

Node.js Documentation

Module 1: Intro to Node.js

- **What is Node.js?**

Node.js is a runtime environment that allows you to run JavaScript on the server side. It uses the V8 JavaScript engine, the same engine that powers Google Chrome, to execute code.

- **PHP vs. Node.js**

PHP and Node.js are both popular choices for server-side development but have different strengths and use cases.

PHP

1. **Blocking I/O:** PHP uses a blocking I/O model where each I/O operation (like reading a file, or querying a database) blocks the execution of the script until the operation is completed.
2. **Synchronous:** PHP scripts are executed in a synchronous manner.
3. **Common Use Case:** Traditional web development with tools like WordPress, Laravel, etc.

Node.js

1. **Non-blocking I/O:** Node.js uses a non-blocking I/O model where operations can be executed without waiting for previous ones to complete.

2. **Asynchronous:** Node.js is designed for asynchronous execution.
3. **Common Use Case:** Real-time applications, API development, and applications requiring high concurrency.

- **RAM vs. I/O Latency**

RAM (Random Access Memory)

RAM is a type of computer memory that can be accessed randomly. It is used to store data that is being actively used or processed by the CPU. Accessing data from RAM is much faster compared to accessing data from a hard drive or SSD.

Key Points:

1. **Fast Access:** Data in RAM can be accessed in nanoseconds, making it extremely fast.
2. **Volatile:** RAM is volatile memory, meaning it loses all stored information when the power is turned off.
3. **Temporary Storage:** RAM is used for temporary storage of data that the CPU needs quick access to.
4. **Limited Size:** RAM is typically smaller in size compared to disk storage but is much faster.
5. **Used for Active Data:** RAM stores data that is actively being used or processed by applications.

6. **Expensive:** RAM is more expensive per byte compared to disk storage.
7. **Performance Impact:** More RAM can significantly improve the performance of applications by reducing the need to access slower storage.
8. **Random Access:** RAM allows for data to be read and written in any order, providing flexibility and speed.

I/O Latency (Input/Output Latency)

I/O latency refers to the time it takes to complete an I/O operation, such as reading from or writing to a disk or network. It is typically measured in milliseconds or microseconds.

Key Points:

- **Slower Access:** I/O operations, such as reading from or writing to a disk or network, have higher latency compared to RAM access.
- **Persistent Storage:** Unlike RAM, data stored in I/O devices (like hard drives and SSDs) is persistent and remains even when the power is off.
- **Physical Limitations:** The speed of I/O operations is limited by the physical properties of the storage medium or network.
- **Measured in Milliseconds:** I/O latency is typically measured in milliseconds or microseconds, which is much slower than nanoseconds for RAM.

- **Blocking Operations:** Synchronous I/O operations can block the execution of a program until the operation is completed.
- **Asynchronous Operations:** Asynchronous I/O operations allow other tasks to continue executing while waiting for the I/O operation to complete, improving efficiency.
- **Used for Long-term Storage:** I/O devices are used for long-term storage of data, including files, databases, and backups.
- **Cost-effective:** I/O devices, especially traditional hard drives, are more cost-effective per byte compared to RAM.

