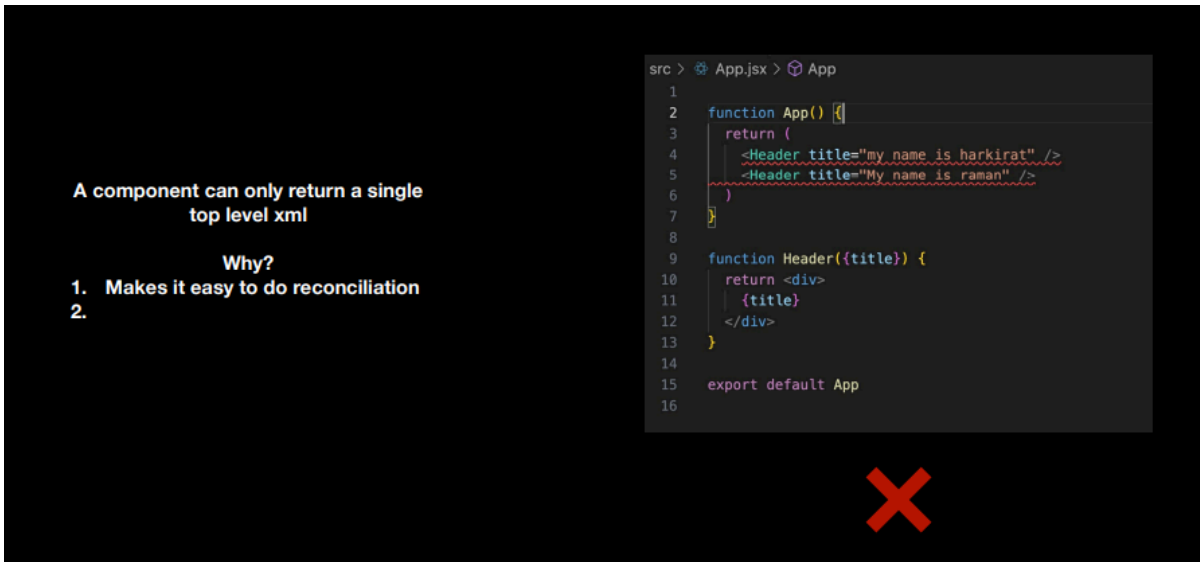
React Deeper Dive

(React returns, re-rendering, key, Wrapper components, useEffect, useMemo, useCallback, useRef)

React component return

A component can only return top level xml (We can't return multiple sibling) . Why??

1. Makes it easy to do reconciliation (the process of figuring out what DOM updation need to be done as our application grows) .



Create a react app that has a

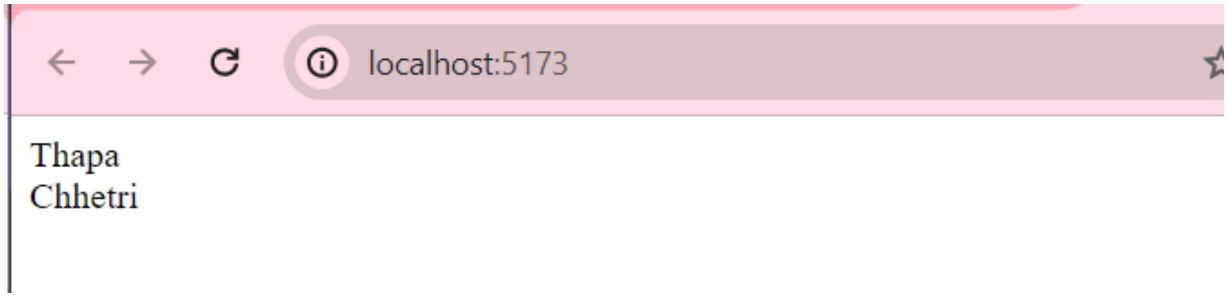
1. Header component that takes a title as a prop and renders it inside a div.
2. The top level App component renders 2 Headers

npm create vite@latest

npm install

```
function App() {
  return(
      <div>
        <Header title="Thapa"></Header>
```

```
        <Header title="Chhetri"></Header>
      </div>
  )
}

function Header({title}){
  return <div>
    {title}
  </div>
}

export default App
```



We can only return xml which has a single parent element

We can use

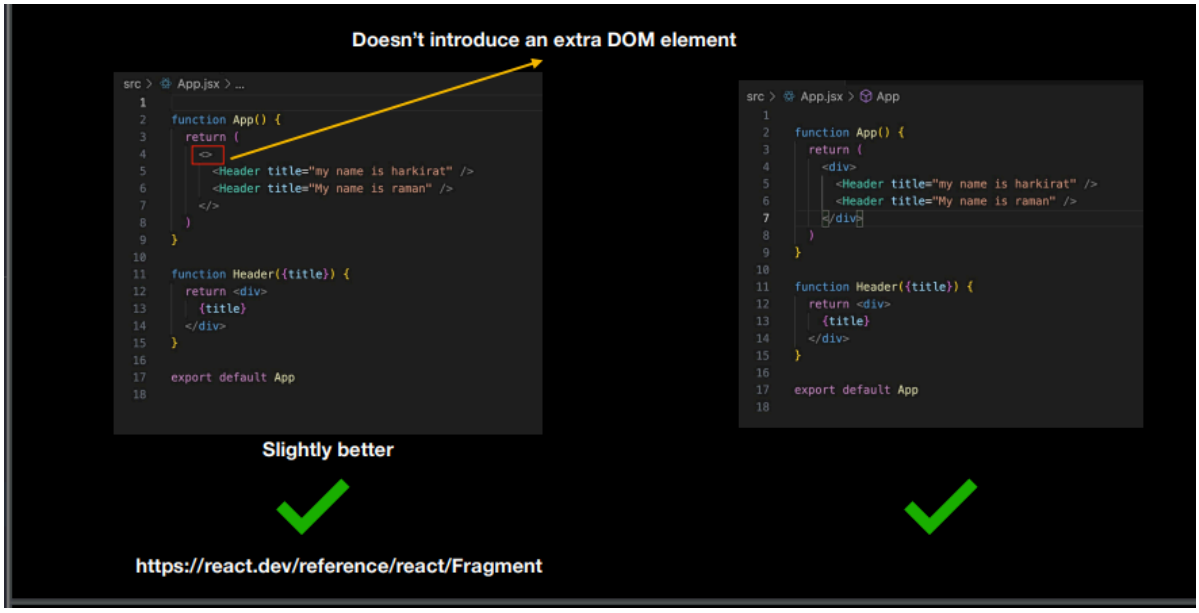
<>

</>

They don’t have parent in final render

```
      <React.Fragment>
        <Header title="Thapa"></Header>
        <Header title="Chhetri"></Header>
      </React.Fragment>
```

This also does the same thing



Re-rendering in react

Rerendering in React refers to the process of updating and rendering components to reflect changes in the application's state or props. When there's a change in the state or props of a component, React re-renders that component and any affected child components. It's important to note that a rerender doesn't necessarily mean a complete re-rendering of the entire DOM; instead, React efficiently updates only the necessary parts of the DOM.

