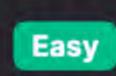
Leetcode Link



Problem Description

The problem requires the creation of an asynchronous function in TypeScript that simulates a sleep or delay for a specified number of milliseconds given by the input parameter 'millis'. The 'millis' is a positive integer representing the time to pause execution of further code. Upon completing the specified pause duration, the function should be able to resume operation although it doesn't need to return any specific value or perform any particular action after the wait is over. The execution should move on, and that is the only requirement.

Intuition

The solution is straightforward due to the nature of JavaScript and TypeScript's handling of asynchronous operations via the Promises API and the built-in setTimeout function.

setTimeout is used to delay the execution of a function by a certain number of milliseconds. It is a non-blocking operation, meaning it allows the JavaScript runtime to continue executing other tasks in the event loop while waiting for the timeout to complete. By wrapping setTimeout in a Promise, we are creating a promise that will resolve after a set amount of time, thus introducing a pause

or sleep effect. This can be done by passing a resolve function r to the promise constructor, and then passing that resolve function to setTimeout. The setTimeout function calls r after the specified millis, signaling to the Promise that it is complete. Since we do not care about the value with which the promise resolves, r can be called with no arguments.

When we call the sleep function with await in an async function or use then() after its invocation, the JavaScript runtime will halt the execution of that function, thus effectively "sleeping", until the set amount of time has passed.

The simplicity of this solution makes it an elegant and widely used approach to introduce a deliberate pause in asynchronous execution without blocking the JavaScript event loop.

The solution takes advantage of the JavaScript event loop and the Promise object to effectively pause execution asynchronously.

resolve function r as an argument.

Solution Approach

There are no complex algorithms, data structures, or patterns needed, as the solution leverages existing JavaScript/TypeScript functionalities to accomplish the 'sleep' behavior. Here's a step-by-step explanation of how the provided solution accomplishes the delay:

An asynchronous function sleep is declared that takes a single parameter, millis, which is a number. This function returns a

- Promise that resolves to void, since the function does not need to return a value once the delay is complete. Inside the function, a new Promise is constructed. The constructor of the Promise takes an executor function, which has the
- Within the executor function, we then call setTimeout. This is a built-in function that schedules another function to be run after a
- given number of milliseconds. In this case, our scheduled function is simply the resolve function r, which will be called after millis milliseconds. By calling the resolve function r after the delay provided by setTimeout, we tell the Promise that it's okay to continue with the
- When the sleep function is used, the caller should either await it (if within an async function) or attach a .then() method to handle the code that should run after the delay. The event loop is not blocked during this delay, allowing other asynchronous

next tasks. Since there's no need for a specific resolve value, r is called without any arguments.

- Thus, once the set time has elapsed, the function within setTimeout (in this case, the resolve function r) is added to the call stack and, if the call stack is clear, is executed, resolving the promise and effectively "waking" the function out of sleep.
- No data structures are used in this solution. The asynchronous nature of the Promise, along with the setTimeout function, provide all the required functionality. This pattern of using setTimeout within a Promise is a common approach to introduce asynchronous

delays in JavaScript and TypeScript codebases. Example Walkthrough

Let's say you want to create a simple application that sends a "hello" message to the console, waits for 2 seconds, and then sends a

"world" message. Here's a step-by-step walkthrough of how you might use the sleep function to achieve this:

operations to continue.

1. Start by declaring an async function named greet where the sleep function will be used. This is your main function.

- 2. Inside the greet function, first output "hello" to the console using console.log("hello"). 3. Directly after printing "hello", call the sleep function with 2000 milliseconds as the argument to create a 2-second delay. This is
- written as await sleep(2000);.
- 4. After the sleep function call, write another console.log() statement to output "world" to the console.
- Here's how the code might look given the provided solution approach:
- return new Promise(r => setTimeout(r, millis));

// Declaration of the async function that uses sleep async function greet()

2 async function sleep(millis: number): Promise<void> {

5. Finally, call the greet function to run the complete sequence.

1 // Declaration of the sleep function

```
console.log("hello"); // Step 2: Output "hello"
     await sleep(2000); // Step 3: Wait for 2 seconds
     console.log("world"); // Step 4: Output "world" after the wait
10
11 }
12
   // Execute the function
14 greet();
When you run this code, you'll see "hello" printed to the console instantly. The program will then pause for 2 seconds, after which
"world" will be printed to the console. This demonstrates the non-blocking behavior of the sleep function, as it allows the "hello"
message to be processed while the event loop handles other tasks during the 2-second wait before "world" is printed out.
```

1 import asyncio # This function creates a coroutine that completes after a specified number of milliseconds. # It uses the asyncio.sleep function internally to delay completion.

await asyncio.sleep(millis / 1000) # asyncio.sleep expects seconds, so convert milliseconds to seconds.

12 # Example usage of the sleep function. It logs the elapsed time (in milliseconds) to the console

* @param millis The number of milliseconds to wait before completing the CompletableFuture.

* @return A CompletableFuture that completes after the specified delay.

