changed, then React updates those objects, and only those objects, on the real DOM. In our example from earlier, React would be smart enough to rebuild your one checked-off list-item and leave the rest of your list alone.

This makes a big difference! React can update only the necessary parts of the DOM. React’s reputation for performance comes largely from this innovation.

In summary, here’s what happens when you try to update the DOM in React:

1. The entire virtual DOM gets updated.
2. The virtual DOM gets compared to what it looked like before you updated it. React figures out which objects have changed.
3. The changed objects, and the changed objects only, get updated on the real DOM.
4. Changes on the real DOM cause the screen to change.

https://www.codecademy.com/article/react-virtual-dom

Q7: What is Reconciliation in React?

A DOM tree is created to render all the components on the page. In case there are updates, React needs to figure out how to efficiently update the UI to match the most recent tree.

When you render a JSX element, every single virtual DOM object gets updated. Once the virtual DOM has been updated, React then compares the virtual DOM with a virtual DOM *snapshot* that was taken right before the update. By comparing the new virtual DOM with a pre-update version, React figures out exactly which virtual DOM objects have changed. This process is called “**diffing**”(an algorithm).

Reconciliation is the process to sync the virtual DOM with the real DOM by a library such as ReactDOM. Diffing algorithm is a technique of reconciliation used in React. This algorithm is also known as reconciler.

Q8: What is React Fiber?

Fiber is the new reconciliation engine in React 16. Its main goal is to enable incremental rendering of the virtual DOM.

The problem that Fiber solves:

Rendering the page, responding to the user actions, running Javascript and almost everything is handled by the browser’s main thread. Since our React code runs on Javascript, the browser’s main thread handles all the processing. If at any time, this main thread is blocked, the user’s experience can become laggy and slow (the white screen of death).

One solution is to move some of things to another thread safely using Web Workers, but they can’t access the DOM and manipulate it. This means that we can only move some of the lengthy heavy computational network request calls that do not change the DOM. There are workarounds, however, they tend not to be the best practice.

React Before Fiber:

Listed below is the process that React followed before Fiber:

1. React will create a tree of nodes (DOM) when the UI renders for the first time. Each node here is a React element.
2. A light-weight clone of the rendered tree called Virtual DOM is created.
3. Traversing the virtual DOM tree, React will update the DOM on whichever classes or elements need to be updated whenever a change occurs.
4. After any state change, React will compare every node from the 2 trees and pass on the changes to the Renderer which finally draws the element on the page.

This whole process would happen **synchronously**, meaning that once it was started, it could not be interrupted by another process until it was done. Given some expensive computation, it could potentially slow down how fast your app feels to your user.

Furthermore, the reconciliation(syncing) and rendering to the DOM were not separate so React can’t pause its traversal of the virtual DOM to jump to processing another render. New render changes also can’t be inserted once the reconciliation process begins to run.

All of this prevents high-priority changes from being made until the virtual DOM stack is completely cleared and ready.

React After Fiber:

Reconciliation and Rendering are now separate processes. By doing this, Fiber can help to prioritize different updates that can happen. React calls this **incremental rendering,** which splits the rendering work into chunks that can spread out over multiple chunks.

The new process is broken down into 2 phases:

1. Phase 1 - Reconciliation:

React makes a list of all the changes that need to be processed and then rendered to the UI. During this time, React can jump to processing another change as well.(main thread is no longer blocked)

Once this list is computed, React will then schedule the changes to be executed in the next phase.

1. Phase 2 — Commit

Out of the scheduled changes that come out of the reconciliation process, React can choose to render a specific set of changes.

Once committed, React notifies the DOM to render the changes that were found while in the reconciliation process.

While the reconciliation phase can be interrupted, the commit phase cannot.

By splitting things up into two phases, React is then able to prioritize which changes to make first. This helps you build high performing React applications that aren’t laggy.

Q9: Why we need keys in React? When do we need keys in React?

When you use React, at a single point in time you can think of the render() function as creating a tree of React elements. On the next state or props update, that render() function will return a different tree of React elements. React then needs to figure out how to efficiently update the UI to match the most recent tree.

There are some generic solutions to this algorithmic problem of generating the minimum number of operations to transform one tree into another. However, the [state of the art algorithms](https://grfia.dlsi.ua.es/ml/algorithms/references/editsurvey_bille.pdf) have a complexity in the order of O(n3) where n is the number of elements in the tree.

If we used this in React, displaying 1000 elements would require in the order of one billion comparisons. This is far too expensive. Instead, React implements a heuristic O(n) algorithm based on two assumptions:

1. Two elements of different types will produce different trees.
2. The developer can hint at which child elements may be stable across different renders with a **key prop.**

Consider the list:

<ul>

  <li>CSS</li>

  <li>Javascript</li>

</ul>

Case - 1) If I want to insert a new list item in the end of the below list, converting between these 2 trees works well. React will match the two <li>CSS</li> trees, match the two <li>Javascript</li> trees, and then insert the <li>React</li> tree. So, no problem.

Case -2) If I want to insert a new list item in the start of the list, the performance will degrade.

React will mutate every child instead of realizing that it can keep <li>CSS</li> and <li>Javascript</li>.

This can be resolved using the key attribute. When children have keys, React uses this key to match children in the original tree with the children in the subsequent updated tree.

<ul>

  <li key="101">CSS</li>

  <li key="102">Javascript</li>

</ul>

So, to insert <li key=”100”>HTML</li>, React knows that this element with key=”100” is a new entry and that the other two have to be just moved.

Q10: Can we use index as keys in React?

Yes, we can use index as keys but it not a good practice to use indexes as keys. Indexes are not stable identifiers and can change when the list order is modified, items are added or removed, or when the list is filtered or sorted. This can lead to components being unnecessarily re-rendered or reordered incorrectly.

Instead of using indexes as keys, it's recommended to use unique and stable identifiers from the data itself, such as IDs or unique attributes. Using meaningful keys ensures that components are properly identified and tracked, even when the list order changes or items are added or removed.

Q11: What is props in React? Ways to use props.

Props are nothing but properties that are used by React components to communicate with each other. Every parent component can pass on some information to its child component by giving them props. Props appear to be like HTML attributes, but you can pass any Javascript value through them like objects, arrays , functions anything.

**Ways to pass props**

1. To pass props, add them to the JSX, just like you would with HTML attributes.
2. To read props, use the function myComponent({ person, size }) destructuring syntax.
3. You can specify a default value like size = 100, which is used for missing and undefined props.
4. You can forward all props with <Avatar {...props} /> JSX spread syntax, but don’t overuse it!
5. Nested JSX like <Card><Avatar /></Card> will appear as Card component’s children prop.

Refer <https://react.dev/learn/passing-props-to-a-component>

Q12: What is a Config Driven UI ?

A configuration driven UI is a user interface design approach where the appearance, behavior, and functionality of the component are defined and controlled by the configuration settings rather than hardcoding them directly in the application code. Example: I have a component that returns an input field. I can have config settings such that this field can accept input text or password, have different style, different behavior etc.

Adv – 1) Creating reusable components.

2) Dynamic rendering

3) Scalability

4) Ease of maintenance