Basically, anytime a final DOM manipulation happens or when react actually updates the DOM it is called a rerender.

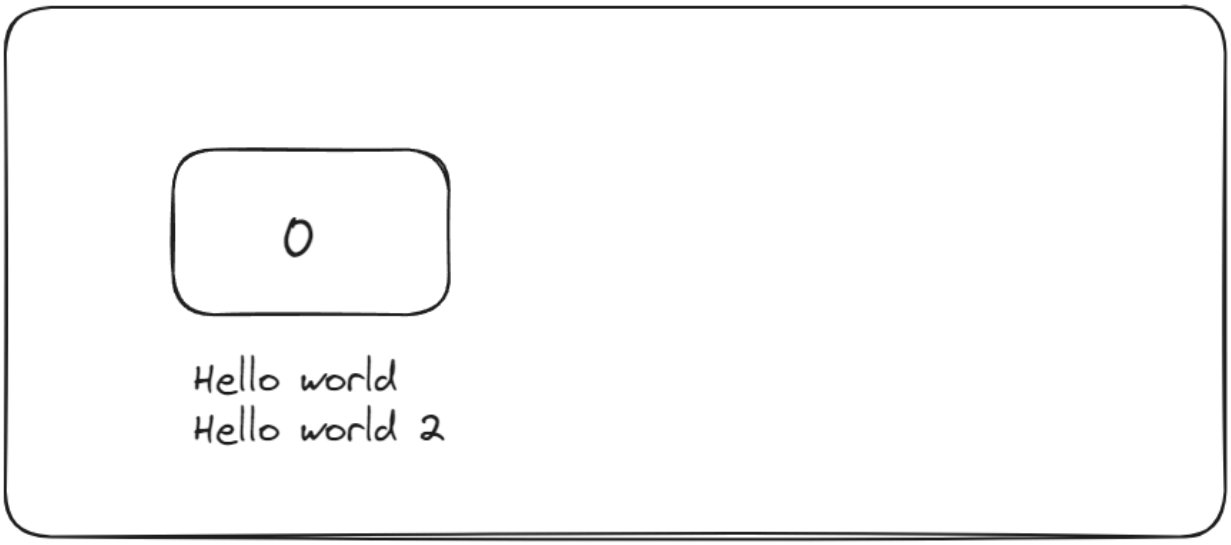
What exactly is re-render??
Install this react developer tool to visulaize it

React —> Dynamic website.
Dynamic feature: insert something to DOM, delete something to DOM or update something in DOM

Counting app is good example of re-render

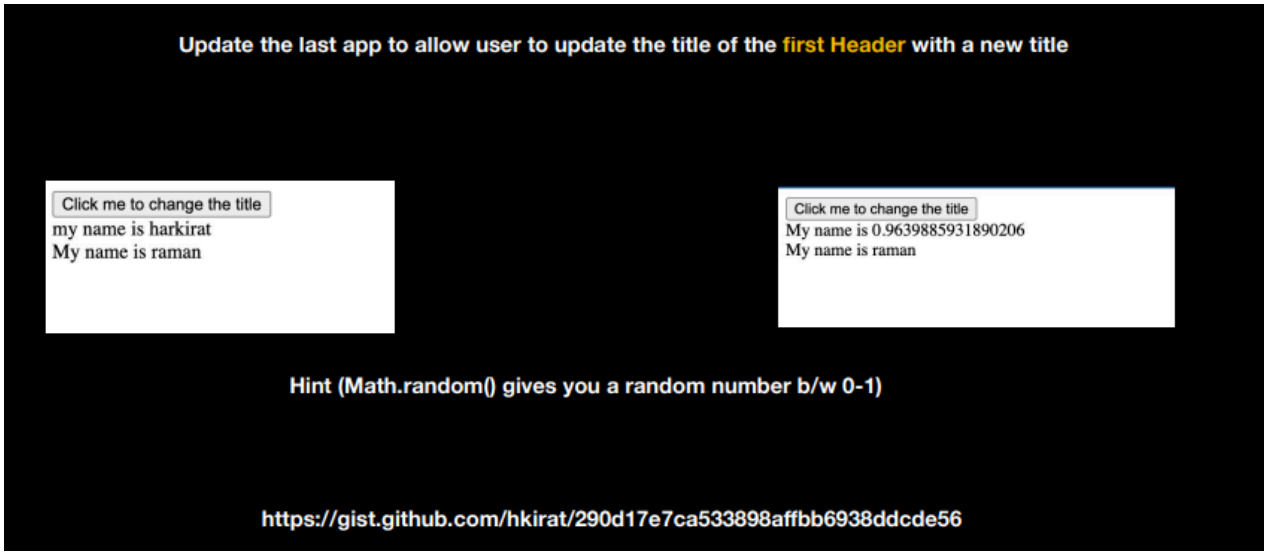
Rule of react is to minimize re-renders

(Special in mobile app and games)



Whenever 0 changes does hello world need to re-render?
Not mostly. Only button should re-render

But in react there are app in which only the button should re-render but even static element like text (“Hello world”) is getting re-render.

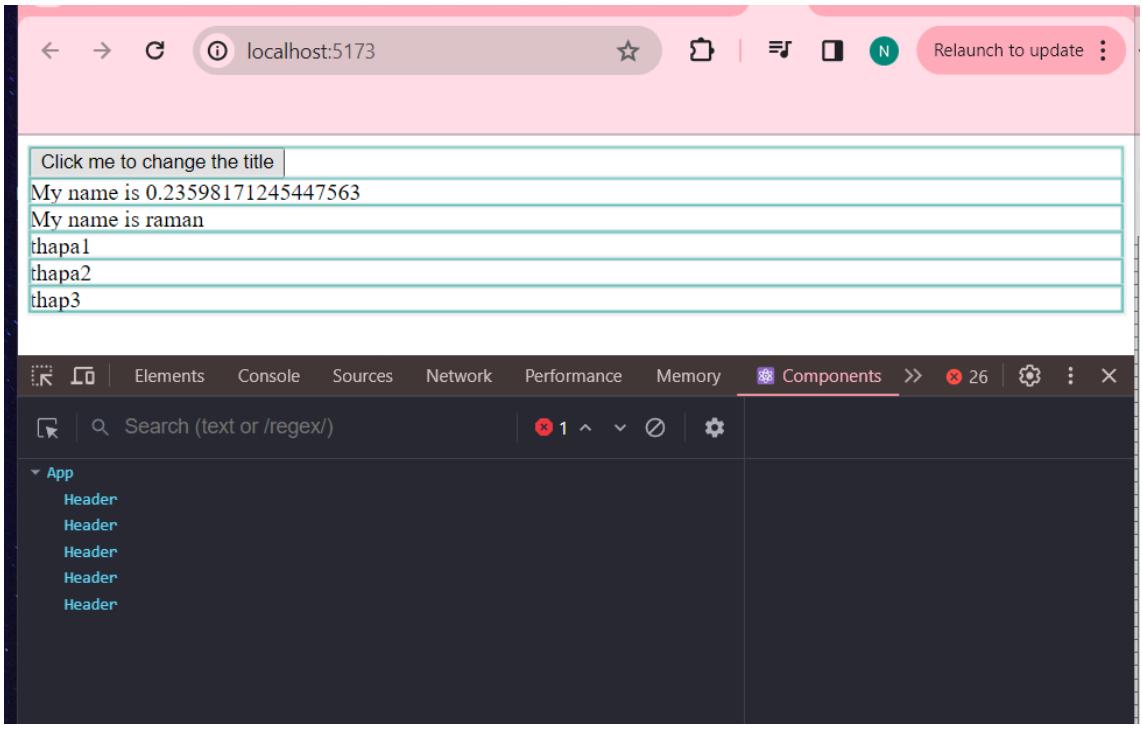


```
import { useState } from "react"

function App() {
  const [firstTitle, setFirstTitle] = useState("my name is harkirat");

  function changeTitle() {
    setFirstTitle("My name is " + Math.random())
  }

  return (
    <div>
      <button onClick={changeTitle}>Click me to change the title</button>
      <Header title={firstTitle} />
      <Header title="My name is raman" />
    </div>
  )
}

function Header({title}) {
  return <div>
    {title}
  </div>
}

export default App
```

