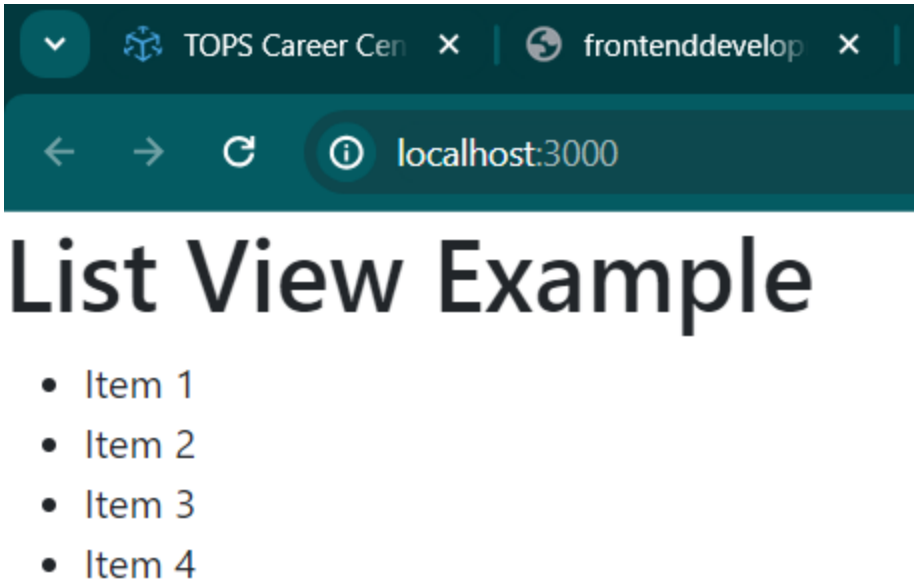
# Lists and Hooks

## Lists:-

src > JS App.js > App

```
1   import React from "react";
2   import Listview from "./Listview";
3
4   function App() {
5       const dataList = [
6           { id: 1, text: 'Item 1' },
7           { id: 2, text: 'Item 2' },
8           { id: 3, text: 'Item 3' },
9           { id: 4, text: 'Item 4' },
10        ];
11    return (
12      <div>
13        <h1>List View Example</h1>
14       <Listview data={dataList} />
15      </div>
16    );
17  }
18
19  export default App;
20
```

