Common hooks in React

(useEffect, useMemo, useCallback, custom hooks Prop drilling**)**

In this lecture, Harkirat explores key aspects of React, starting with React Hooks like useEffect, useMemo, useCallback, and more, providing practical insights into state management and component functionalities. The discussions extend to creating custom hooks for reusable logic. Prop drilling, a common challenge in passing data between components, is addressed, offering effective solutions. The lecture also covers the Context API, a powerful tool for simplified state management across an entire React application.

Custom hooks (something which we can write and other use it)

Two Jargons before we start:

1. Side effects
2. Hooks

Side effects

In react , the concept of side effects encompasses any operation any operations that reach outside the functional scope of a React component. These operations can affect other components, interest with the browser , or perform asynchronous data fetching.

In the context of React, side effects refer to operations or behaviors that occur outside the scope of the typical component rendering process. These can include data fetching, subscriptions, manual DOM manipulations, and other actions that have an impact beyond rendering the user interface.

Thus, "side effects" are the operations outside the usual rendering process, and "hooks," like useEffect, are mechanisms provided by React to handle these side effects in functional components. The useEffect hook allows you to incorporate side effects into your components in a clean and organized manner.

Example : setTimeout, fetch, setInterval

Hooks

Allow you to use state and other React features without writing a class.
They enable functional components to have access to stateful logic and lifecycle features, this lead to more concise and readable way of writing components in React.

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 common hooks are:

1. useState
2. useEffect
3. useCallback
4. useMemo
5. useRef
6. useContext

useState

Lets us describe the state of our application whenever state updates, it triggers a re-render which finally results in DOM update.

Example: Simple counter app (text inside the button is dynamic)

```
import { useState } from "react";
function App(){
  const [count,setCount]=useState(0)
  return <div>
    <button onClick={function(){
      setCount(count+1)
  }}>CLick me {count}</button>
  </div>
}
export default App;
```

useState() hooks is sufficient until we need to use the side effects.

useEffect

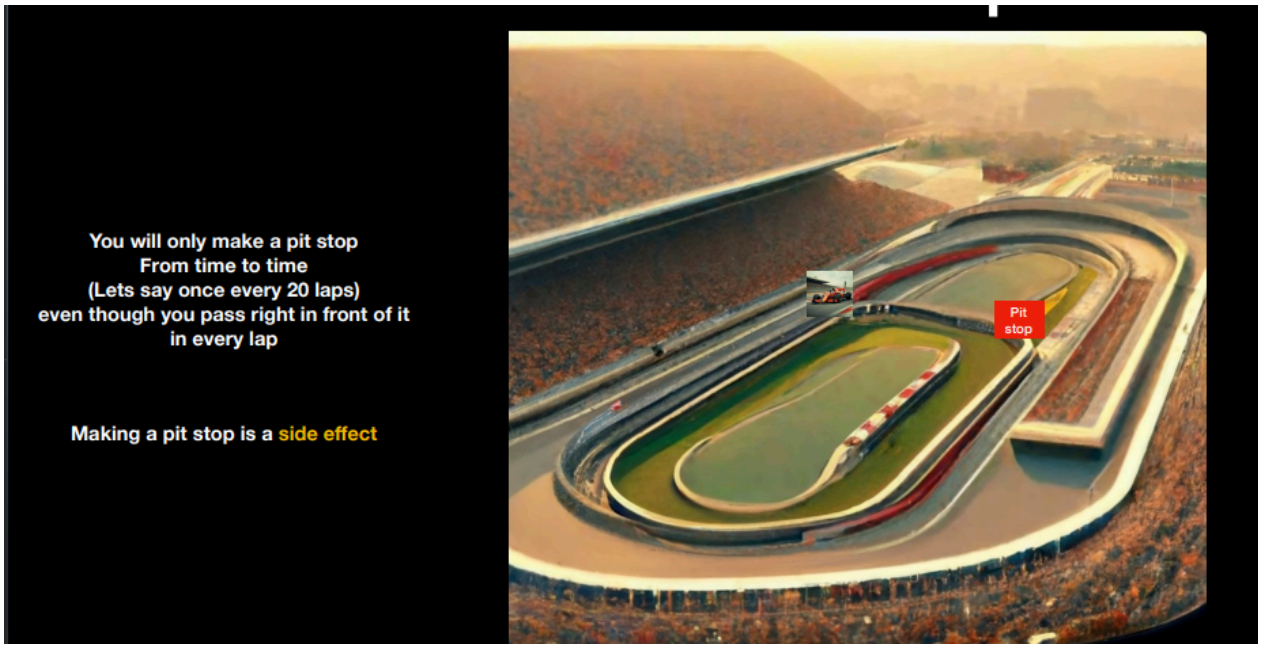
The `useEffect` hook is a feature in React, a popular JavaScript library for building user interfaces. It allows you to perform side effects in function components. Side effects are operations that can affect other components or can't be done during rendering, such as data fetching, subscriptions, or manually changing the DOM in React components.

The `useEffect` hook serves the same purpose as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` in React class components, but unified into a single API.

Side effects: example doing a backend call, using setTimeout() using setInterval().
When we want to fetch from backend example when components mount.

Let's start with an example

You are a car racer that has to do 100 laps across a stadium . You are allowed to take a pit stop from time to time. Do you take the stop in b/w every lap?? Or do you take a stop after every 10 laps lets say?