- **Blocking vs Non-blocking**

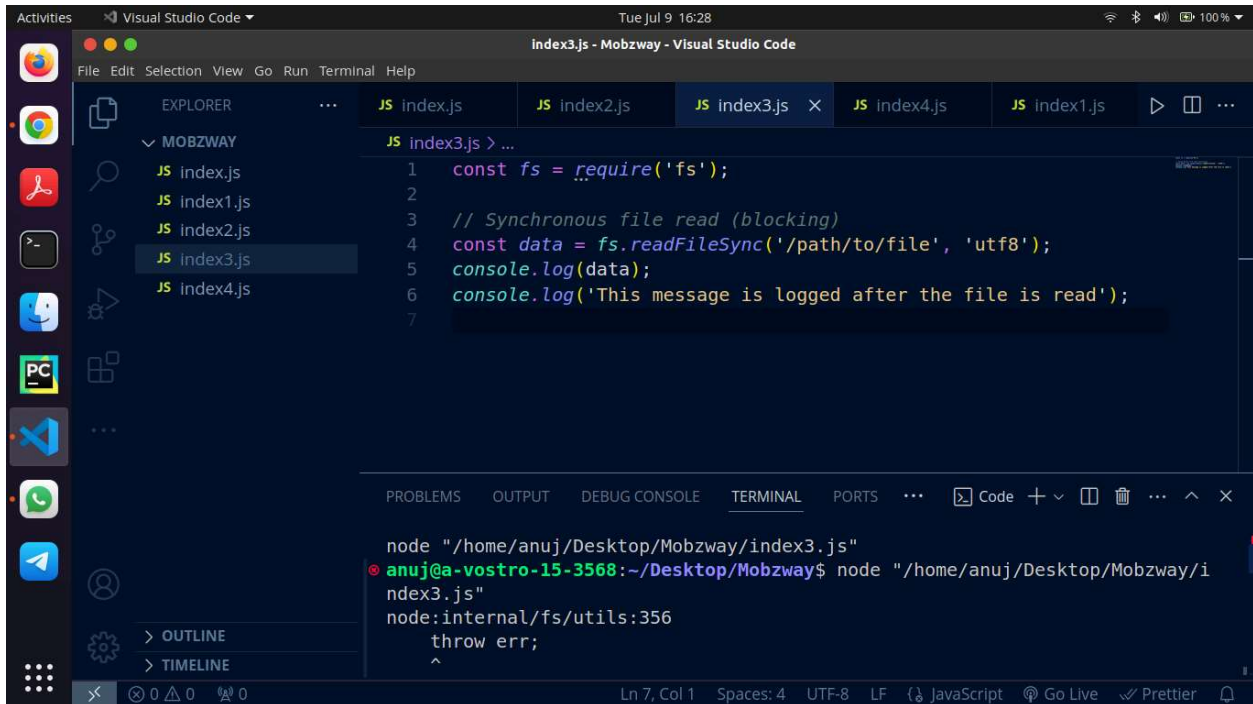
Blocking Operations

Blocking operations cause the execution of subsequent code to wait until the current operation completes. This means that no other operations can be processed until the blocking operation finishes.

Characteristics of Blocking Operations:

1. **Synchronous Execution:** The code is executed line by line, and each operation must complete before the next one starts.
2. **Potential for Delays:** Long-running operations can cause significant delays, making the application unresponsive.
3. **Simple but Inefficient:** Easy to write and understand but can lead to performance bottlenecks.

Example of Blocking Code:



The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists files in a folder named 'MOBZWAY': index.js, index1.js, index2.js, index3.js (selected), and index4.js. The main editor displays the content of index3.js:

```
1  const fs = require('fs');
2
3  // Synchronous file read (blocking)
4  const data = fs.readFileSync('/path/to/file', 'utf8');
5  console.log(data);
6  console.log('This message is logged after the file is read');
7
```

Below the editor is the TERMINAL panel. It shows the command to run the file and the output:

```
node "/home/anuj/Desktop/Mobzway/index3.js"
anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/index3.js"
node:internal/fs/utils:356
  throw err;
  ^
```

In this example:

- `fs.readFileSync` is a blocking operation that reads the file synchronously.
- The execution of `console.log (This message is logged after the file is read')` is blocked until the file read operation completes.

