JavaScript Debouncing

Strategy to Enhance Performance

What is Debouncing?

Debouncing is a technique used to **optimize performance** when handling events that are triggered frequently, like window resizing, scrolling, or typing in a search input.

This can be useful for scenarios where we want to avoid unnecessary or repeated function calls that might be expensive or time-consuming.

How Debouncing Works

User Triggered an Event
(e.g., typing in a search box).

Delay Timer Set:
A timer is started.

Reset Timer on New Event:
If the event fires again before the timer finishes, the previous timer is cleared and reset.

Execute Function:
After the event stops firing for a set period (like 500ms), the function is executed.

Benefits of Debouncing

- **Improved performance:** It reduces the number of function executions, making the app more efficient.

- **Better user experience:** Prevents excessive and unnecessary function calls, leading to smoother interactions.

- **Reduced server load:** Without debouncing, an API request would be made on every keystroke, which could overload the server.

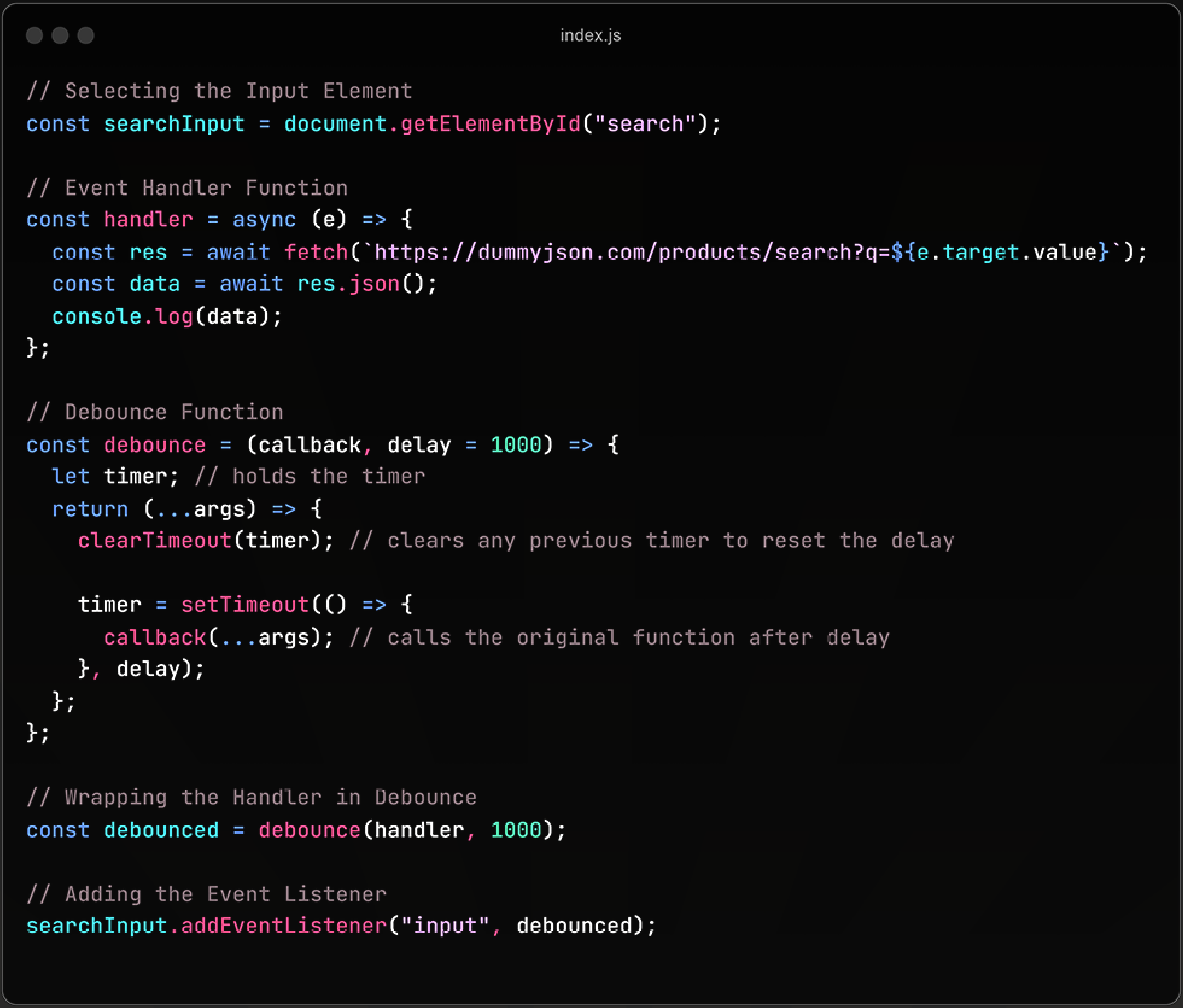
Let's implement debouncing

Debouncing Search Input

If you want to fetch search suggestions or filter results as the user types, but not trigger a request on every keystroke.

index.html

```html
<!-- HTML Input Element -->
<input type="text" id="search" placeholder="Search..." />
```



index.js

```javascript
// Selecting the Input Element
const searchInput = document.getElementById("search");

// Event Handler Function
const handler = async (e) => {