In this example the code inside the useEffect will run only one time

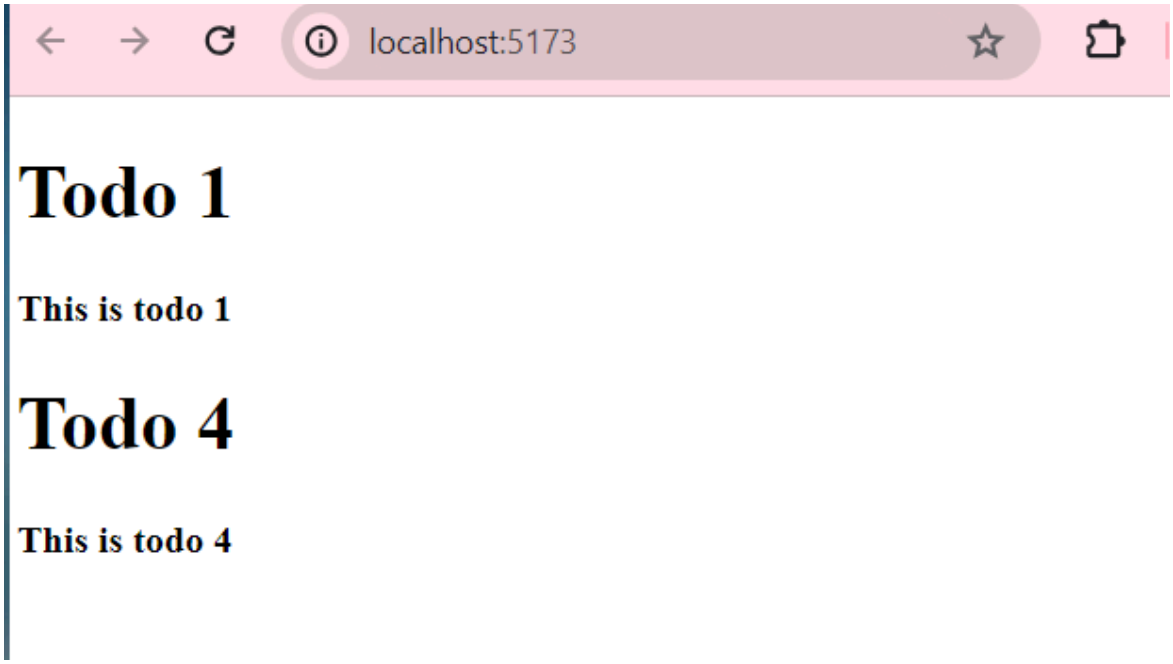
```
import { useState } from "react";
import { useEffect } from "react";

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

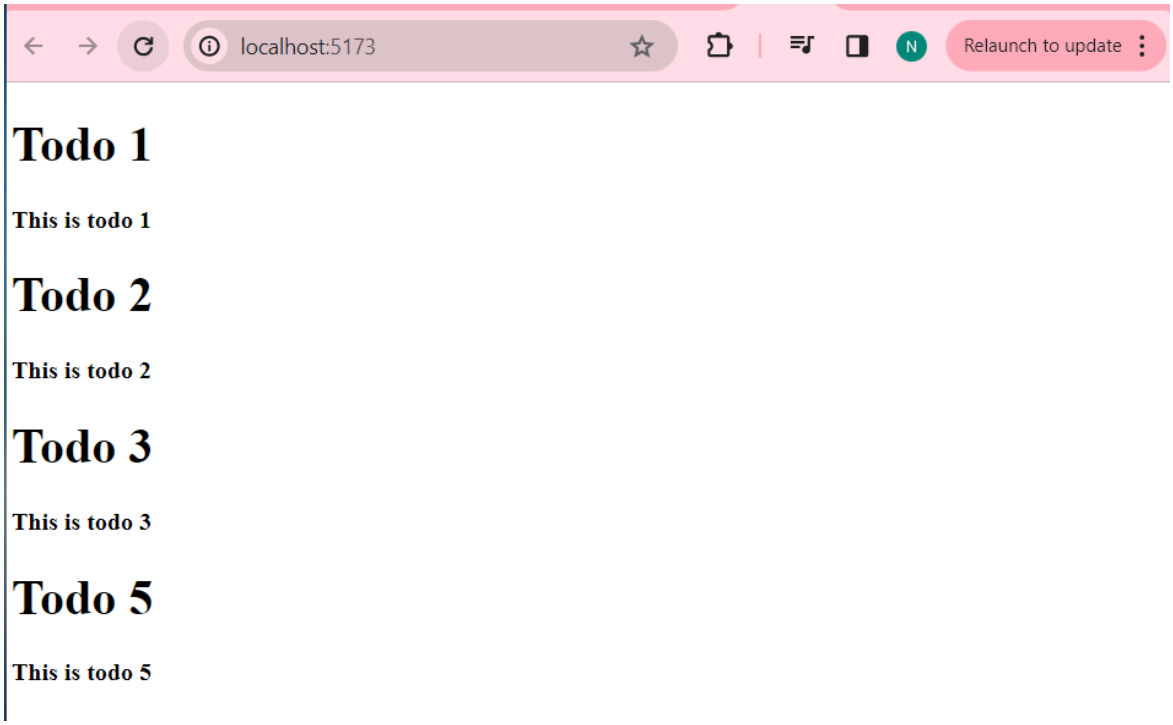
```
  useEffect(() => {
    fetch("https://sum-server.100xdevs.com/todos")
      .then(async function(res) {
        const json = await res.json();
        setTodos(json.todos);
      })
  }, [])

  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;
```

If we reload the page (calling App component for the first time) then the data show will change

The empty dependency array [] ensures that the effect runs only once after the initial render.

useEffect is a powerful tool for managing side effects in React components, providing a clean way to handle asynchronous operations and component lifecycle events.


localhost:5173
Todo 1
This is todo 1
Todo 4
This is todo 4


localhost:5173
Relaunch to update
Todo 1
This is todo 1
Todo 2
This is todo 2
Todo 3
This is todo 3
Todo 5
This is todo 5

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

What should happen?


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

When should it happen?

1. Tyre burst
2. Tyre pressure is up
3. 10 laps have passed
4. Engine is making a noise
5. Want to change the car

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

Dependency array

When should the callback fn run

When will we hit the pitstop:

1. Tyre burst
2. Tyre pressure is up etc

If there is any condition when we want to hit the specific code , we will put it inside the dependency array . It take **state variables** as the input.

Task

npm create vite@latest

npm install axios

Starting code:

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

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

  useEffect(()=>{

  },[])
```

```
  return (
    <>

    </>
  )
}

export default App();
```

Axios syntax to access backend code:

```
  useEffect(()=>{
    axios.get("https://sum-server.100xdevs.com/todos")
    .then(function(response){
      setTodos(response.data.todos)
    })
  },[])
```

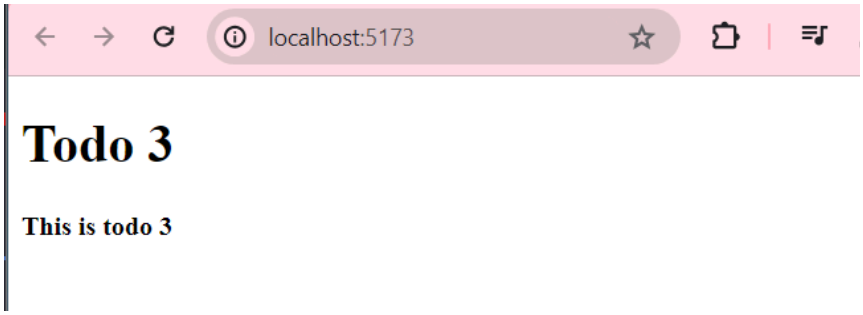
In axios we don’t have to use await. It already does the parsing for us