Non-Blocking Operations

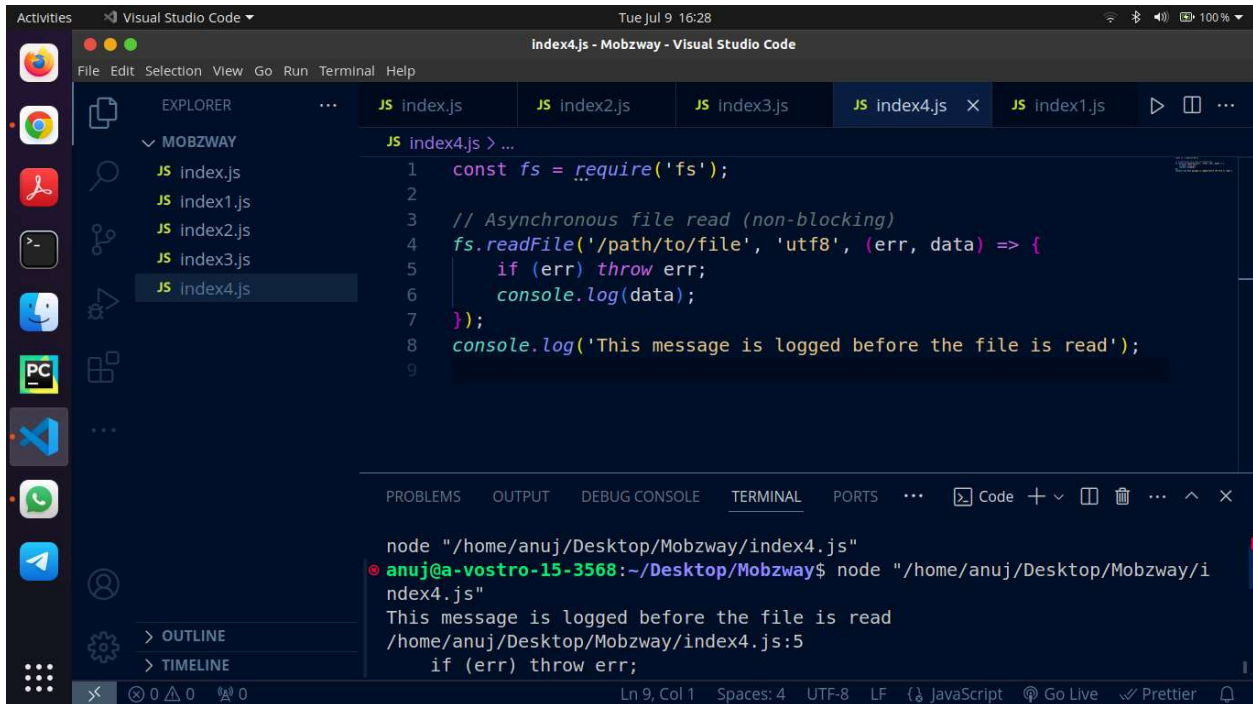
Non-blocking operations allow other tasks to be processed while the current operation is being executed. This approach enables higher concurrency and better performance, especially for I/O-bound tasks.

Characteristics of Non-Blocking Operations:

1. **Asynchronous Execution:** The code continues to execute without waiting for the current operation to complete.

2. **Concurrency:** Multiple operations can be handled concurrently, improving efficiency.
3. **More Complex:** Requires handling callbacks, promises, or async/await, making the code more complex but highly performant.

Example of Non-Blocking Code:



```
JS index4.js > ...
1  const fs = require('fs');
2
3  // Asynchronous file read (non-blocking)
4  fs.readFile('/path/to/file', 'utf8', (err, data) => {
5    if (err) throw err;
6    console.log(data);
7  });
8  console.log('This message is logged before the file is read');
9
```

```
node "/home/anuj/Desktop/Mobzway/index4.js"
anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/index4.js"
This message is logged before the file is read
/home/anuj/Desktop/Mobzway/index4.js:5
    if (err) throw err;
```

In this example:

- `fs.readFile` is a non-blocking operation that reads the file asynchronously.
- `console.log('This message is logged before the file is read')` is executed immediately without waiting for the file read operation to complete.

- **Event-Driven Programming:** Event-driven programming means a callback function will be executed when the corresponding event occurs.

For this, we make an object of the EventEmitter class and through this object, we can access any method in this class.