  const res = await fetch(`https://dummyjson.com/products/search?q=${e.target.value}`);
  const data = await res.json();
  console.log(data);
};

// Debounce Function
const debounce = (callback, delay = 1000) => {
  let timer; // holds the timer
  return (...args) => {
    clearTimeout(timer); // clears any previous timer to reset the delay

    timer = setTimeout(() => {
      callback(...args); // calls the original function after delay
    }, delay);
  };
};

// Wrapping the Handler in Debounce
const debounced = debounce(handler, 1000);

// Adding the Event Listener
searchInput.addEventListener("input", debounced);
```

Explanation:

Selecting the Input Element : This selects the <input> element with the id="search" and stores it in the searchInput variable.

Event Handler Function: handler is an asynchronous function that Fetches data from https://dummy-json.com/products/search?q=<userInput>.

- Converts the response into JSON format.
- Logs the fetched data to the console.

Debounce Function: Ensures that handler does not run on every keystroke.

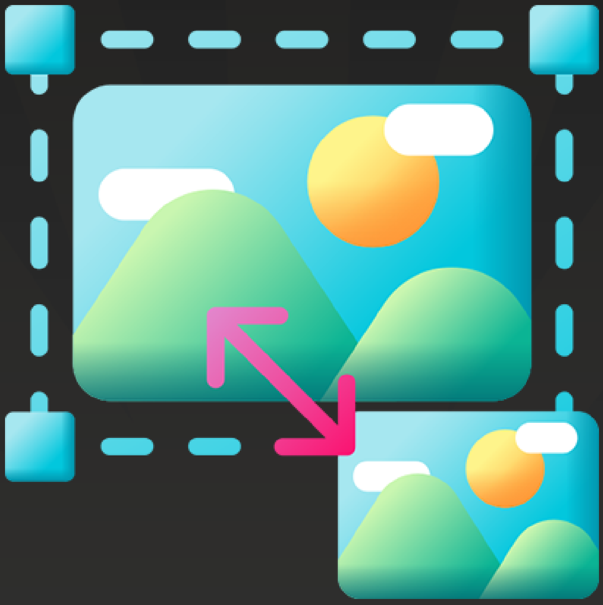
- If a user types, the previous timer is cleared (clearTimeout(timer)).
- A new timer starts with a delay of 1000ms
- When the user stops typing for 1 second, callback (i.e., handler) is executed.

Wrapping the Handler in Debounce: This creates a debounced version of handler, delaying execution by 1000ms after the last keystroke.

Adding the Event Listener: Attaches the debounced function to the input field that will only trigger **after the user stops typing for 1 second**, reducing unnecessary API calls.