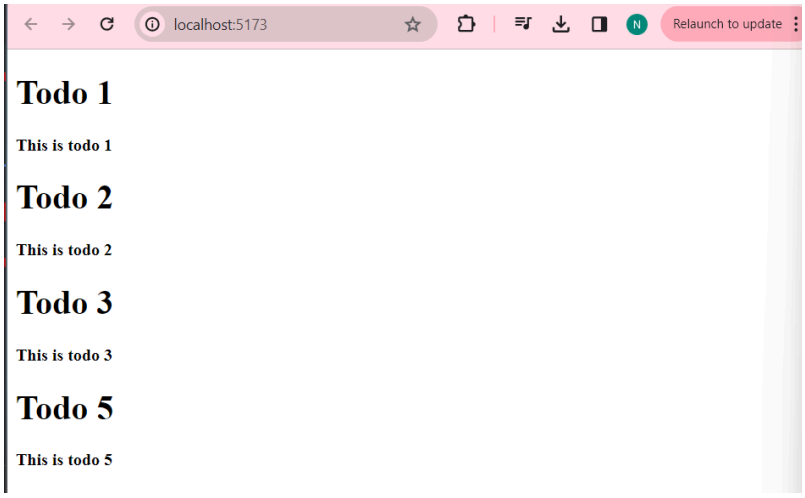
Solution

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

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

  useEffect(()=>{
    axios.get("https://sum-server.100xdevs.com/todos")
    .then(function(response){
      setTodos(response.data.todos)
    })
  },[]);

  return (
    <>
```

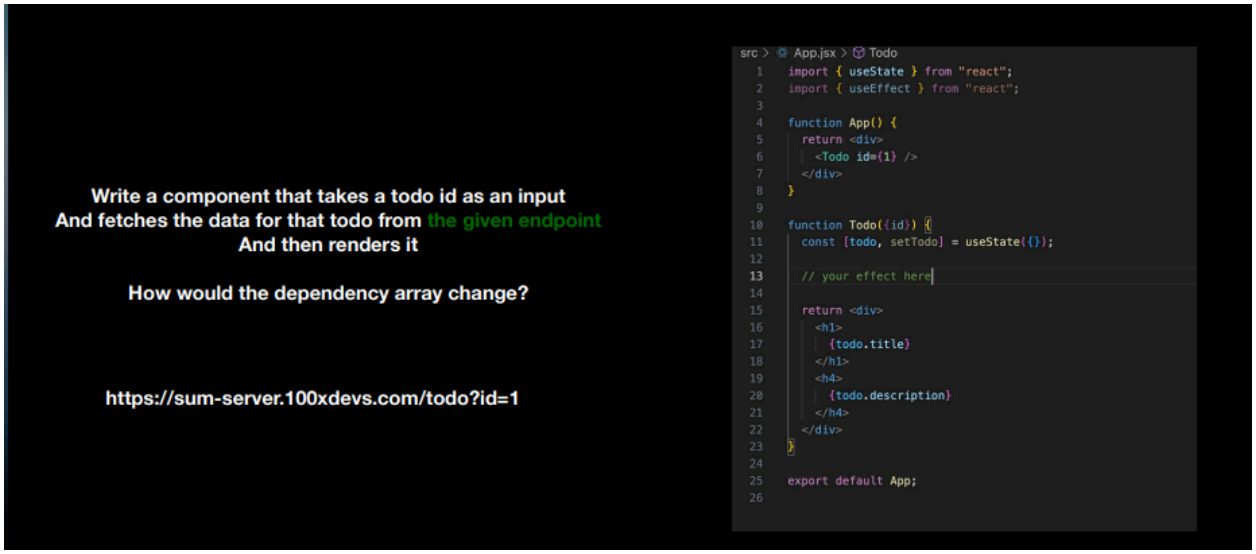
```
      {todos.map(todo => <Todo title={todo.title}
description={todo.description} />)}
    </>
  )
}

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

If we don’t use **useEffect()** hook then it will do infinite call





Now lets do the assignment



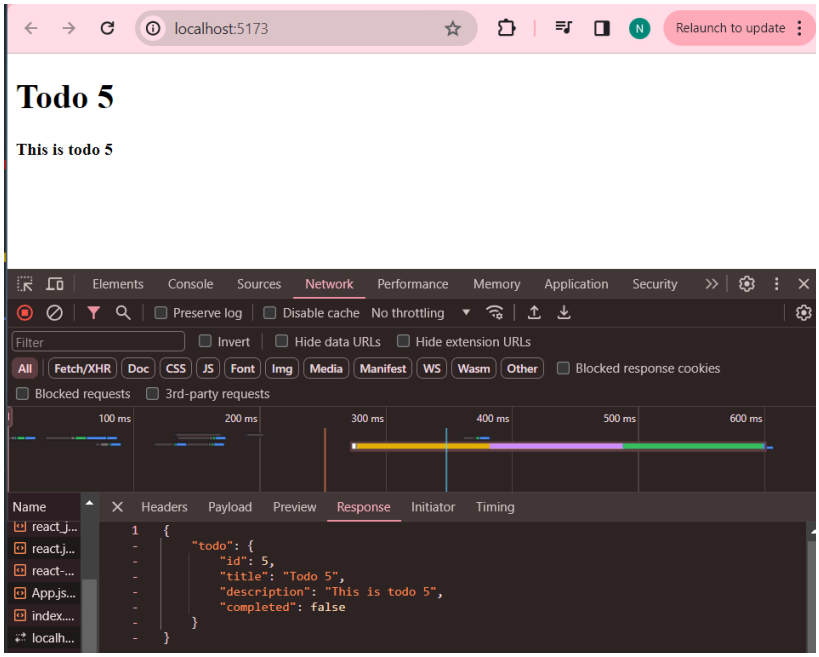
Initial Code:

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

export default function App(){
  const [todo,setTodo] = useState({});

  return (

  )
}

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

Final solution

```
import { useEffect, useState } from "react";
import axios from "axios";
export default function App(){
  return <div>
```

```
    <Todo id={5}/>
  </div>
}

function Todo({id}){
  const [todo,setTodo] = useState({});

  useEffect(()=>{
    axios.get("https://sum-server.100xdevs.com/todo?id=" + id)
    .then(response => {
      setTodo(response.data.todo)
      // what our backend returns
    })
  },[])

  return <div>
    <h1>{todo.title}</h1>
    <h4>{todo.description}</h4>
  </div>
}
```



Strict Mode

Strict Mode enables the following development-only behaviors:

Your components will re-render an extra time to find bugs caused by impure rendering.

Your components will re-run Effects an extra time to find bugs caused by missing Effect cleanup.

Your components will be checked for usage of deprecated APIs.

Q/Na

- useEffect(()=>{

  },[count===5])

Wrong way to write\

- bind. this in Javascript
- How to use sync await in useEffect()

useEffect(async ()=> {

  const response = await.axios(“https://sum-server.100xdevs.com”)

}

We can’t use directly use async await as top level function.

```
  async function main(){
    const response = await
axios.get("https://sum-server.100xdevs.com/todos")
    setTodos(response.data.todos)
  }
  useEffect(()=>{
    main()
  })
```

This is not good way since what if count changes and still backend is getting called and it is again called

.