React should re-render only the first header ideally , but we can see(inspect -> Components -> then click the Update the title button)app getting re-rendered than all three are getting re rendered

```
import React, {Fragment} from "react"
import { useState } from "react"
function App() {
  const [title, setTitle] = useState("my name is Thaaape");
  function changeTitle() {
    setTitle("My name is " + Math.random())
  }
  return (
    <div>
      <button onClick={changeTitle}>Click me to change the title</button>
```

```
      <Header title={title} />
      {/* Dynamic header */}
      <Header title="My name is raman" />
      <Header title="thapa1" />
      <Header title="thapa2" />
      {/* Bunch of static header */}
      <Header title="thap3" />
    </div>
  )
}

function Header({title}) {
  return <div>
    {title}
  </div>

}

export default App
```



The blue box show the things which are getting re-rendered

button is not even a component
Ideally App should get re-rendered(React calling that function) as child is re-rendered.
It is thing which is wrong.
Even static Header are getting re-render

Re-rendering in react
A re-render means that

1. React did some work to calculate what all should update in this component.
2. The component actually got called (you can put a log to confirm this)
3. The inspector shows you a bounding box around the component

It happens when

1. A state variable that is being used inside a component changes
2. **A parent component re-render triggers all children re-rendering**

irrespective whether children has used state variable or not. This cascading effect ensures synchronization throughout the component tree.

We want to minimize the number of re-renders to make a highly optimal react app
The more the components that are getting re-rendered , the worse.

Reconciliation
One of the most prominent reasons for it is **Reconciliation**. The single-root element rule in React facilitates the reconciliation process, where React efficiently **updates the real DOM based on changes in the virtual DOM**. By having a single root element, React can easily perform the comparison between the previous and current states of the virtual DOM.

Reconciliation involves identifying what parts of the virtual DOM have changed and efficiently updating only those parts in the actual DOM. The single-root structure

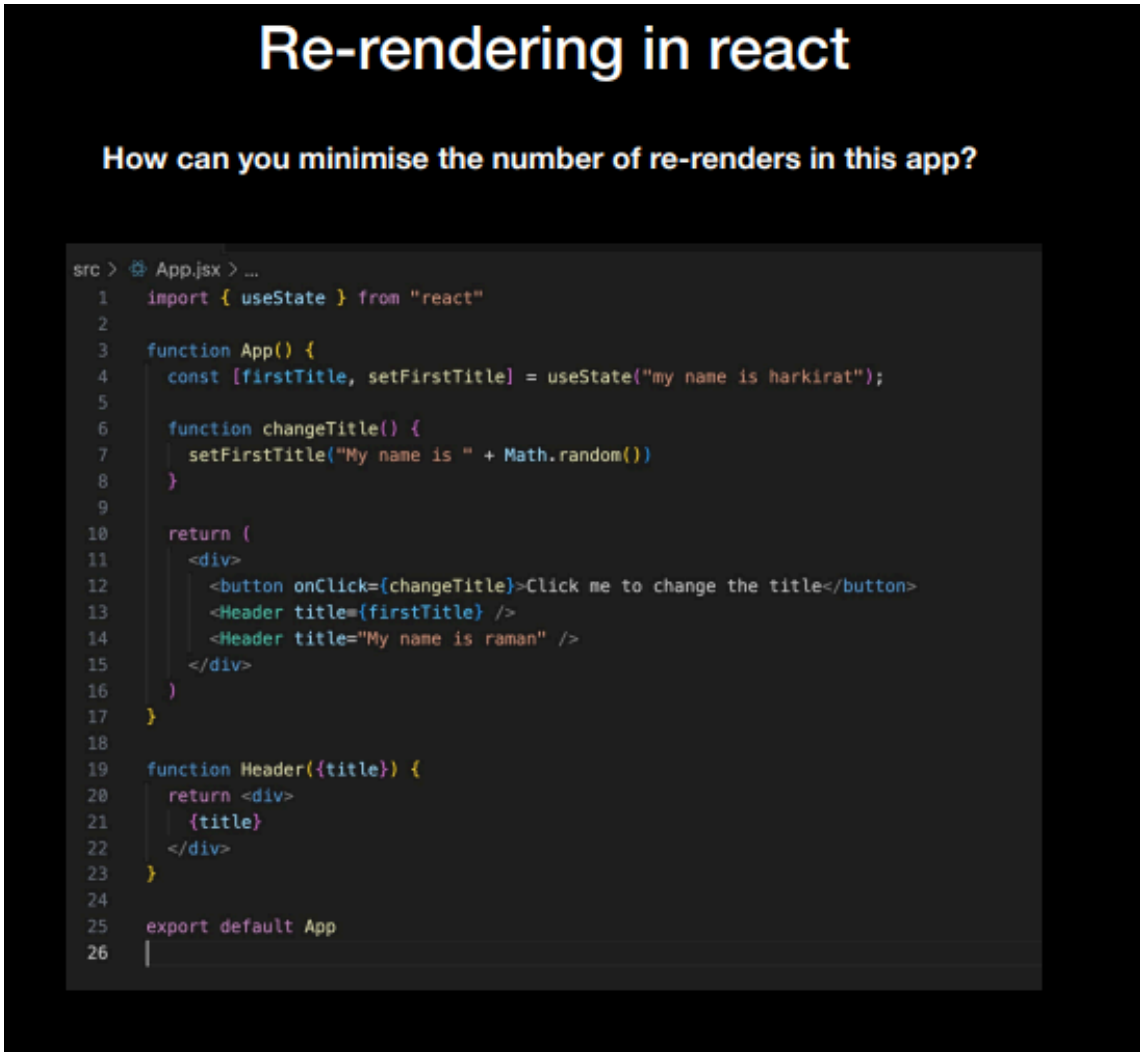
simplifies this process by providing a clear entry point for React to determine where updates should occur.