Debouncing Window Resize

Optimize performance by debouncing window resize events—update the layout only after the user stops resizing, not during every change!



index.js

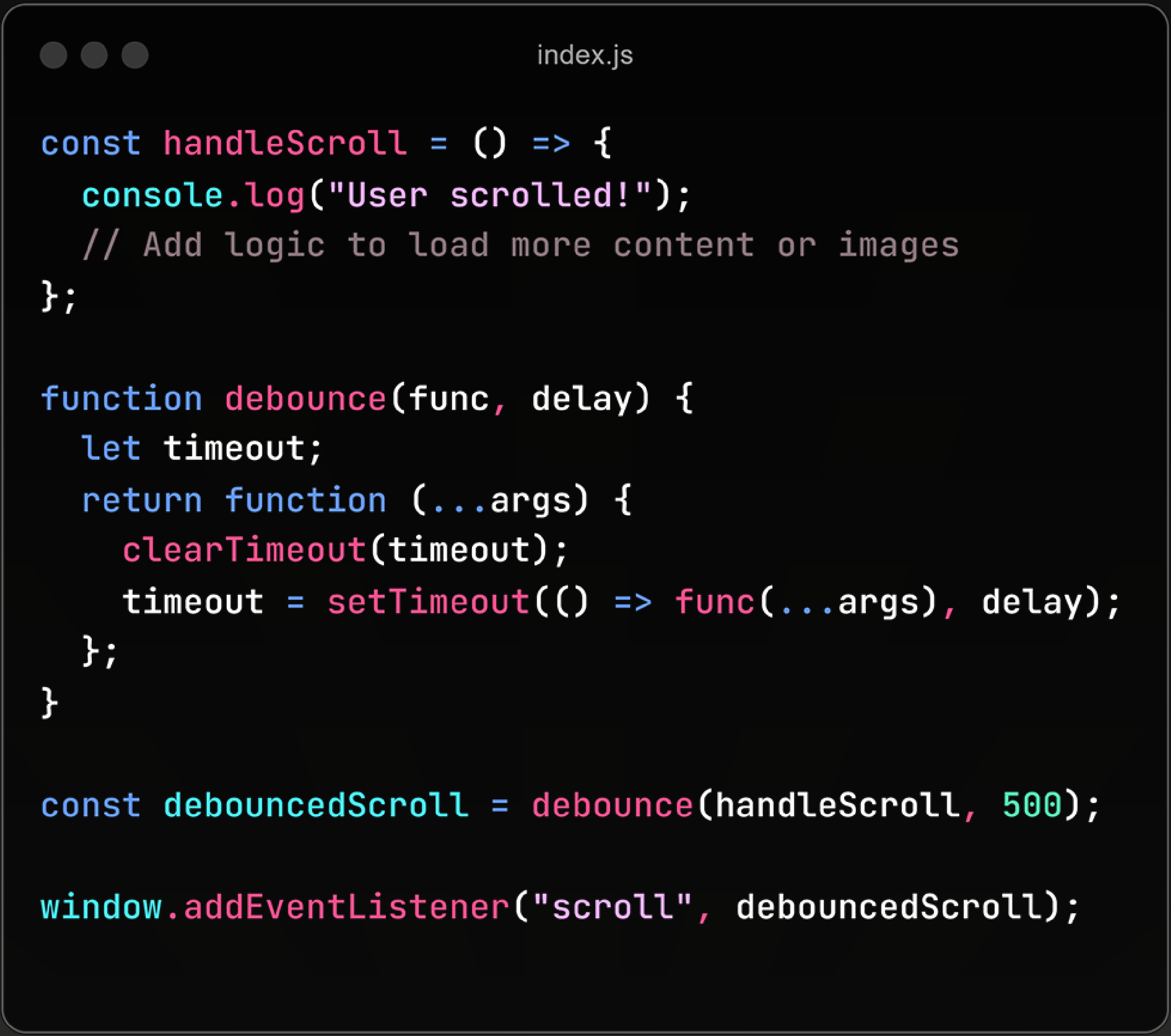
```javascript
// Simulate resizing logic
const handleResize = () => {
  const width = window.innerWidth;
  const height = window.innerHeight;
  console.log("width:", width);
  console.log("height:", height);
};

// Debounce Function
function debounce(func, delay) {
  let timeout;
  return function (...args) {
    clearTimeout(timeout);
    timeout = setTimeout(() => func(...args), delay);
  };
}

// Apply debounce to the resize handler
const debouncedResize = debounce(handleResize, 1000);

// Attach the debounced resize handler to the window resize event
window.addEventListener("resize", debouncedResize);
```