- As we know that node_modules are heavy and more we uses more our application can get slow we use concept of tree shaking

Example : Axios is heavy suppose we need to use only get function from axios **import {get} from “axios”**

Now let implement another assignment

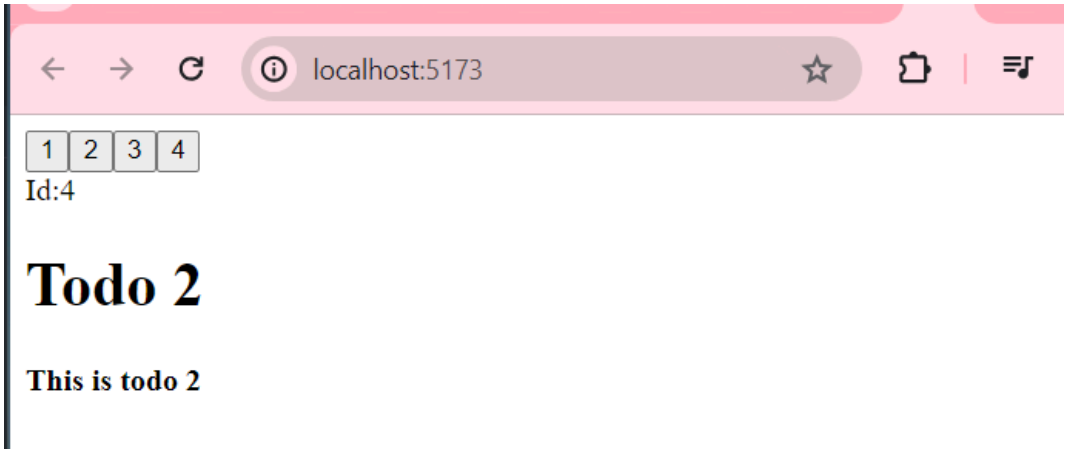
Wrong code:

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

export default function App(){
  const [currentId,setCurrentId] = useState(0)

  return <div>
    <button onClick={setCurrentId(1)}>1</button>
    <button onClick={setCurrentId(2)}>2</button>
    <button onClick={setCurrentId(3)}>3</button>
    <button onClick={setCurrentId(4)}>4</button>

    <Todo id={currentId}/>
  </div>
}
```

This result in infinite re-renders

```
import { useEffect, useState } from "react";
import axios from "axios";
// cleaner syntax to do the fetch call

export default function App(){
  const [currentId,setCurrentId] = useState(1)

  return <div>
    <button onClick={function(){
      setCurrentId(1)
    }}>1</button>
    <button onClick={function(){
      setCurrentId(2)
    }}>2</button>
    <button onClick={function(){
      setCurrentId(3)
    }}>3</button>
    <button onClick={function(){
      setCurrentId(4)
    }}>4</button>

    <Todo id={currentId}/>
  </div>
}

function Todo({id}){

  const [todo,setTodo] = useState({});

  useEffect(()=>{
    axios.get("https://sum-server.100xdevs.com/todo?id=" + id)
    .then(response => {
      setTodo(response.data.todo)
      // what our backend returns
    })
  },[])

  return <div>
```

```
    Id:{id}
    <h1>{todo.title}</h1>
    <h4>{todo.description}</h4>
  </div>
}
```



The problem with this code is that we are not calling backend when the value of id is changing on click the button.

This will fix the code:

```
  useEffect(()=>{
    axios.get("https://sum-server.100xdevs.com/todo?id=" + id)
    .then(response => {
      setTodo(response.data.todo)
      // what our backend returns
    })
  },[id])
```

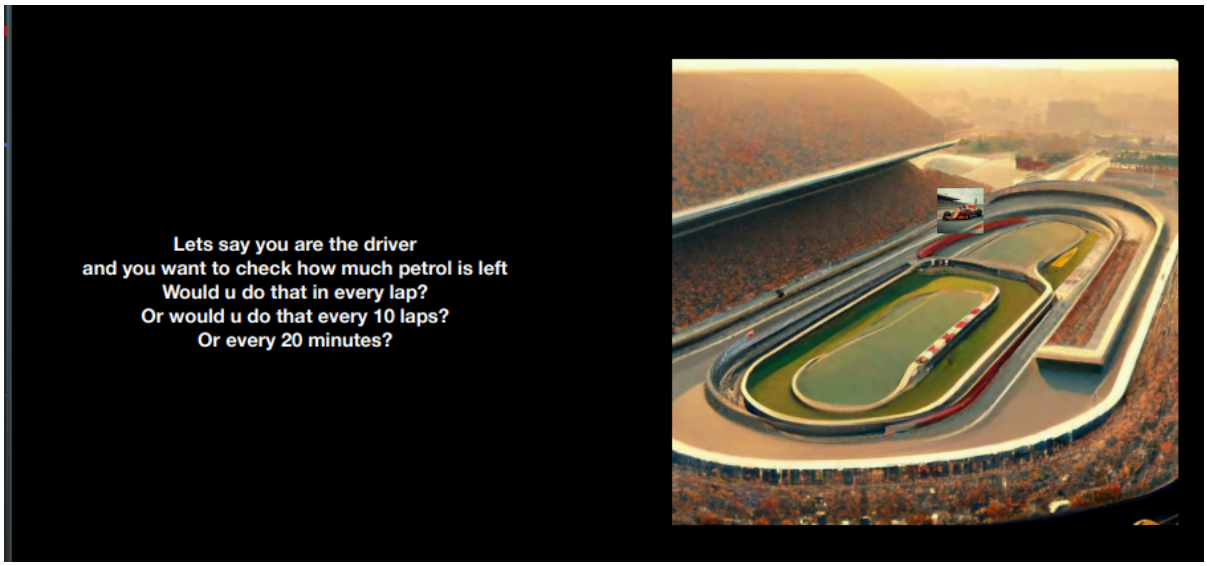
Axios now re-run anytime id changes.

Now lets say we want little bit delay in displaying todo we will use setInterval() function.

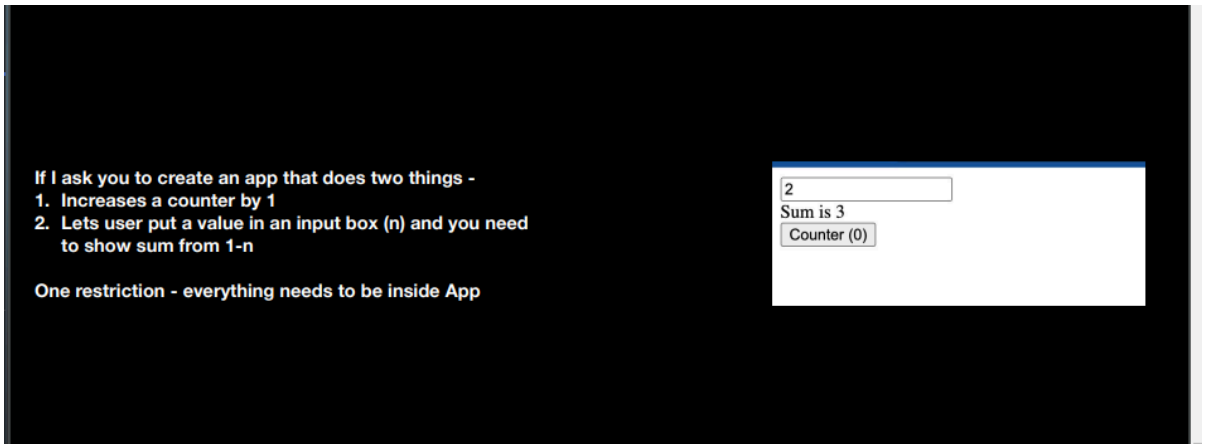
useMemo()

Let’s understand what memoization means

It means remembering some output given an input and not computing it again.



Lets try this assignment

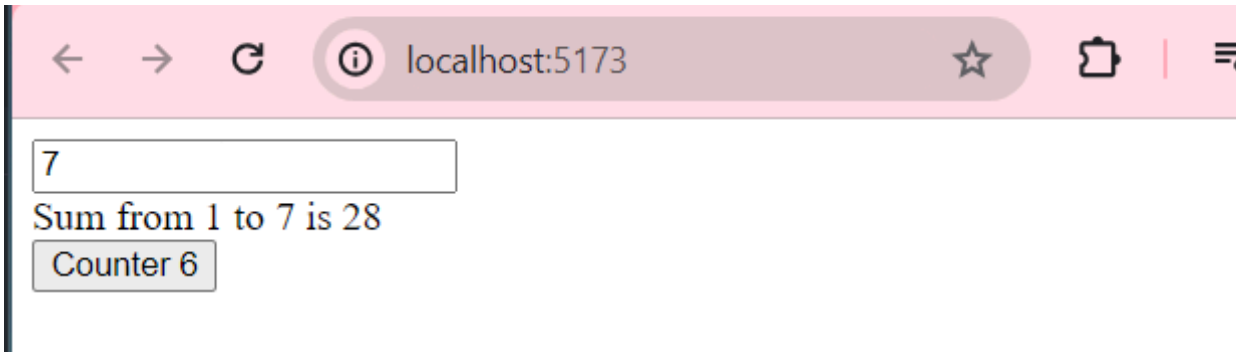


e.target.value will give us the actual input inside the input box

