DEBOUNCING

THROTTLING

RACE CONDITION

DEBOUNCING

Debouncing is a technique used to limit the frequency of a function call, particularly useful for handling events that fire rapidly, such as scroll or resize events. It ensures that a function is only executed after a certain amount of time has passed since the last invocation of the function.

- Reducing unnecessary API calls or expensive operations triggered by rapid events.

- Improving performance by reducing the workload on the browser.

DEBOUNCING

Use Cases

- **Input fields with live search**: When users type in an input field, you can debounce the search action so that the search is triggered only after they have finished typing or after a certain delay.

- **Window resize or scroll events**: Debouncing these events helps optimize the performance by reducing the number of event-handling operations.

```
const useDebounce = (value, delay) => {
  const [debouncedValue, setDebouncedValue] =
useState(value);

  useEffect(() => {
    const timer = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => {
      clearTimeout(timer);
    };
  }, [value, delay]);

  return debouncedValue;
}
```

```jsx
const App = () => {
  const [searchTerm, setSearchTerm] =
useState('');
  const debouncedSearchTerm =
useDebounce(searchTerm, 500);

  useEffect(() => {

    console.log('Searching for:', debouncedSearchTerm);
  }, [debouncedSearchTerm]);

  const handleChange = (event) => {
    setSearchTerm(event.target.value);
  };

  return (
    <input
      type="text"
      value={searchTerm}
      onChange={handleChange}
      placeholder="Search..."
    />
  );
}
```

THROTTLING

Throttling is another technique that limits the rate at which a function can be executed. It ensures that a function is executed at most once within a specified time interval, even if it is triggered multiple times.

- Preventing excessive resource consumption caused by rapid event triggering.
- Improving user experience by enforcing controlled interaction rates.

THROTTLING

Use Cases

- **Scroll or mouse move events**: Throttling these events ensures that the associated actions, such as updating UI elements or performing calculations, are executed at a controlled rate to avoid overwhelming the system.

- **Button clicks or form submissions**: Throttling can prevent rapid consecutive clicks or submissions that may cause unintended side effects.

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

const useThrottle = (value, delay) => {
  const [throttledValue, setThrottledValue] = useState(value);
  const [lastExecutedTime, setLastExecutedTime] = useState(Date.now());

  useEffect(() => {
    const timer = setTimeout(() => {
      const now = Date.now();
      const timeSinceLastExecution = now - lastExecutedTime;

      if (timeSinceLastExecution >= delay) {
        setThrottledValue(value);
        setLastExecutedTime(now);
      }
    }, delay);

    return () => {
      clearTimeout(timer);
    };
  }, [value, delay, lastExecutedTime]);

  return throttledValue;
};
```

```jsx
const App = () => {
  const [scrollPosition, setScrollPosition] = useState(0);
  const throttledScrollPosition = useThrottle(scrollPosition, 200);

  useEffect(() => {
    console.log("Scroll position:", throttledScrollPosition);
  }, [throttledScrollPosition]);

  const handleScroll = (event) => {
    setScrollPosition(event.target.scrollTop);
  };

  return (
    <>
      scrollPosition : {scrollPosition}
      <br />
      throttledScrollPosition : {useThrottle(scrollPosition, 200)}
      <div
        style={{
          height: "200px",
          overflow: "auto",
          backgroundColor: "#282c34",
        }}
        onScroll={handleScroll}
      >
        A text that needs more than 200px height
      </div>
    </>
  );
};
```

RACE CONDITION

A race condition occurs when the behavior of a program is dependent on theordering and timing of events, and the outcome of the program becomes unpredictable when those events occur concurrently or in a different order than expected.
In the context of React, race conditions can occur when multiple asynchronous operations are triggered, such as API calls or state updates, and their completion order affects the final result or behavior of the component.

```jsx
const Counter = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    async function fetchData() {
      const response = await fetch("https://api.example.com/data");
      const data = await response.json();
      setCount(data.count);
    }

    fetchData();
  }, []);

  const increment = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <button onClick={increment}>Increment</button>
      <p>Count: {count}</p>
    </div>
  );
};
```

RACE CONDITION

imagine the following scenario:

1. The component mounts, and the API call is initiated.
2. Before the API call completes, the user clicks the "Increment" button.
3. The increment function is called, and the count state is updated by incrementing its current value.
4. Shortly after, the API call completes, and the count state is updated with the fetched value.

RACE CONDITION

In this scenario, the race condition occurs because the value fetched from the API is overwritten by the incremented value. As a result, the user's action to increment the count is effectively lost.
To address race conditions, you can use techniques such as synchronizing asynchronous operations, implementing proper locking mechanisms, or using atomic state updates. In the case of the example above, you can use the useState setter function with a callback to ensure the latest value is used in the state update:

```
const increment = () => {
  setCount((prevCount) => prevCount + 1);
};
```

Thank you for taking the time to read this document. Your attention and engagement are greatly appreciated.

❤️❤️