In addition to reconciliation, it aids in maintaining a straightforward and predictable structure in React components, making the code more readable and understandable. This constraint encourages developers to create components with well-defined boundaries, which enhances code organization and modularity.

While a single root element is required, React provides a feature called fragments (**<></>** or **<React.Fragment></React.Fragment>**) that allows you to group multiple elements without introducing an extra node in the real DOM. Fragments don't create an additional parent in the DOM but still satisfy the single-root rule.

There are broadly 2 ways of minimizing the amount of rerenders

- Push the State down
- By Using Memoization



One way is to pushing the state down:

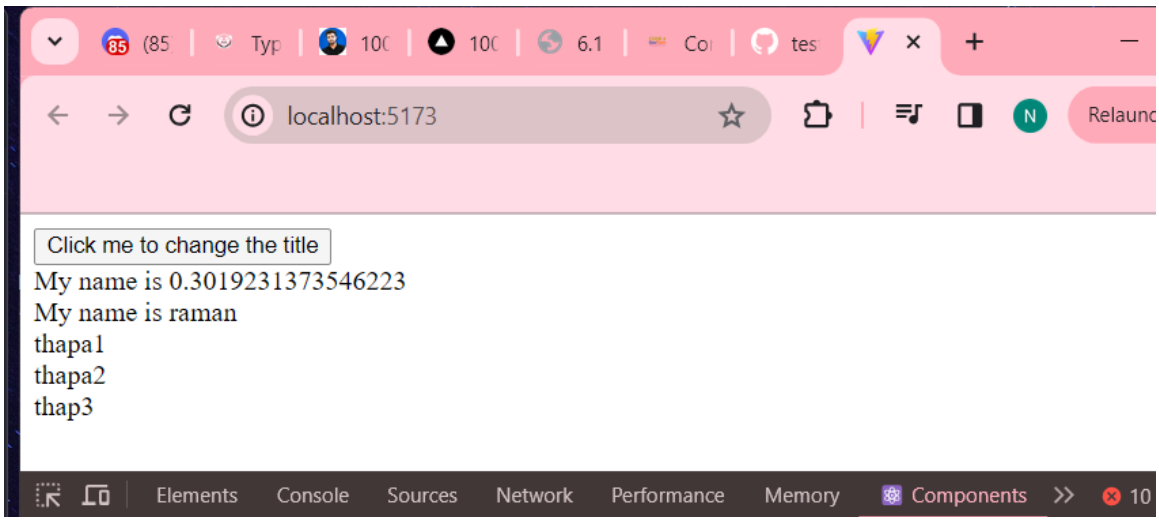
Pushing the state down in React refers to the practice of managing state at the lowest possible level in the component tree. By doing so, you localize the state to the components that absolutely need it, reducing unnecessary re-renders in higher-level components.

When state is kept at a higher level in the component tree, any changes to that state can trigger re-renders for all child components, even if they don't directly use or depend on that particular piece of state. However, by pushing the state down and ensuring that each component only has access to the state it needs, you can minimize the impact of state changes on the overall component tree.

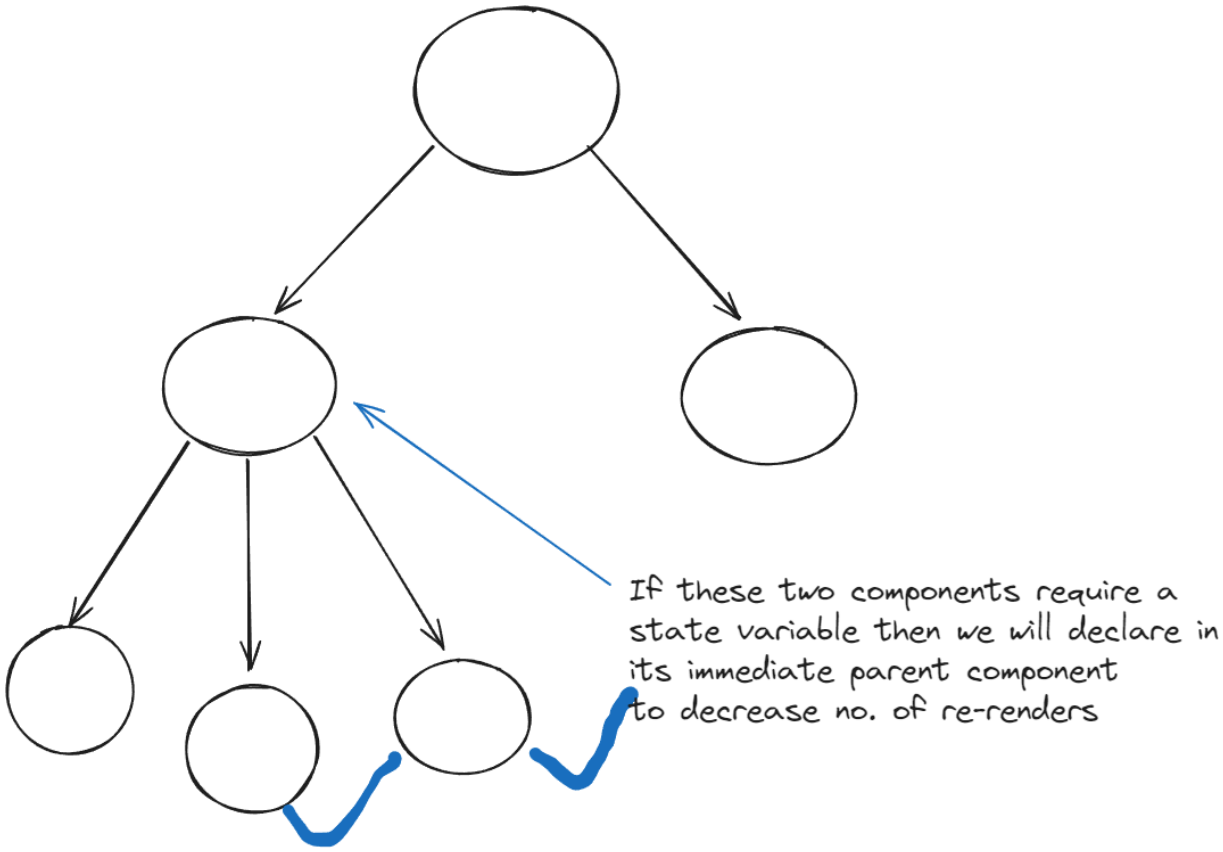
src > App.jsx > ...

```jsx
import { useState } from "react"

function App() {
  return (
    <div>
      <HeaderWithButton />
      <Header title="My name is raman" />
    </div>
  )
}

function HeaderWithButton() {
  const [firstTitle, setFirstTitle] = useState("my name is harkirat");

  function changeTitle() {
    setFirstTitle("My name is " + Math.random())
  }

  return <>
    <button onClick={changeTitle}>Click me to change the title</button>
    <Header title={firstTitle} />
  </>
}

function Header({title}) {
  return <div>
    {title}
  </div>
}

export default App
```

Neither App function re-render nor Header component , only **HeaderWithButton** component will re-render



Only the HeaderWithButton re-render



Never keep at root.