```
    <input onChange={function(e){
      setInputValue(e.target.value);
    }} placeholder="Find sum from 1 to n" />
```

Solution

```
import {  useState } from "react";
export default function App(){
  const [counter,setCounter]= useState(0);
  const [inputValue,setInputValue] = useState(1);

  let count =0;
  for(let i=1; i<= inputValue;i++){
    count = count + i;
  }

  return <div>
    <input onChange={function(e){
      setInputValue(e.target.value);
    }} placeholder="Find sum from 1 to n" />

    <br />
    Sum from 1 to {inputValue} is {count}
    <br />
    <button onClick={()=>{
      setCounter(counter+1);
    }}>Counter {(counter)}</button>
  </div>
}
```



This codebase is not optimal as for example we ,click on the button increase count state by 1, this will cause re-render

Which result in re-running of an expensive operation

```
 let count =0;
  for(let i=1; i<= inputValue;i++){
    count = count + i;
  }
```

Hence it is not optimal.

Lets see another approach

```
import {  useEffect, useState } from "react";
export default function App(){
  const [counter,setCounter]= useState(0);
  const [inputValue,setInputValue] = useState(1);
  const [finalValue,setFinalValue] = useState(0);

  useEffect(()=>{
    let count =0;
  for(let i=1; i<= inputValue;i++){
    count = count + i;
  }
  setFinalValue(count);
  },[inputValue])

  return <div>
    <input onChange={function(e){
      setInputValue(e.target.value);
    }} placeholder="Find sum from 1 to n" />

    <br />
    Sum from 1 to {inputValue} is {finalValue}
    <br />
    <button onClick={()=>{
      setCounter(counter+1);
    }}>Counter {(counter)}</button>
  </div>
}
```

In this approach we have used the useEffect.

Lets see another approach to this assignment more optimized

We will use hook **useMemo** because in the above approach we had defined a state variable which completely depend on the other state variable and useMemo also decrease computation time

```
import {  useState,useMemo } from "react";
export default function App(){
  const [counter,setCounter]= useState(0);
  const [inputValue,setInputValue] = useState(1);

  let count = useMemo(()=>{
    let finalCount = 0;
    for(let i=1; i<= inputValue; i++){
      finalCount = finalCount +i;
    }
    return  finalCount;
  }, [inputValue])

  return <div>
    <input onChange={function(e){
      setInputValue(e.target.value);
    }} placeholder="Find sum from 1 to n" />

    <br />
    Sum from 1 to {inputValue} is {count}
    <br />
    <button onClick={()=>{
      setCounter(counter+1);
    }}>Counter {(counter)}</button>
  </div>
}
```

useCallback()

‘**useCallback’** is a hook in React, a popular JavaScript library for building user interfaces. It is used to memoize functions, which can help in optimizing the

performance of your application, especially in cases involving child components that rely on reference equality to prevent unnecessary renders.

var a = 1;
var b = 1;
a == b; // true
Are they referentially equal??
No

Lets see another example:
function sum(a, b) {return a + b}
function sum2(a, b) {return a + b}
sum == sum2
// false
Even though they are exactly the same , but they are not referentially same , they are not in same places in memory.

When does React re-render??

```
import {  useState} from "react";
export default function App(){
  const [counter,setCounter]= useState(0);
  var a=1;

  return <div>
    <button onClick={()=>{
      setCounter(counter+1);
    }}>Counter {(counter)}</button>
    <Demo a={a} />
  </div>
}

const Demo = memo(function({a}){
```

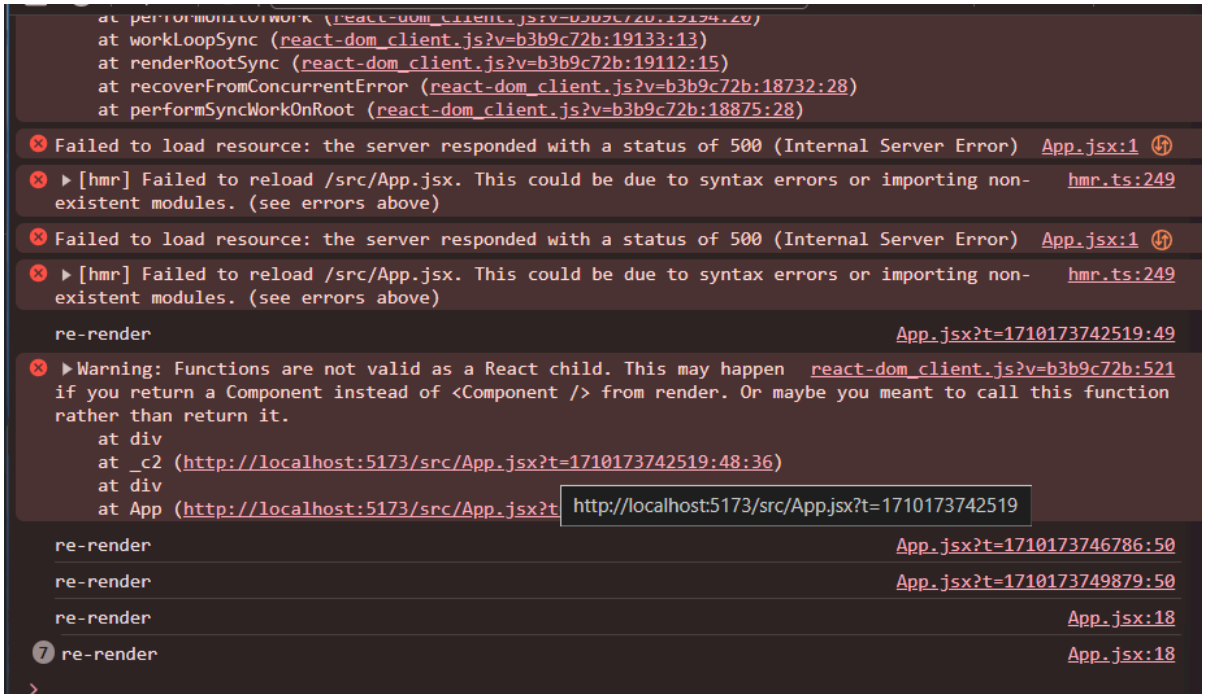
```
  console.log("re-render")
  return <div>
    hi there
  </div>
})
```

In this example should the **Demo** component re-render when we will click on the button ? We have used memo on Demo component hence it will re-render only if something changes in **Demo** component.

It does not re-render .