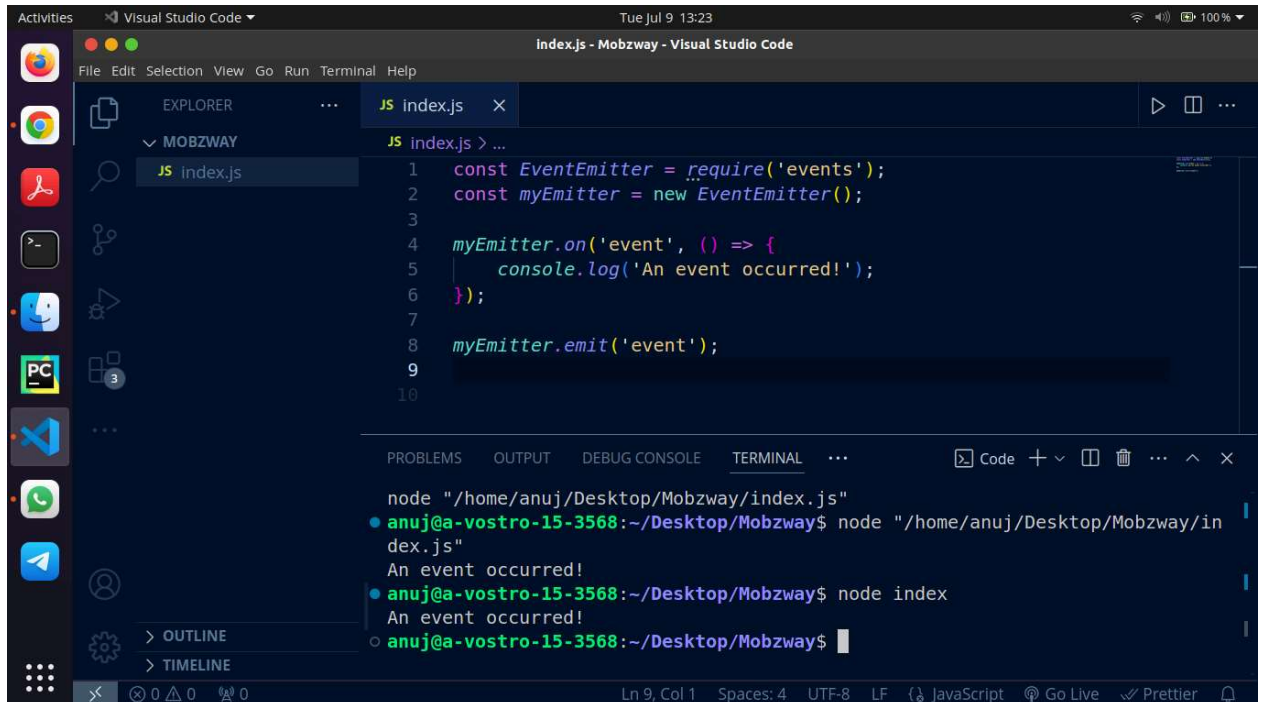
We use .emit method for the occurrence of the event.

.emit method takes the event name as an argument.

.on method has two parameters:

1. Event name
2. Listener: It is a callback function that will be executed when the corresponding event occurs.

Below code is an example of event-driven programming.



The screenshot shows the Visual Studio Code editor with a file named 'index.js' open. The code in the editor is as follows:

```
1 const EventEmitter = require('events');
2 const myEmitter = new EventEmitter();
3
4 myEmitter.on('event', () => {
5   console.log('An event occurred!');
6 });
7
8 myEmitter.emit('event');
```

Below the editor, the terminal window shows the execution of the code:

```
node "/home/anuj/Desktop/Mobzway/index.js"
anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "index.js"
An event occurred!
anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node index
An event occurred!
anuj@a-vostro-15-3568:~/Desktop/Mobzway$
```

In the above code, when an event occurs in line 8, the corresponding callback function, which is in line 4, will be executed.