// Call the sleep function with a delay of 100 milliseconds.

auto end_time = std::chrono::high_resolution_clock::now();

let elapsedTime: number = Date.now() - startTime;

console.log(elapsedTime); // This will print a value close to 100.

// Log the elapsed time to the console.

// Calculate the elapsed time by subtracting the start time from the current time.

auto elapsed_time = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time).count();

6 # @param millis: The number of milliseconds to wait before completing. # @return: A coroutine that completes after the specified delay. 9 async def sleep(millis: int) -> None:

13 # after the sleep coroutine delay has completed.

11

Python Solution

```
14 #
15 # Usage:
16 # asyncio.run(main())
17
  async def main():
       # Initialize a variable to track the start time.
       start_time = asyncio.get_event_loop().time() * 1000 # start_time is set in milliseconds
20
21
       # Call the sleep coroutine with a delay of 100 milliseconds.
       await sleep(100)
23
24
25
       # Calculate the elapsed time by subtracting the start time from the current time.
       elapsed_time = asyncio.get_event_loop().time() * 1000 - start_time # Convert loop time to milliseconds
26
27
28
       # Log the elapsed time to the console.
29
       print(elapsed_time) # This will print a value close to 100.
30
  # Run the main function if this script is executed
   if __name__ == '__main__':
33
       asyncio.run(main())
34
Java Solution
   import java.util.concurrent.CompletableFuture;
   import java.util.concurrent.TimeUnit;
   // This class contains methods for asynchronous operations.
   public class AsyncHelper {
       /**
        * This method creates a CompletableFuture that completes after a specified number of milliseconds.
        * It uses the scheduled executor internally to delay the completion of the CompletableFuture.
```

public CompletableFuture<Void> sleep(long millis) { 14 return CompletableFuture.runAsync(() -> { 15 try { 16

10

11

12

13

```
TimeUnit.MILLISECONDS.sleep(millis);
17
                } catch (InterruptedException e) {
18
19
                   Thread.currentThread().interrupt();
20
           });
21
22
23
24
   // The main class to run the example usage.
   public class SleepExample {
27
28
       public static void main(String[] args) {
29
           // Create an instance of AsyncHelper to use the sleep method.
30
           AsyncHelper asyncHelper = new AsyncHelper();
31
32
           // Initialize a variable to track the start time.
33
            long startTime = System.currentTimeMillis();
34
35
           // Call the sleep method with a delay of 100 milliseconds.
           asyncHelper.sleep(100).thenRun(() -> {
36
37
               // Calculate the elapsed time by subtracting the start time from the current time.
38
                long elapsedTime = System.currentTimeMillis() - startTime;
39
               // Log the elapsed time to the console.
40
               System.out.println(elapsedTime); // This will print a value close to 100.
           });
42
43
44 }
45
C++ Solution
 1 #include <iostream>
 2 #include <chrono>
   #include <thread>
   // This function pauses execution for a specified number of milliseconds.
 7 // @param millis The number of milliseconds to wait before resuming execution.
   void sleep(int millis) {
       std::this_thread::sleep_for(std::chrono::milliseconds(millis));
10 }
11
  // Example usage of the sleep function. It logs the elapsed time (in milliseconds) to the console
   // after the sleep function delay has completed.
   int main() {
15
       // Initialize a variable to track the start time.
       auto start_time = std::chrono::high_resolution_clock::now();
16
17
```

return 0; 28 29 }

sleep(100);

18

19

20

21

22

23

24

24

25

26

28

27 });

```
25
       // Log the elapsed time to the console.
26
       std::cout << elapsed_time << std::endl; // This will print a value close to 100.
27
30
Typescript Solution
1 // This function creates a Promise that resolves after a specified number of milliseconds.
2 // It uses the setTimeout function internally to delay the resolution of the Promise.
3 //
4 // @param millis The number of milliseconds to wait before resolving the Promise.
5 // @returns A promise that resolves after the specified delay.
6 async function sleep(millis: number): Promise<void> {
       return new Promise(resolve => setTimeout(resolve, millis));
8 }
  // Example usage of the sleep function. It logs the elapsed time (in milliseconds) to the console
11 // after the sleep function delay has completed.
13 // Usage:
14 // let elapsedTime = Date.now();
  // sleep(100).then(() => console.log(Date.now() - elapsedTime)); // Prints approximately 100
16
  // Initialize a variable to track the start time.
   let startTime: number = Date.now();
19
  // Call the sleep function with a delay of 100 milliseconds.
21 sleep(100).then(() => {
       // Calculate the elapsed time by subtracting the start time from the current time.
22
```

Time and Space Complexity

call that enqueues a callback to be executed after a minimum delay. The complexity of the JavaScript code itself is constant,

internally, which use a constant amount of space. No additional space that grows with the input size is used.

regardless of the size of millis. Space Complexity: The space complexity of the sleep function is also 0(1). The function creates a new Promise and a single timer

Time Complexity: The time complexity of the sleep function is 0(1). This is because scheduling a timeout using setTimeout is an API