We try to push down state as possible.

Method 2: By using Memoization

The above problem of reducing the number of rerenders can also be tackled using Memoization. Memoization in React, achieved through the useMemo hook, is a technique used to optimize performance by memoizing (caching) the results of expensive calculations. This is particularly useful when dealing with computations that don't need to be recalculated on every render, preventing unnecessary recalculations and re-renders.

In the context of minimizing re-renders, useMemo is often employed to memoize the results of computations derived from state or props. By doing so, you can ensure that the expensive computation is only performed when the dependencies (specified as the second argument to useMemo) change.

react.memo

memo lets you skip re-rendering a component when its props are unchanged.

```
import React from "react"
import { useState } from "react"

function App() {
const [title,setTitle] = useState("My name is Thapuu");
function updateTitle(){
  setTitle("My name is " + Math.random())
}

  return (
    <div>
      <button onClick={updateTitle}>Update the Title</button>
      {/* Dynamic header */}
      <Header title={title} />
      <Header title="thapa1" />
      <Header title="thapa2" />
      {/* Bunch of static header */}
      <Header title="thap3" />
      <Header title="thap4" />
      <Header title="thap5" />
    </div>
  )
}

const Header = React.memo(function Header({title}) { //we can also
//desctructure and use it
  return <div>
    {title}
  </div>
})

export default App
```

Parent div re-renders only the static components dont rerender.

Q/Na

1. **We still can’t put state variable inside App**
2. React compare Real DOM and Virtual DOM
3. Can we pass function of one component to another? Yes
4. No one memoization , it matter in React-native application
5. What is lifting up the state??
6. State should always be pushed from parent to child
7. Example of re-rendering can be like stop watch on our chrome screen

Keys in React

Significance of Key in React

In React, when rendering a list of elements using the map function, it is crucial to assign a unique key prop to each element. The "key" is a special attribute that helps React identify which items have changed, been added, or been removed. This is essential for efficient updates and preventing unnecessary re-renders of the entire list.

When the key prop is not provided or not unique within the list, React can't efficiently track the changes, leading to potential issues in the application's performance and rendering.

Lets create a simple todo app that renders 3 todos.

1. Create a Todo component that accepts title, description as input.
2. Initialise a state array that has 3 todos
3. Iterate over the array to render all the TODOs
4. A button in the top-level App component to add a new TODO

```
import { useState } from "react"
```

```
function App() {
  return (
    <div>

    </div>
  )
}

function Todo({title, description}){
  return <div>
   <h1>{title}</h1>
   <h5>{description}</h5>
  </div>
}

export default App
```

Initialising a state array that has 3 todos

```
  const [todos, setTodos] = useState([
    { id: 1,
      title: 'Todo 1',
      description:"Going to Sac at 6PM"
    },
    {
      id: 2,
      title: 'Todo 2',
      description:"Start studying from 9PM"
    },
    { id: 3,
      title: 'Todo 3',
      description:"Sleep before 1 AM"
    }
  ]);
```

Iterate over the array to render all the TODO’s

```
return (
    <div>
```

```
      {todos.map(todo => <Todo title={todo.title}
description={todo.description}/>)}
      {/* {todos.map(function(todo){
        return <Todo title={todo.title} description={todo.description} />
      })} */}
    </div>
  )
```

A button in top level to add all new todos

```
      <button onClick={addTodo}>Add a todo</button>
```

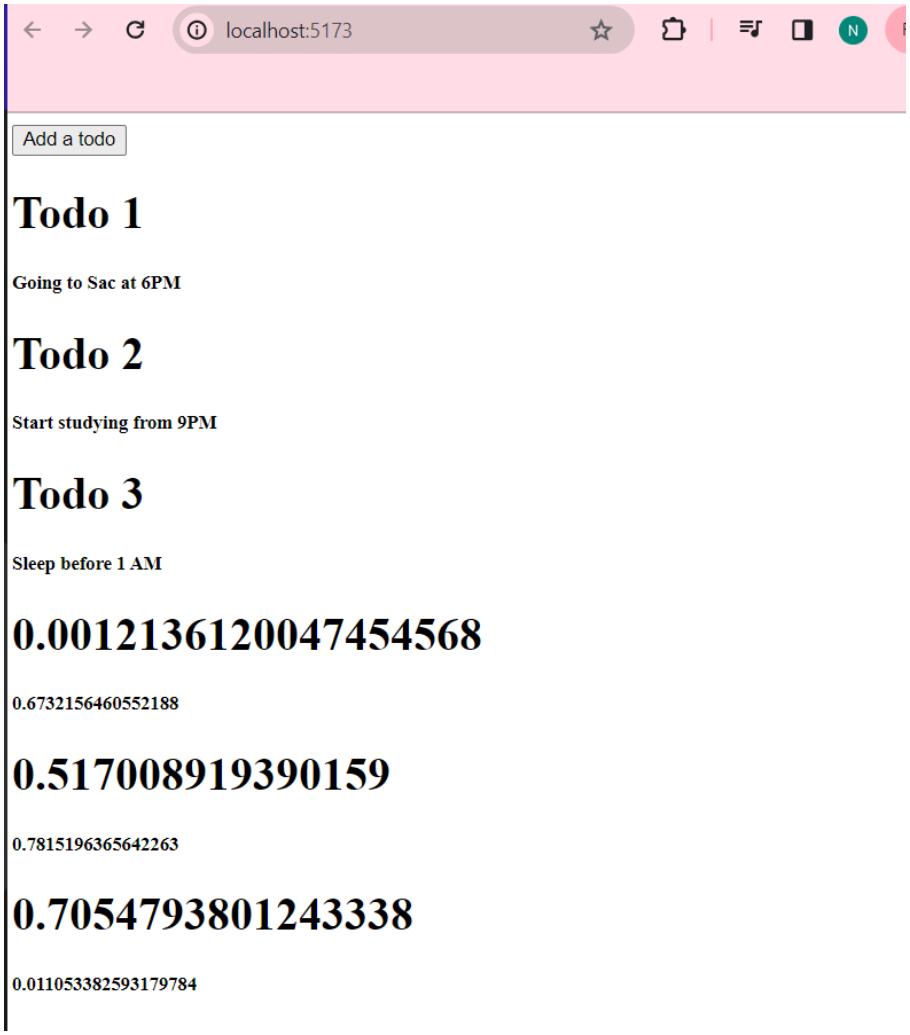
```
  function addTodo(){
    // spread operator
    setTodos([...todos,{
      id: 4,
      title: Math.random(),
      description: Math.random()
    }])
    // const newTodos = [];
    // for(let i=0; i<todos.length; i++){
    //   newTodos.push(todo[i]);
    // }
    // newTodos.push({
    //   id:4,
    //   title:Math.random(),
    //   description:Math.random()
    // })
    // setTodos(newTodos)
  }
```