```
import { memo, useState} from "react";
export default function App(){
  const [counter,setCounter]= useState(0);

  function a(){
    console.log("hi there");
  }

  return <div>
    <button onClick={()=>{
      setCounter(counter+1);
    }}>Counter {(counter)}</button>
    <Demo a={a} />
  </div>
}

const Demo = memo(function({a}){
  console.log("re-render")
  return <div>
    hi there {a}
  </div>
})
```

This will cause re-render



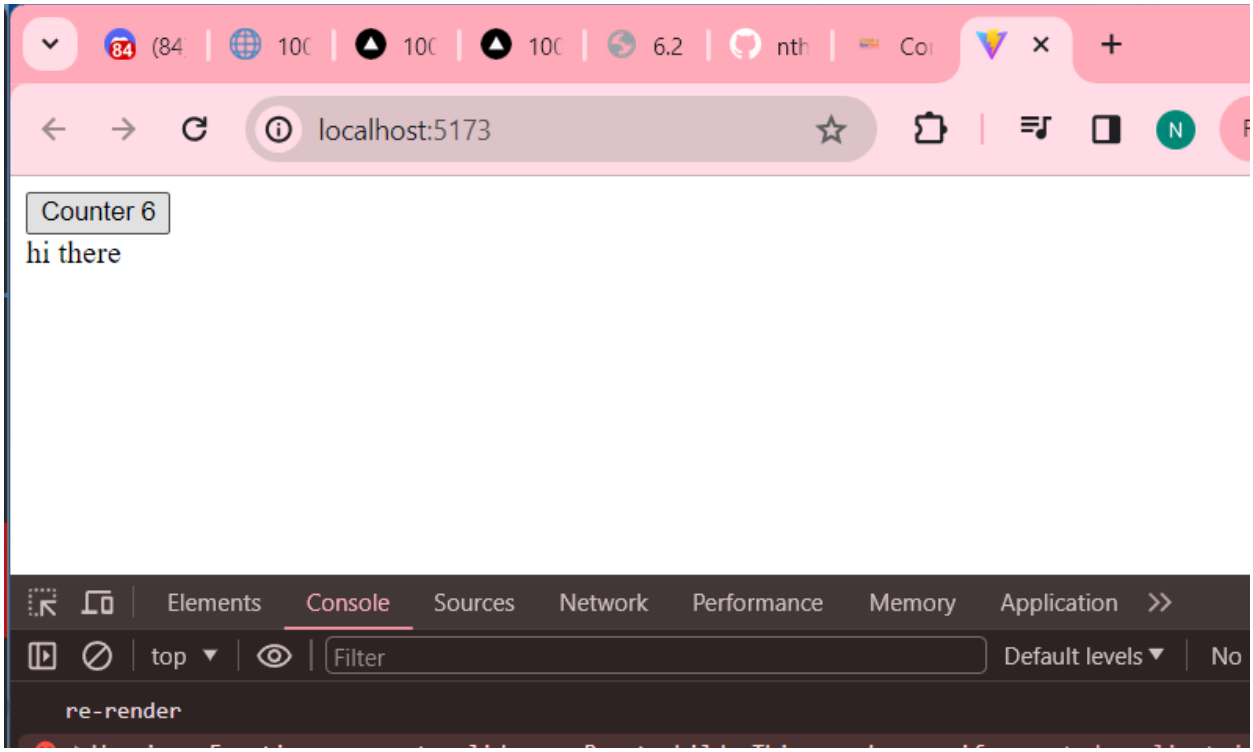
useCallback is used to memoize function which can help in optimizing the performance of your application especially in cases involving child components that rely on reference equality to prevent unnecessary renders.

var a = { }

var b = { }

a == b //false

```
var a = useCallback(function(){
   console.log("hi there ");
 },[])
```



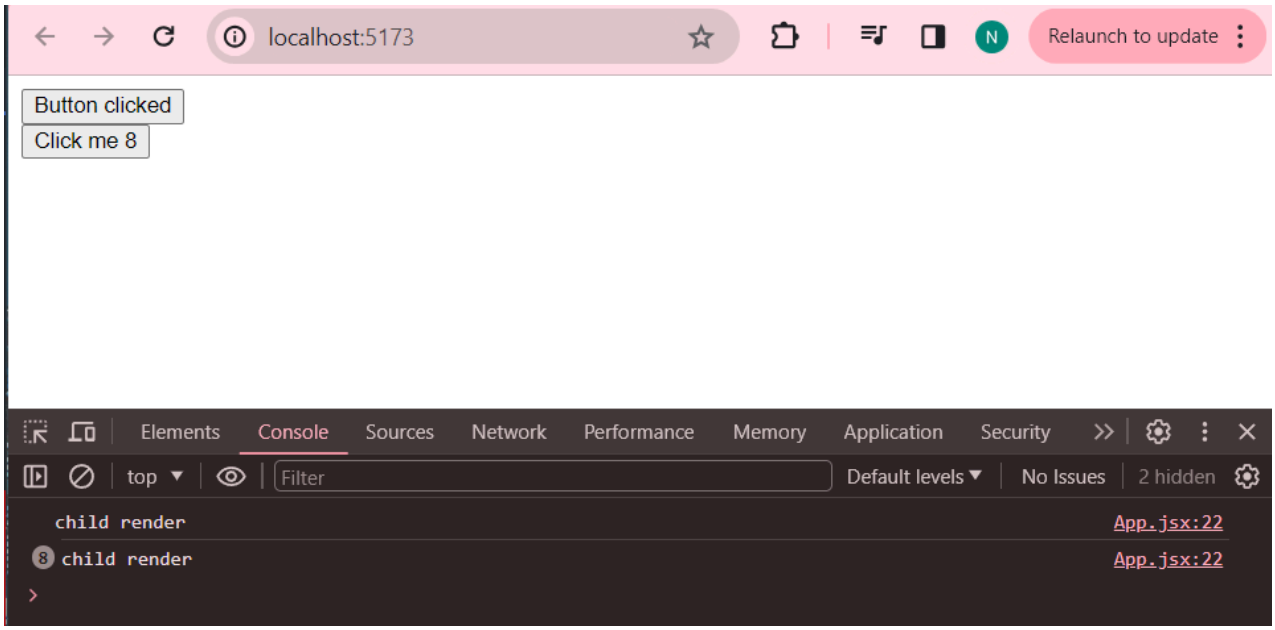
What is problem in this code??