Explanation:

The resize update runs only after the user stops resizing for 1 second, avoiding unnecessary recalculations!

Debouncing Scroll Events

Debouncing scroll events is a powerful technique to boost performance and enhance user experience by reducing unnecessary function calls during rapid scrolling.



index.js

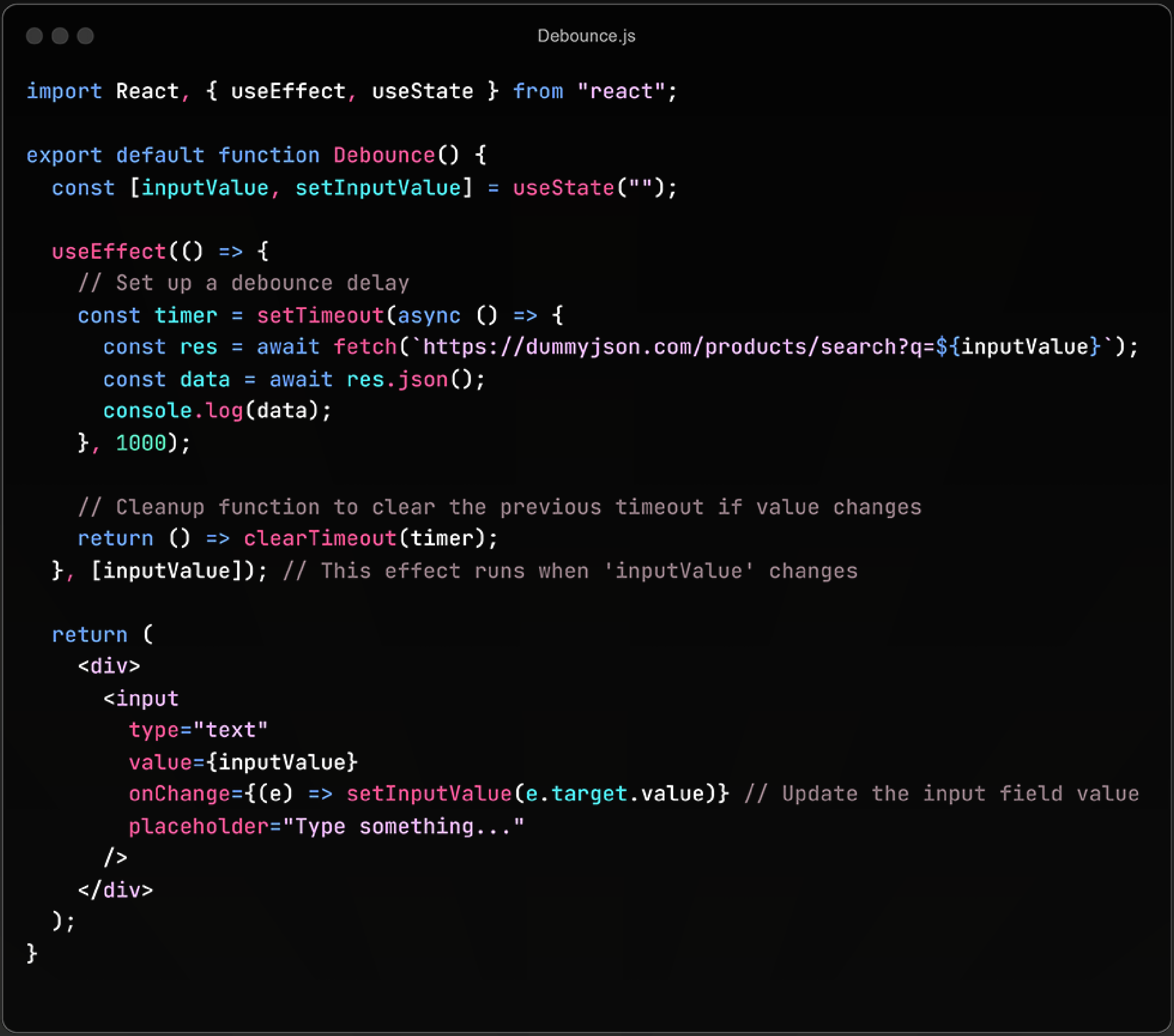
```javascript
const handleScroll = () => {
  console.log("User scrolled!");
  // Add logic to load more content or images
};

function debounce(func, delay) {
  let timeout;
  return function (...args) {
    clearTimeout(timeout);
    timeout = setTimeout(() => func(...args), delay);
  };
}

const debouncedScroll = debounce(handleScroll, 500);

window.addEventListener("scroll", debouncedScroll);
```