- **Event Loop**

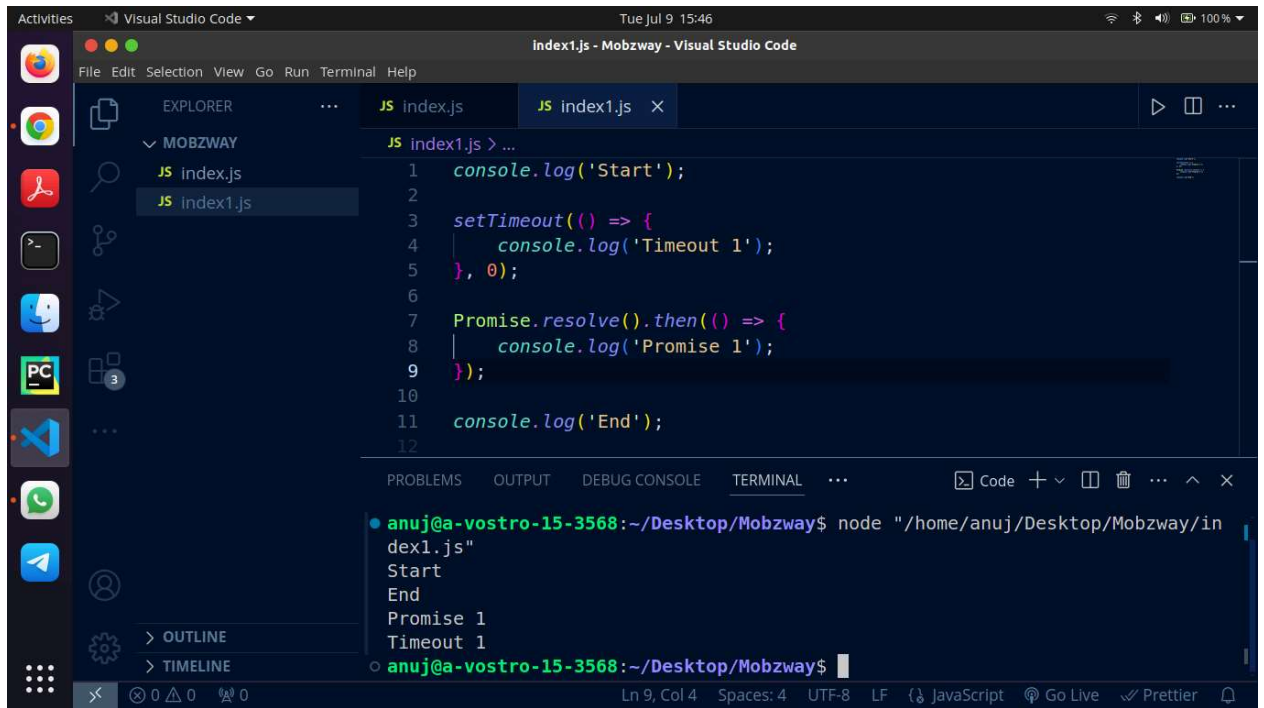
The event loop is a fundamental concept in Node.js that allows it to handle asynchronous operations and non-blocking I/O. It is the mechanism that Node.js

uses to manage the execution of multiple pieces of code without the need for multiple threads.

Event Loop Explained

1. **Single-Threaded:** Node.js operates on a single-threaded event loop, which is responsible for executing JavaScript code, handling events, and performing I/O operations asynchronously.
2. **Non-blocking I/O:** The event loop enables Node.js to perform non-blocking I/O operations despite being single-threaded. This means it can handle other tasks while waiting for I/O operations to complete.
3. **Phases of the Event Loop:**
 - **Timers:** Executes callbacks scheduled by `setTimeout` and `setInterval`.
 - **Pending Callbacks:** Executes I/O callbacks deferred to the next iteration of the loop.
 - **Idle, Prepare:** Internal use only.
 - **Poll:** Retrieves new I/O events; executes I/O callbacks (almost all events except close, timers, and `setImmediate`).
 - **Check:** Executes callbacks scheduled by `setImmediate`.
 - **Close Callbacks:** Executes close event callbacks like `socket.on('close' ...)`.
4. **Microtasks:** Tasks such as `process.nextTick` and resolved Promises have a higher priority than other tasks and are executed immediately after the currently executing script before the event loop continues.