```
import { memo, useState } from "react";

function App() {
  const [count, setCount] = useState(0)

  function onClick() {
    console.log("child clicked")
  }

  return <div>
    <Child onClick={onClick} />
    <button onClick={() => {
      setCount(count + 1);
    }}>Click me {count}</button>
  </div>
}

const Child = memo(({onClick}) => {
  console.log("child render")
```

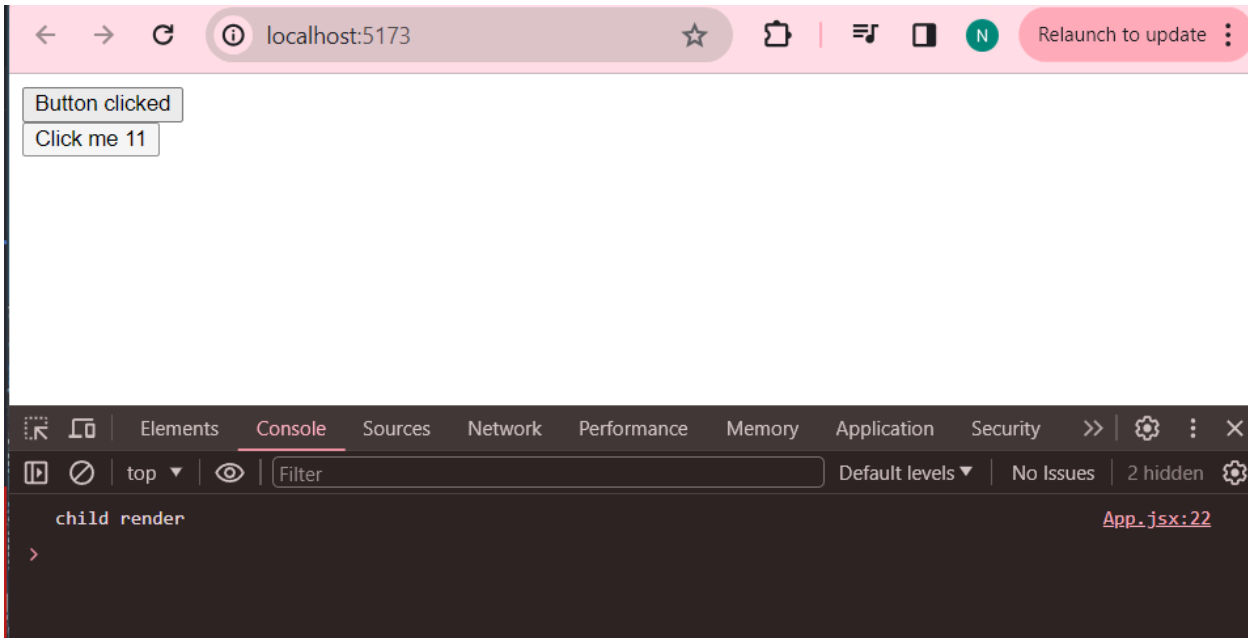
```
  return <div>
    <button onClick={onClick}>Button clicked</button>
  </div>
})

export default App;
```

```
import { memo, useState } from "react";

function App() {
  const [count, setCount] = useState(0)

  function logSomething() {
    console.log("child clicked")
  }
  // native function , not a state variable, function is constant

  return <div>
    {/* button component and passing props */}
    <ButtonComponent inputFunction={logSomething} />
    <button onClick={() => {
      setCount(count + 1);
    }}>Click me {count}</button>
  </div>
}

const ButtonComponent = memo(({inputFunction}) => {
  // if we don't add memo child will 100% re-render if we change parent
  console.log("child render")
  return <div>
    <button>Button clicked</button>
  </div>
})

export default App;
```

If we click on the button , ideally it should not re-render because we are using memo but the problem is that function1 == function2 will be false even if the body of the both the function is same , this will result in re-render.

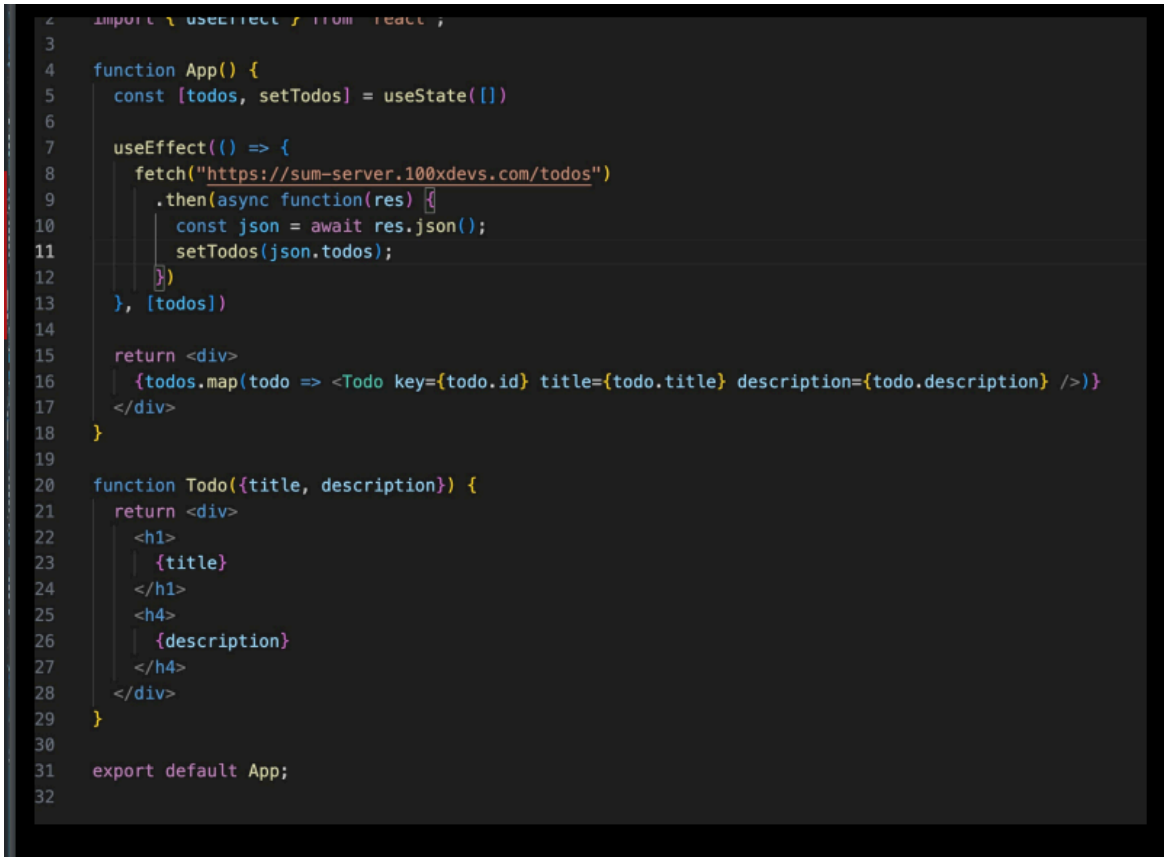


To prevent this from re-rendering we will use the inputFunction as a variable which is a function wrapped inside useCallback().

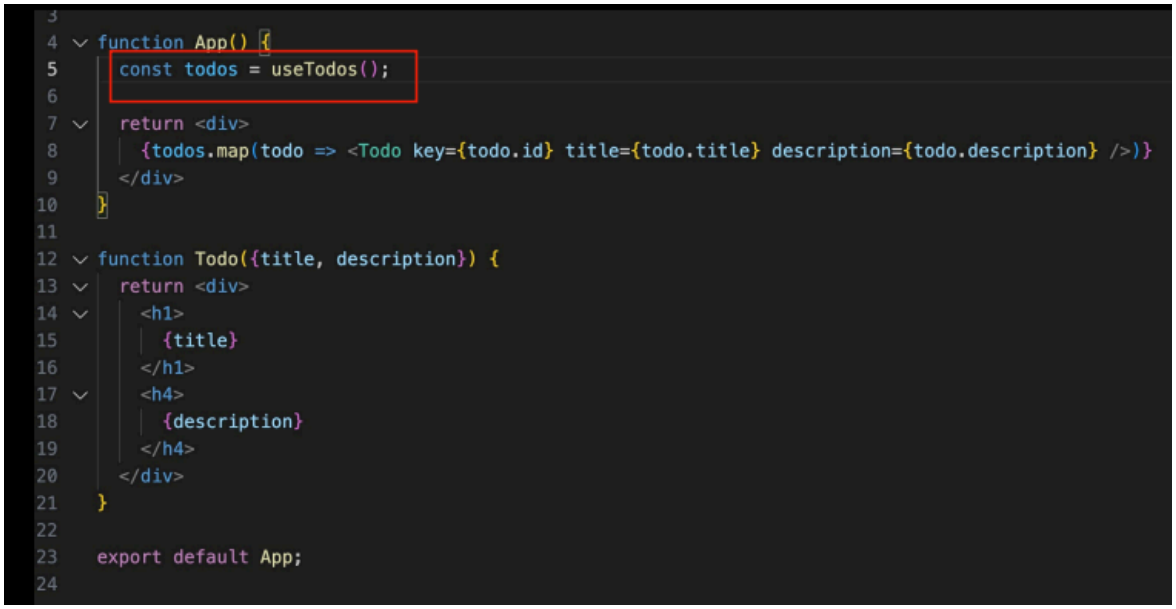


Custom Hooks

We can write our own hook , it must start with use (naming convention).



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

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

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

  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;
```



```
function App() {
  const todos = useTodos();

  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;
```