Solution:

```
import { todo } from "node:test";
import { useState } from "react"


let counter = 4;
// normal js variable
```

```
// we haven't kept it inside the App function so that whenever App
re-render it wont reset its value to 4


function App() {
  const [todos, setTodos] = useState([
    { id: 1,
      title: 'Todo 1',
      description:"Going to Sac at 6PM"
    },
    {
      id: 2,
      title: 'Todo 2',
      description:"Start studying from 9PM"
    },
    { id: 3,
      title: 'Todo 3',
      description:"Sleep before 1 AM"
    }
  ]);

  function addTodo(){
    // spread operator
    setTodos([...todos,{
      id: counter++,
      title: Math.random(),
      description: Math.random()
    }])
  }

  return (
    <div>
      <button onClick={addTodo}>Add a todo</button>
      {todos.map(todo => <Todo title={todo.title}
description={todo.description}/>)}
    </div>
  )
}

function Todo({title,description}){
```

```
  return <div>
    <h1>{title}</h1>
    <h5>{description}</h5>
  </div>
}

export default App
```



Id : is hardcoded to 4

For that we can define a global variable

We will see an error when we inspect our code:

We need to give each array item a key – a string or a number that uniquely identifies it among other items in that array.

Keys tell React which array item each component corresponds to so that it can match them up later. This becomes important if your array items can move (e.g. due to sorting). get inserted, or get deleted. A well-chosen key help React infer what exactly has happened, and make the correct updates to the DOM tree.

This basically increases the number of re-renders . Since React gets confuse.
Now after adding

```
      {todos.map(todo => <Todo key={todo.id} title={todo.title}
description={todo.description}/>)}
```

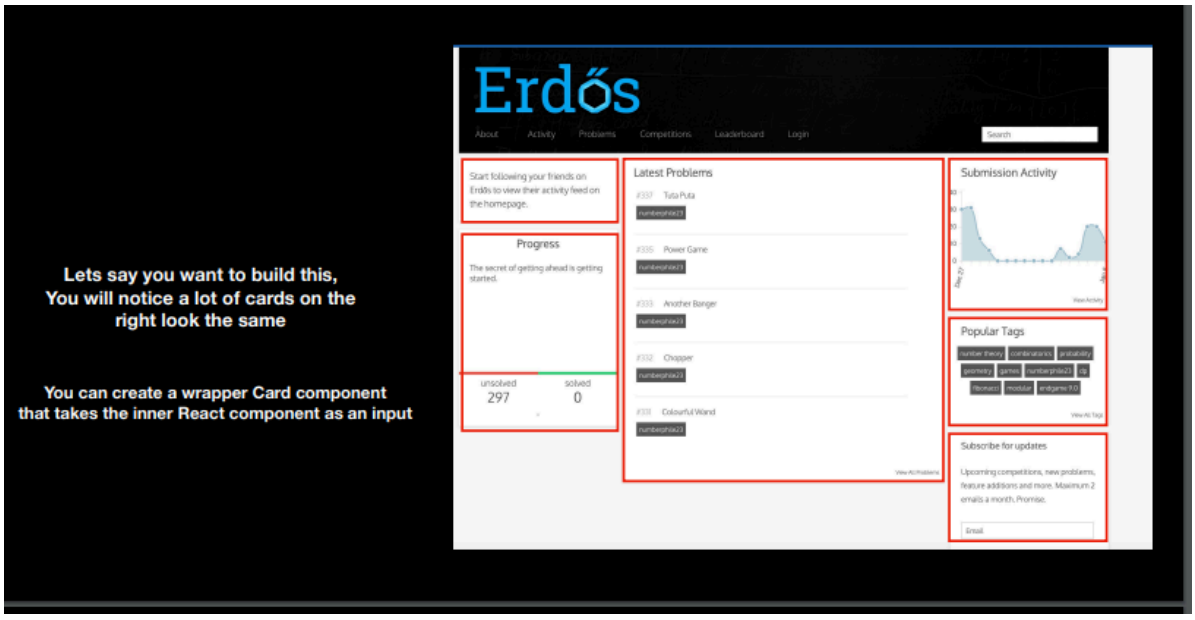
Now even if array is swap it still give us in output in least re-renders.
For example our todo array has three todos with id 1,2,3
If we move todo with id = 3 to the top, and without passing the keys to the React then React won't be able to calculate that id=3 existed and has been shifted to the top , and only do that operation instead it will remove all three todos and add them again .

Keys let react figure out if a
TODO has been update,
Which has been delete,
Which has been added.

Wrapper components:

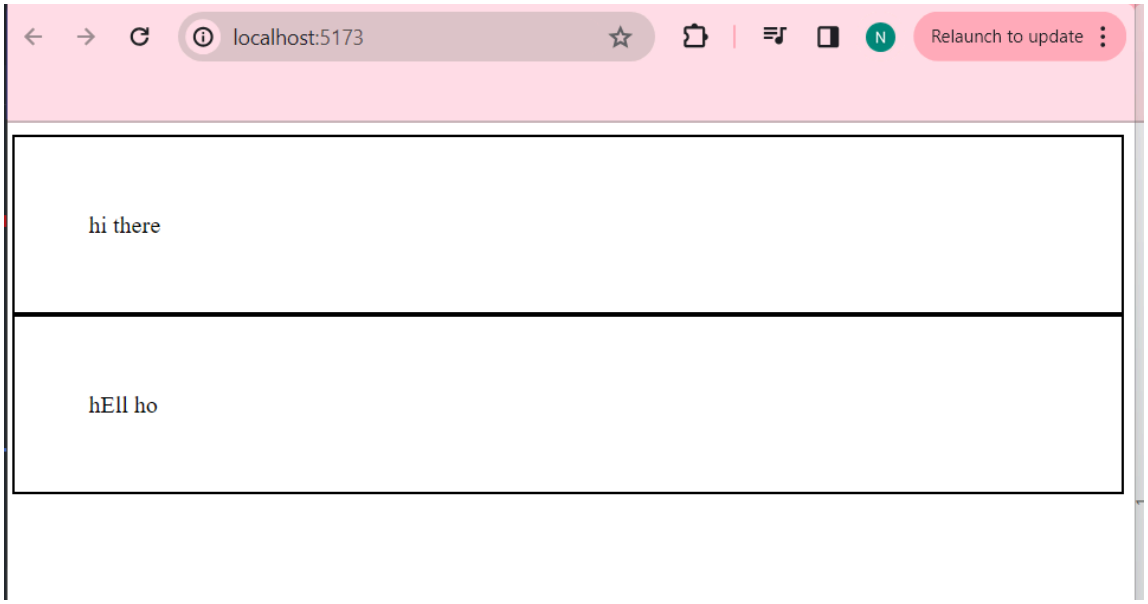
In React, wrapper components are used to encapsulate and group common styling or thematic elements that need to be applied consistently across different parts of an application. These components act as containers for specific sections or functionalities, allowing for a clean and modular structure.