Example Explanation:



The screenshot shows the Visual Studio Code editor with a file named `index1.js` open. The file contains the following JavaScript code:

```
1 console.log('Start');
2
3 setTimeout(() => {
4   console.log('Timeout 1');
5 }, 0);
6
7 Promise.resolve().then(() => {
8   console.log('Promise 1');
9 });
10
11 console.log('End');
```

The terminal at the bottom shows the command `node "/home/anj/Desktop/Mobzway/index1.js"` being executed, with the output:

```
Start
End
Promise 1
Timeout 1
```

Let's break down the execution of this example step by step.

1. **Synchronous Code Execution:** The code starts executing from the top.
 - `console.log('Start');` is executed, and "Start" is printed to the console.
2. **setTimeout:**
 - `setTimeout` is called with a delay of 0 milliseconds. This means the callback function is added to the Timer phase of the event loop but will only be executed after the current stack and microtasks are completed.
3. **Promise:**
 - `Promise.resolve().then(...)` schedules the callback function to be executed as a microtask (part of the microtask queue).

4. Synchronous Code Execution Continues:

- `console.log('End');` is executed, and "End" is printed to the console.

5. Microtasks Execution:

- After the synchronous code is executed, the event loop checks the microtask queue. The callback from `Promise.resolve().then(...)` is executed, printing "Promise 1" to the console.

6. Timers Phase:

- Once all microtasks are completed, the event loop moves to the Timers phase and executes the `setTimeout` callback, printing "Timeout 1" to the console.

Summary of the Event Loop Execution

1. **Initial Execution:** `console.log('Start');` and `console.log('End');` are executed sequentially.
2. **Scheduling Tasks:** `setTimeout` schedules a callback in the Timers phase, and `Promise.resolve().then(...)` schedules a microtask.
3. **Microtasks:** Microtasks from the microtask queue (like resolved Promises) are executed after the current stack is completed but before moving to the next phase of the event loop.
4. **Timers Phase:** Once all synchronous code and microtasks are executed, the event loop proceeds to execute the `setTimeout` callback in the Timers phase.

- **Blocking the Event Loop**

Blocking the event loop in Node.js is a critical concept to understand because it directly affects the performance and responsiveness of your application. Node.js is designed to handle many operations concurrently, but blocking the event loop can prevent it from processing other tasks, leading to delays and decreased performance.

