**React JS Notes**

1. **Class base Component: -**

* React started with Class based component. There was nothing like functional component, hooks, useState, useEffect.
* Writing code in class base component was very messy, no clean code. Class base component code was very big as comparedto functional component.
* Render() 🡪 We cannot make a class based component without render() method.
* API call like useEffect 🡪 **componentDidMount** is used in class based for API call.
* Component life cycle sequence –

1. Constructor
2. Render
3. componentDidMount

* ..
* ..

1. **Route: -**

* Nested Route 🡪 We can create nested route by Children of Children
* For children of children route we don’t give “/”. Example below:

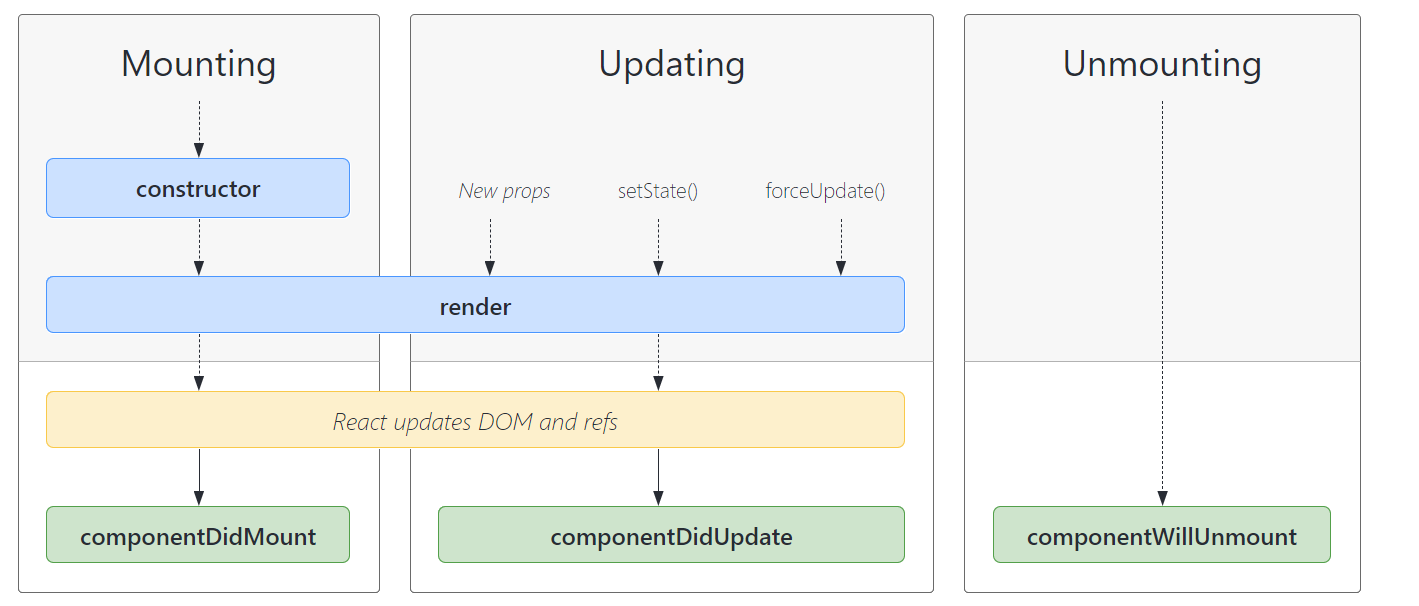


* ..

1. **Why do we do super(props) ?**
2. In a React class-based component, the **super(props)** method is used to call the constructor of the parent class (i.e., the **React.Component** class) and pass the props that were passed to the current component as an argument. This is necessary because the constructor of the parent class is responsible for initializing the component's state and props, and by calling **super(props)**, we ensure that this initialization is done before we start using the props in the child component's constructor.
3. It's also important to note that **super(props)** should be called before any other statement in the constructor because the **this** keyword can only be used after the parent class's constructor has been called.
4. Additionally, when the parent component pass the props and state to the child component, it's important to make sure that the child component can access those props and state. Calling **super(props)** makes it possible for the child component to access the props and state passed down from the parent component.
5. In summary, **super(props)** is used to call the constructor of the parent class, pass the props to the parent class, and ensure that the parent class has initialized the component's state and props before we start using them in the child component's constructor.
6. **2 Phases are there when reconcilliation happes**
7. Render Phase 🡪 Here 1st constructore is executed and then render executes
8. Commit Phase 🡪 Here 1st DOM updates and then componentDidMount executes
9. **We can make componentDidMount async but we cannot make useEffect() async ? Reason ?**

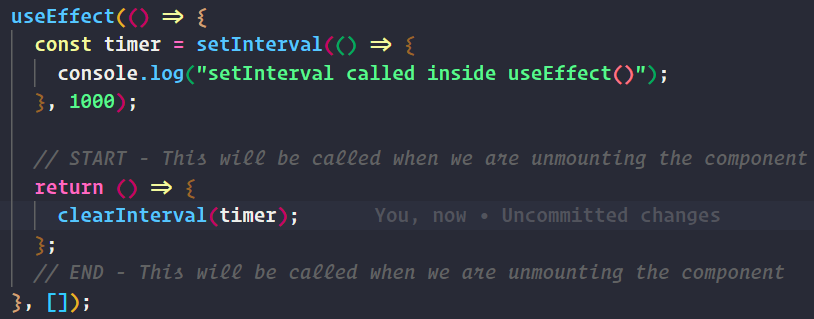
* **useEffect** is a hook that allows you to synchronously synchronize a component with an external system. It is not designed to handle asynchronous operations, such as fetching data from a remote server. On the other hand, **componentDidMount** is a lifecycle method in a class component that is called after the component has been rendered. Because it is called after the component has been rendered, it is the appropriate lifecycle method to use when you need to perform an asynchronous operation, such as fetching data. It is possible to make useEffect async by wrapping the effect with an async function but it's not recommended because it would cause an unnecessary re-rendering and it is not guaranteed to work as expected.

1. **componentDidMount()** 🡪 Will be called after 1st render().
2. **componentDidUpdate() 🡪** Will be called after every 1st render(). That is from 2nd render time this will be called
3. **React life cycle method diagram below** – https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/



1. Note 🡪 Never compare class-based life cycle with functional based life cycle.
2. useEffect 🡪 When we don’t put any dependency array in it then useEffect will be called after every render()
3. useEffect 🡪 When we put empty dependency array [], then it will be called only once in the initial render()
4. UnMount in Functional Based Component:

* We need to use return() method inside useEffect()]to unmount a component in class based component. Example below



* useEffect return will execute once we leave that component.

1. **Custome Hooks:**
2. Why do we build Hooks?

* Reusability, Readability, Testable, Modularity, Seperation of Concern, Maintainability.
* Wrap a small logic in function and can reuse it anywhere we want to.
* Modularity 🡪 Means code broken down into smaller meaning full pieces.

1. ..
2. ..
3. ..
4. ..
5. ..
6. ..
7. ..
8. ..
9. ..
10. ..
11. ..
12. ..
13. ..
14. ..
15. ..
16. ..
17. ..