Goal to write single component as parent and which take input and has some style
Example:
Let's consider an example where we have a wrapper component called Card that provides a consistent styling for various content sections, such as blog posts. The Card component maintains the overall styling, while different contents can be dynamically injected



With this structure, we maintain a consistent card styling across different sections of our application, promoting reusability and making it easy to manage the overall theme. This approach is especially beneficial when you want to keep a uniform appearance for similar components while varying their internal content.

```
function App(){
  return <div>
    <CardWrapper innerComponent={<TextComponent/>}/>
    <CardWrapper innerComponent={<TextComponent2/>}/>
  </div>
}

// CardWrapper need to accept some React component as an  input and then
render it inside some extra style
function CardWrapper({innerComponent}){
    // create a div which has a border
    // inside the div , renders the prop
    return <div style={{border:"2px solid black",padding:"50px"}}>
      {innerComponent}
    </div>
}
function TextComponent(){
  return <div>
    hi there
  </div>
}
function TextComponent2(){
  return <div>
    hEll ho
  </div>
}

export default App;
```



Another way / Better way (To create a Wrapper)(Giving Structure of Card)(Now people can write the content of it)

```
// better way , we don't pass props here
function App(){
  return <div>
    <CardWrapper >
      hi there
      {/* bunch of children can be text or a div etc  */}
      <div>
        This is an div element
      </div>
    </CardWrapper>
  </div>
}

// here we accept children, which has everything we have written inside
function CardWrapper({children}){
    console.log("displaying children content")
    console.log(children)
    return <div style={{border:"2px solid black",padding:"50px"}}>
      {children}
    </div>
```

```
}

export default App;
```

Displaying the content of the children object

We can do Wrapper inside Wrapper, etc

Solution:

```
function App() {

  return (
    <div style={{display: "flex"}}>
      <Card>
        hi there
      </Card>
      <Card>
        <div>
          hello from the 2nd card
        </div>
      </Card>
    </div>
  )
}

function Card({children}) {
```

```
  return <div style={{
    border: "1px solid black",
    padding: 10,
    margin: 10
  }}>
    {children}
  </div>
}

export default App
```

Class Components Vs Functional Components

In React, components are the building blocks of user interface. There are two main types of components: class-based components and functional components

1. Class Based Components
    - Class- Based Components are ES6 classes that extend from React.Component.
    - They have access to the lifecycle methods provided by React, such as **componetnDidMount**, **componentDidUpdate**, and **componentWillUnmount.**
    - State and lifecycle methods are managed within class-based components.
    - They were primary type of components before the React 16.8

2. Functional Components
    - Functional components are simpler and more concise. They are essentially Javascript functions that take props as an argument and return React elements.

- With the introduction of React hooks in version 16.8 functional components gained the ability to manage state and use lifecycle methods through hooks like **useState** and **useEffect.**

Notes:

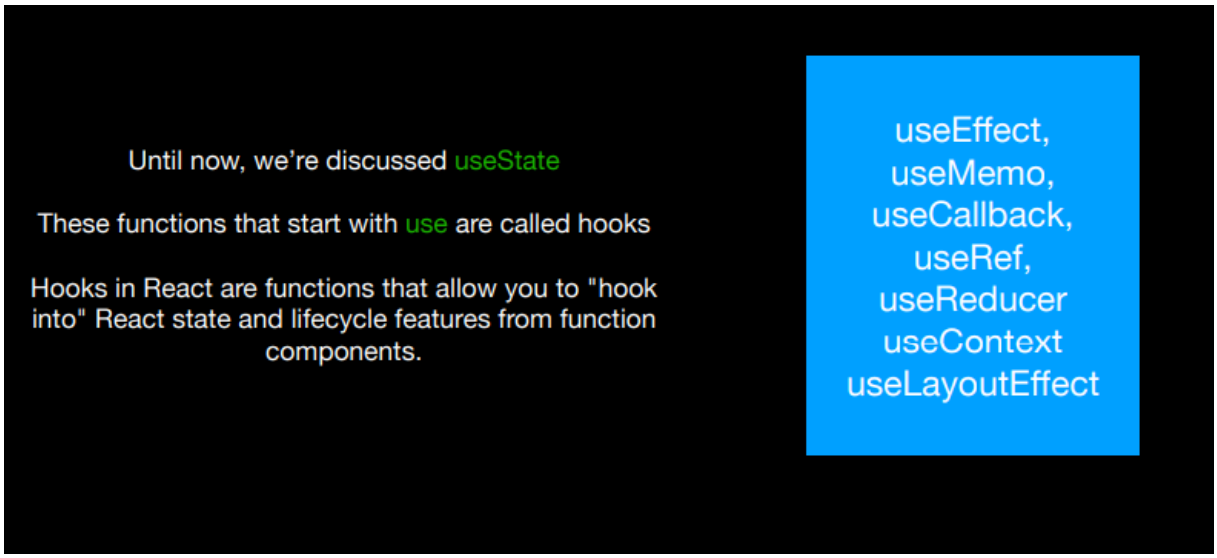
- Functional components are now the preferred way to write components in React due to their simplicity and the additional capabilities provided by hooks.
- Hooks like useState and useEffect allow functional components to manage state and perform side effects, making them as powerful as class-based components.
- Class-based components are still used in some codebases, especially in projects that haven't migrated to functional components or are working with older React versions.

Hooks

React Hooks are functions that allow functional components in React to have state and lifecycle features that were previously available only in class components.

Using these hooks, developers can manage state, handle side effects, optimize performance, and create more reusable and readable functional components in React applications. Each hook serves a specific purpose, contributing to a more modular and maintainable codebase.

