Name of Student: Naveen Gummella			
Roll Number: A-16		**LAB Assignment Number:** 4	
Title of LAB Assignment: Create an application to demonstrate Node.js Functions-timer function.			
DOP: 09/09/2023		**DOS:** 29/09/2023	
CO Mapped: CO1	**PO Mapped:** PO3,PO5,PSO1, PSO2	**Signature:**	**Marks:**

Aim:

Create an application to demonstrate Node.js Functions-timer function (displays every 10 seconds).

Description:

Node.js provides several timer functions that allow you to schedule and execute code at specific times or after specific intervals. These timer functions are part of the Node.js event loop and are crucial for handling asynchronous operations. Here are some theories and concepts related to Node.js timer functions:

Event Loop: Node.js is built around an event-driven, non-blocking I/O model. The event loop is at the core of this model and is responsible for managing timers and executing callback functions when certain events occur.

setTimeout(): The setTimeout() function is used to schedule a callback function to run after a specified delay, measured in milliseconds. It adds the callback to the event loop's queue, and it will be executed once the specified time elapses.

setInterval(): The setInterval() function is used to repeatedly execute a callback function at specified intervals. It works similarly to setTimeout() but continues to run the callback function at the specified interval until explicitly canceled.

Clearing Timers: You can cancel a scheduled timer using clearTimeout() for single-time executions and clearInterval() for repeated executions. Pass the timer ID returned by the corresponding setTimeout() or setInterval() function.

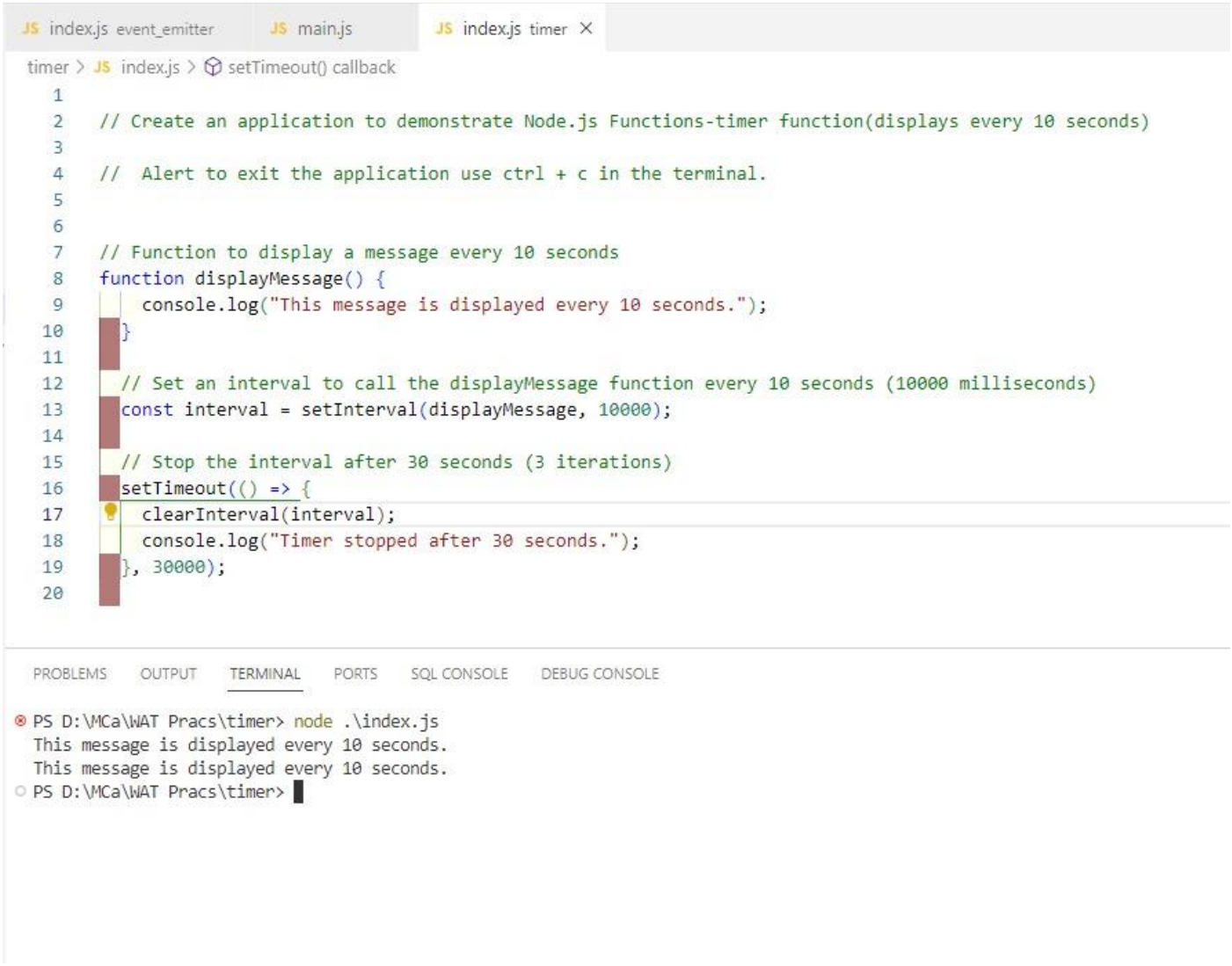
Process.nextTick(): While not a timer function in the traditional sense, process.nextTick() allows you to schedule a callback to be executed on the next iteration of the event loop. It is often used for high-priority asynchronous operations.

Promises and async/await: While timers are useful for scheduling tasks, modern JavaScript development in Node.js often relies on Promises and async/await for handling asynchronous operations more cleanly and predictably.

Timers and Concurrency: When working with timers, it's important to understand how they interact with other asynchronous operations. Timers are executed in a single-threaded manner, and long-running operations can block the event loop, leading to performance issues. Techniques like clustering and worker threads can help mitigate this.

Event Emitters: Node.js timer functions are often used in conjunction with event emitters to trigger custom events when specific conditions are met or when timers expire.

Code & Output:



```
JS index.js event_emitter   JS main.js   JS index.js timer ×
timer > JS index.js > setTimeout() callback
 1
 2  // Create an application to demonstrate Node.js Functions-timer function(displays every 10 seconds)
 3
 4  //  Alert to exit the application use ctrl + c in the terminal.
 5
 6
 7  // Function to display a message every 10 seconds
 8  function displayMessage() {
 9      console.log("This message is displayed every 10 seconds.");
10  }
11
12  // Set an interval to call the displayMessage function every 10 seconds (10000 milliseconds)
13  const interval = setInterval(displayMessage, 10000);
14
15  // Stop the interval after 30 seconds (3 iterations)
16  setTimeout(() => {
17      clearInterval(interval);
18      console.log("Timer stopped after 30 seconds.");
19  }, 30000);
20

PROBLEMS  OUTPUT  TERMINAL  PORTS  SQL CONSOLE  DEBUG CONSOLE

PS D:\MCa\WAT Pracs\timer> node .\index.js
This message is displayed every 10 seconds.
This message is displayed every 10 seconds.
PS D:\MCa\WAT Pracs\timer>
```

Conclusion:

Node.js timer functions are essential for managing asynchronous code execution. Understanding how they fit into the event loop and how to use them effectively is crucial for developing scalable and responsive Node.js applications.