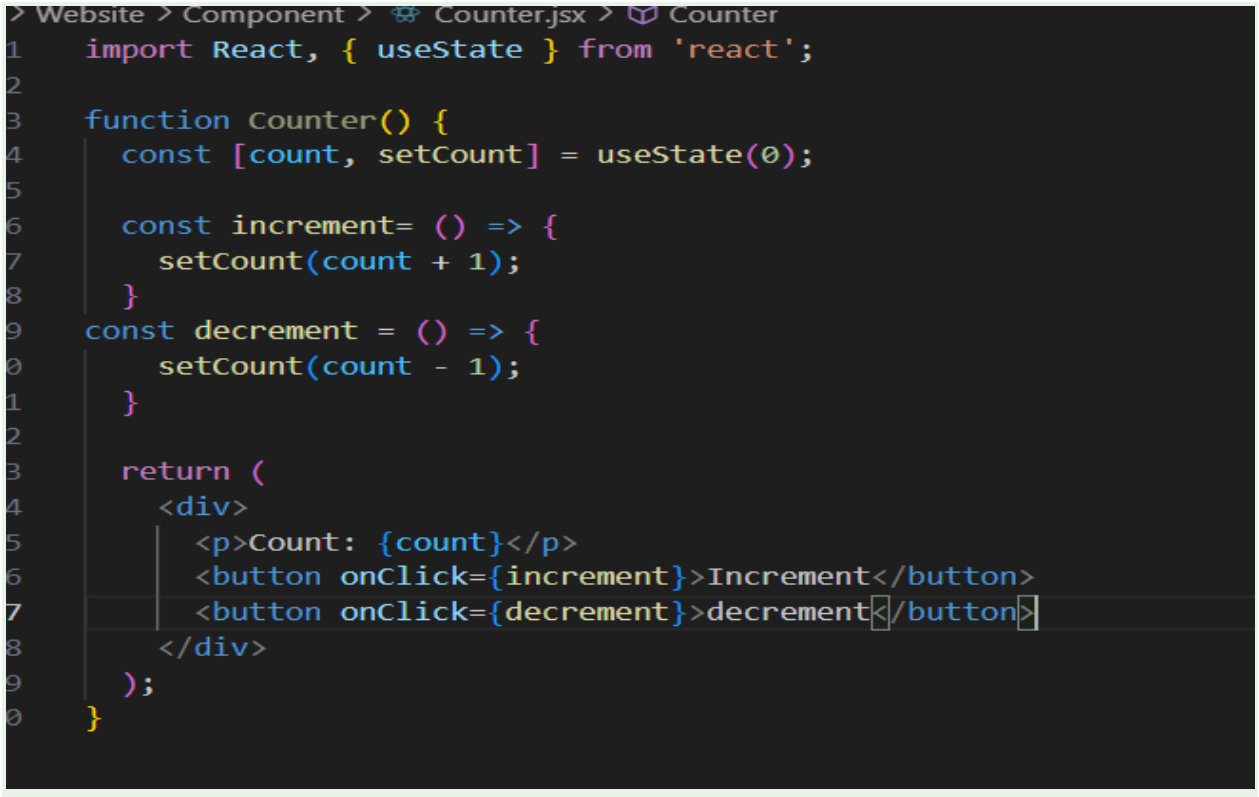



**Hooks:-**React hooks are a powerful feature in React that allow you to add state and other features to functional components. They were introduced in React 16.8 and have since become an integral part of React development. In this blog, we'll explore some of the most commonly used React hooks and provide easy-to-understand examples.

**1: useState hook**

The useState hook allows you to add state to a functional component. It takes an initial value as an argument and returns an array with two elements: the current state value and a function to update it.

Here's an example of how to use useState to add a counter to a functional component:



## 2: useEffect hook

The useEffect hook allows you to perform side effects in a functional component. Side effects include things like fetching data from an API, updating the DOM, or subscribing to an event.

Here’s an example of how to use useEffect to fetch data from an API:

```jsx
import React, { useState, useEffect } from 'react';

function DataFetcher() {
  const [data, setData] = useState([]);

  useEffect(() => {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setData(data));
  }, []);

  return (
    <ul>
      {data.map(item => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
}
```

## 3: useContext hook

The useContext hook allows you to access a context object in a functional component. Context is a way to pass data down the component tree without having to pass props manually.

Here's an example of how to use useContext to access a theme context:

```
import React, { useContext } from 'react';

const ThemeContext = React.createContext('light');

function ThemeButton() {
  const theme = useContext(ThemeContext);

  return (
    <button style={{ background: theme === 'dark' ? 'black' : 'white', color: theme === 'dark' ? 'white' : 'black' }}>
      Toggle Theme
    </button>
  );
}
```

## 4: useCallback hook

The useCallback hook allows you to memoize a function so that it's only re-created when its dependencies change. This can help improve performance by preventing unnecessary re-renders.

Here's an example of how to use useCallback to memoize a function:


## 5: useMemo hook

The useMemo hook allows you to memoize a value so that it's only re-computed when its dependencies change. This can help improve performance by preventing unnecessary re-computations.

Here's an example of how to use useMemo to memoize a calculated value:

```
import React, { useState, useMemo } from 'react';

function ExpensiveCalculation({ a, b }) {
  const result = useMemo(() => {
    console.log('Calculating...');
    return a * b;
  }, [a, b]);

  return <p>Result: {result}</p>;
}
```

## 7: useRef hook

The useRef hook allows you to create a mutable ref object that persists for the lifetime of the component. You can use it to store and access values that don't trigger a re-render.

Here's an example of how to use useRef to access the value of an input element:

```jsx
import React, { useRef } from 'react';

function InputWithFocus() {
  const inputRef = useRef();

  const handleClick = () => {
    inputRef.current.focus();
  }

  return (
    <div>
      <input type="text" ref={inputRef} />
      <button onClick={handleClick}>Focus Input</button>
    </div>
  );
}
```