Explanation:

The scroll event will trigger the handler 500ms after the user stops scrolling, reducing unnecessary calls while scrolling.

Debouncing with React

Debouncing in React follows the same core principle as vanilla JavaScript, but it's important to handle it correctly within React's component lifecycle.



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

export default function Debounce() {
  const [inputValue, setInputValue] = useState("");

  useEffect(() => {
    // Set up a debounce delay
    const timer = setTimeout(async () => {
      const res = await fetch(`https://dummyjson.com/products/search?q=${inputValue}`);
      const data = await res.json();
      console.log(data);
    }, 1000);

    // Cleanup function to clear the previous timeout if value changes
    return () => clearTimeout(timer);
  }, [inputValue]); // This effect runs when 'inputValue' changes

  return (
    <div>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)} // Update the input field value
        placeholder="Type something..."
      />
    </div>
  );
}
```

Explanation:

State Management: inputValue stores the text input value entered by the user and setInputValue updates the state whenever the user types.

useEffect for Debouncing: Whenever inputValue changes (i.e., when the user types), the effect is triggered.

- A timeout (setTimeout) of 1000ms (1 second) is set up before making an API request.
- If the user keeps typing before 1 second passes, the previous timeout is cleared (clearTimeout), preventing unnecessary API calls.
- If the user stops typing for **at least 1 second**, the API request is triggered.