Making our own hook in this case is very beneficial to someone who doesnt handle http code or make backend calls.

We cannot use hooks or useState in a raw function , it should be hook or a component

Q/Na:

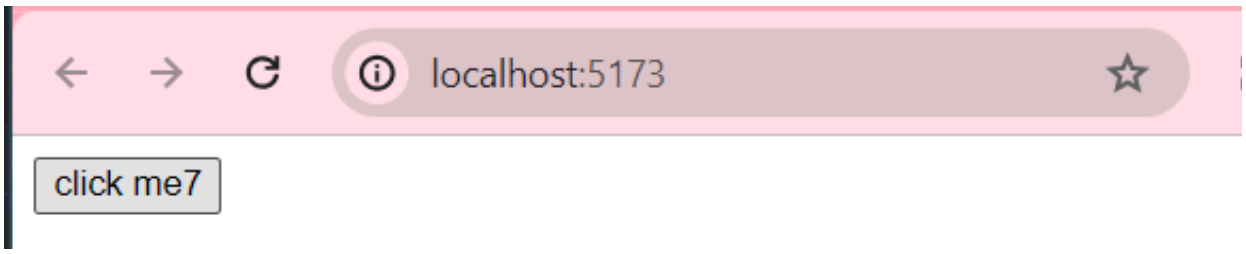
1. const [todos, setTodos] = useState([]);
2. const filteredTodos = useMemo(todos.find((x) => x.id % 2 ==0));

Using useMemo will help that it wont re-render when other state variable changes.

3. How this code works

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

function App(){
  const [count, setCount] = useState(0);
  return <div>
    <button onClick={()=>{
      setCount(count + 1);
      setCount(count + 1);
    }}>click me{count}</button>
  </div>
}

export default App;
```

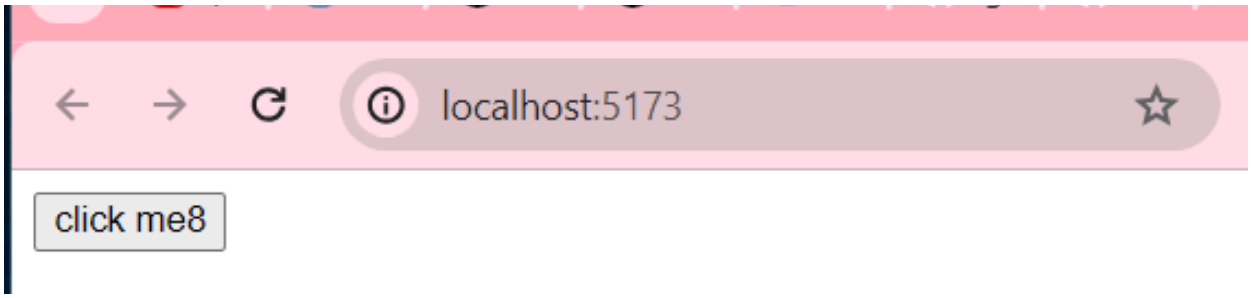
It will increase count by only one



Count variable does not changes immediately. Right way to do is to pass it as a function

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

function App(){
  const [count, setCount] = useState(0);
  return <div>
    <button onClick={()=>{
      setCount(c => c + 1);
      setCount(c => c + 1);
    }}>click me{count}</button>
  </div>
}

export default App;
```

We are giving you function and saying whenever you are getting state variable increase the function by one.



4. Lets try to understand whatsapp encryption

Supppose your friend has a locker and want to send to you. Your friend will send you the locker with the lock whose keys he has , then you will send back the locker to your friend with new lock whose keys you have and then your friend will send you the locker back , after opening his lock and then you will open your lock and access the locker.

5. Use memo cant take async function as input hence we can do fetch call
6. We can memoized object by useMemo().
7. const inputFunction = useCallback(()=>{
   console.log(“hiThere”)
   },[])

We may think that when if useCallback is called again then the reference of the inputFunction will change and this will create some problem

8. Why isn’t this running infinitely??

```
const handleClick = useCallback(()=>{
        setCount(count+1);
},[count])
return <div>
        <Child onClick = {handleClick}/>
</div>
```

It it not running infinitely because anytime the count changes it is changing the signature of the handleClick , its not actually running setCount again and again

9. While using useCallback() we must define array dependencies correctly and carefully.