Some commonly used React Hooks are: **useEffect, useMemo, useCallback, useRef, useReducer, useContext, useLayoutEffect**



Lifecycle features:

useEffect()

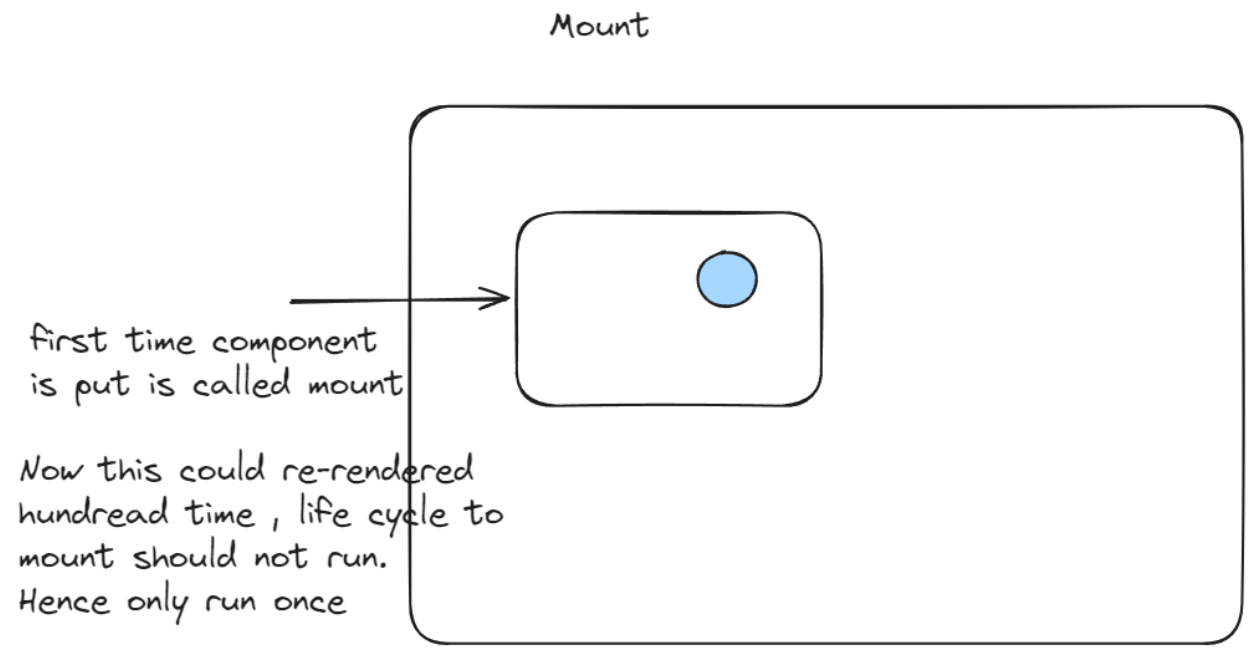
useEffect is a React Hook used for performing side effects in functional components. It is often used for tasks such as data fetching, subscriptions, or manually changing the DOM.

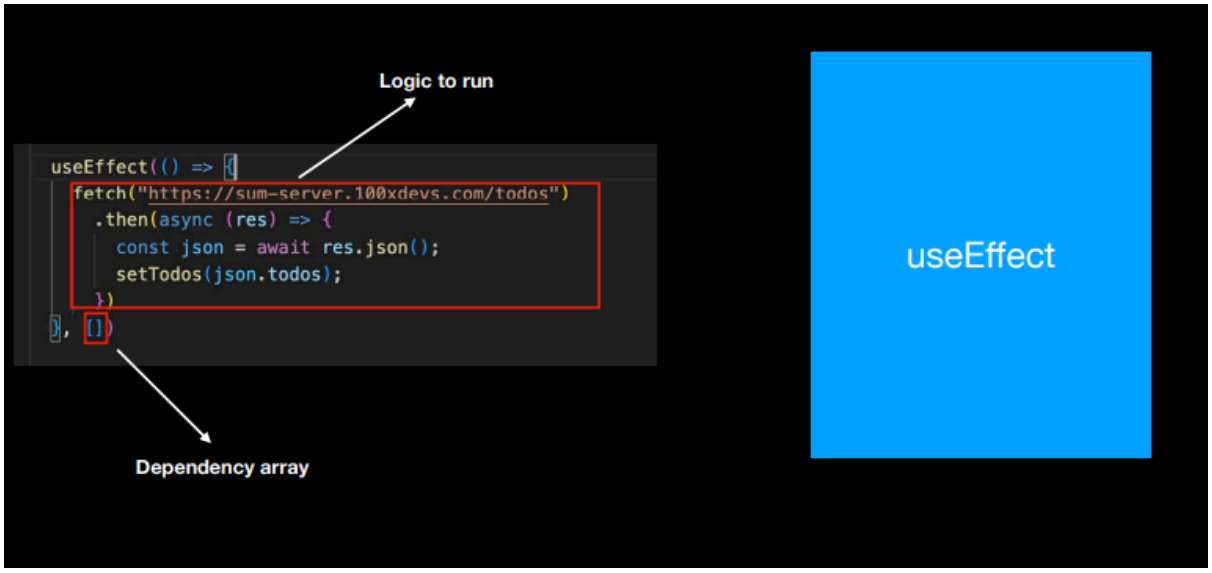
The useEffect hook accepts two arguments: a function that contains the code to execute, and an optional array of dependencies that determines when the effect should run.

Example of new and old syntax.(called lifestyle event)

```
const {useEffect} = require("react")

function App(){
  useEffect(function(){
    alert("hi")
  },[])
```

```
  return <div>

  </div>
}

class App{
  componentDidMount(){ // first time it mount
    alert("hi")
  };
}
```





Assignment

Backend Server: https://sum-server.100xdevs.com/todos

Send a set of request every some seconds to get the current set of todos

```
import { useState } from "react";
import { useEffect } from "react";

function App(){
  const [todos,setTodos] = useState([])

  // we can't write first argument an async function inside useEffect
```

```
  useEffect(()=>{
    setInterval(()=>{
      fetch("https://sum-server.100xdevs.com/todo?id=1")
    .then(async function(res){
        const json = await res.json();
        setTodos(json.todos);
    })
    },10000)
  },[])

  // asking one request when the component mounts
  return <div>
    {todos.map(todo => <Todo key={todo.id} title={todo.title}
description={todo.description}/>)}
  </div>
}

function Todo({title,description}){
  return( <div>
    <h1>
      {title}
    </h1>
    <h4>
      {description}
    </h4>
  </div>)
}
export default App;
```

Q/Na:

1. **What is dependency array**

```
import { useState } from "react";
import { useEffect } from "react";

function App(){
  const[counter,setCounter] = useState(0)
  useEffect(()=>{
```

```
    // run this code whenever counter changes.
  },[counter])
  return <div>
    <button onClick={()=>{
      setCounter(counter++);
    }}>Increases Count</button>
  </div>
}
export default App;
```

2. Memo let us skip re-rendering of a component whenever props are unchanged.
3. useAsyncEffect library
4. How do you unmount(something got removed) a component .

There are 3 todo initially and then 0 todo afterwards we say 3 todos are unmounted.

5. We have to passs dependency array otherwise it will run infinitely
6. No re-render if we don't have state
7. We can't use current index as id , as it will tell react not the correct thing but it will work

{todos.map((todo,index) => <Todo key={index} title={todo.title} description={todo.description} />)}

8. When we use memo, and if we return everything inside <> (Fragment) then it whole thing will be re-rendered.
9. Memo : lets you skip re-rendering a component when its props are unchanged.
   useEffect(): let us run a certain code , when anything inside the array changes.