Input Field: The user types in the input field and onChange updates inputValue, triggering the useEffect.

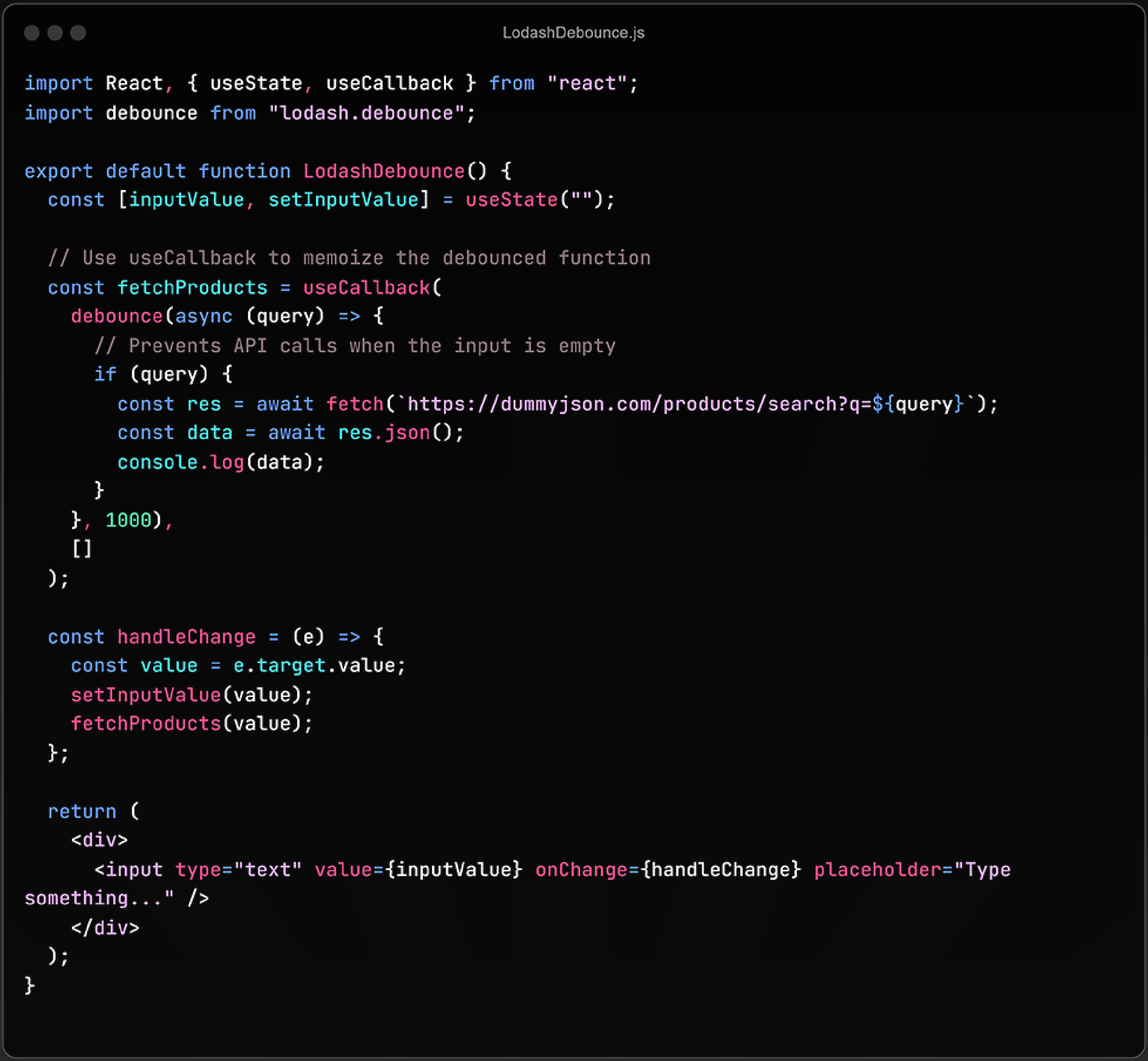
Using a Third-Party Library

You can also use libraries like **lodash** or **use-debounce** for debouncing:

Install Use lodash.debounce

Terminal

```
npm install lodash.debounce
```



LodashDebounce.js

```
import React, { useState, useCallback } from "react";
import debounce from "lodash.debounce";

export default function LodashDebounce() {
  const [inputValue, setInputValue] = useState("");

  // Use useCallback to memoize the debounced function
  const fetchProducts = useCallback(
    debounce(async (query) => {
      // Prevents API calls when the input is empty
      if (query) {
        const res = await fetch(`https://dummyjson.com/products/search?q=${query}`);
        const data = await res.json();
        console.log(data);
      }
    }, 1000),
    []
  );

  const handleChange = (e) => {
    const value = e.target.value;
    setInputValue(value);
    fetchProducts(value);
  };

  return (
    <div>
      <input type="text" value={inputValue} onChange={handleChange} placeholder="Type
something..." />
    </div>
  );
}
```

Explanation:

useCallback to Memoize fetchProducts: Ensures that the debounced function doesn't get recreated on every render.

debounce Inside useCallback: Prevents unnecessary API calls by waiting for 1 second after the user stops typing.

Moves API Call Outside useEffect: No need for useEffect, making the component cleaner.

Prevents API calls when the input is empty: Avoids unnecessary API requests when the input is cleared.