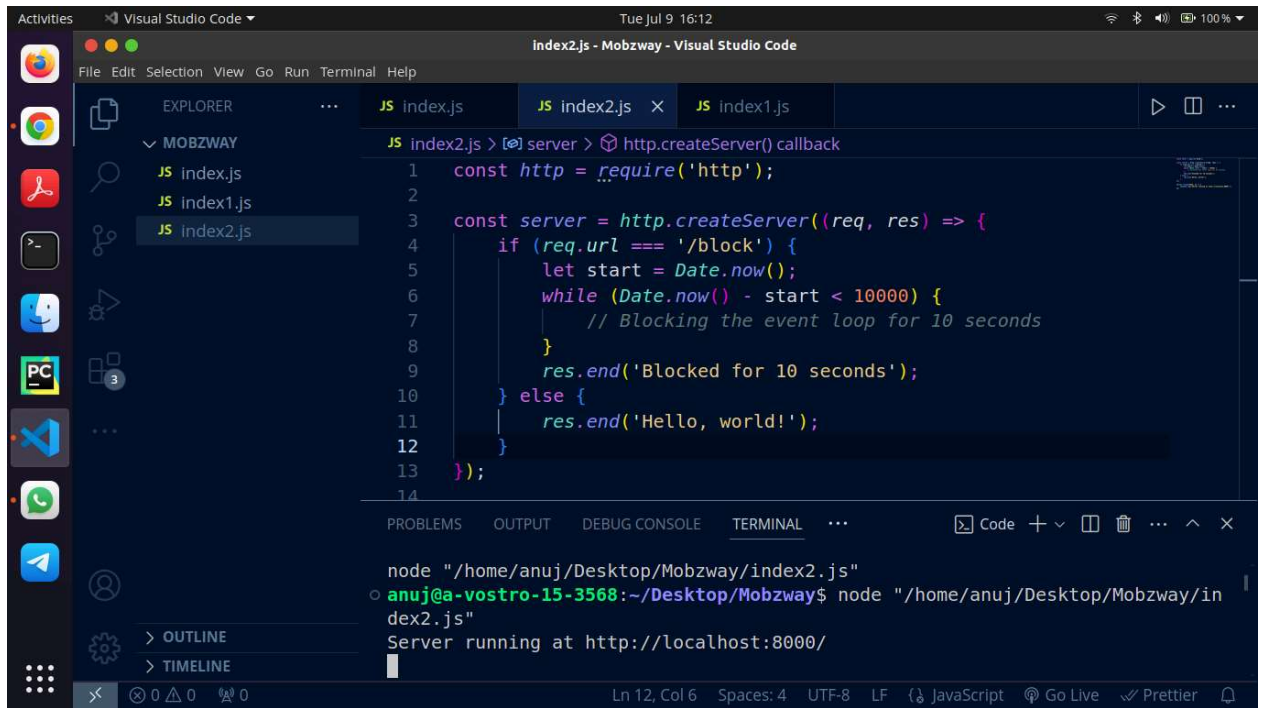
What is Blocking the Event Loop?

Blocking the event loop occurs when a task takes a long time to complete and prevents other tasks from executing. Since Node.js uses a single-threaded event loop to handle all operations, any function that takes a long time to execute will block the event loop and delay the execution of other pending tasks.

How Does Blocking Happen?

1. **Synchronous Code:** Any synchronous code that takes a long time to execute will block the event loop. Examples include complex calculations, large loops, or heavy file operations.
2. **I/O Operations:** While Node.js is designed to handle I/O operations asynchronously, if you use synchronous I/O methods, they will block the event loop.
3. **Long-Running Operations:** Functions that perform long-running operations without yielding control back to the event loop will cause blocking.

Example of Blocking Code



```
index2.js - Mobzway - Visual Studio Code
File Edit Selection View Go Run Terminal Help

EXPLORER
MOBZWAY
  JS index.js
  JS index1.js
  JS index2.js

JS index2.js > [?] server > http.createServer() callback
1  const http = require('http');
2
3  const server = http.createServer((req, res) => {
4    if (req.url === '/block') {
5      let start = Date.now();
6      while (Date.now() - start < 10000) {
7        // Blocking the event loop for 10 seconds
8      }
9      res.end('Blocked for 10 seconds');
10   } else {
11     res.end('Hello, world!');
12   }
13 });
14

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
node "/home/anuj/Desktop/Mobzway/index2.js"
anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/index2.js"
Server running at http://localhost:8000/

Ln 12, Col 6 Spaces: 4 UTF-8 LF JavaScript Go Live Prettier
```

Breakdown of the Example

1. **Create HTTP Server:** An HTTP server is created.
2. **Blocking Route:** If the request URL is `/block`, the server will enter a while loop that runs for 10 seconds.
3. **Non-blocking Route:** For any other request URL, the server responds with "Hello, world!".

When a request is made to `/block`, the server will not be able to handle any other requests until the 10-second loop completes. This blocks the event loop and makes the server unresponsive during that time.

How to Avoid Blocking the Event Loop

1. **Use Asynchronous Operations:** Always prefer asynchronous methods over synchronous ones. For example, use `fs.readFile` instead of `fs.readFileSync`.
2. **Break Long-Running Tasks:** Split long-running tasks into smaller chunks and use `setImmediate` or `process.nextTick` to yield control back to the event loop.
3. **Offload Intensive Tasks:** Offload CPU-intensive tasks to worker threads or child processes.

