Episode-07 | sync, async, setTimeoutZero - code

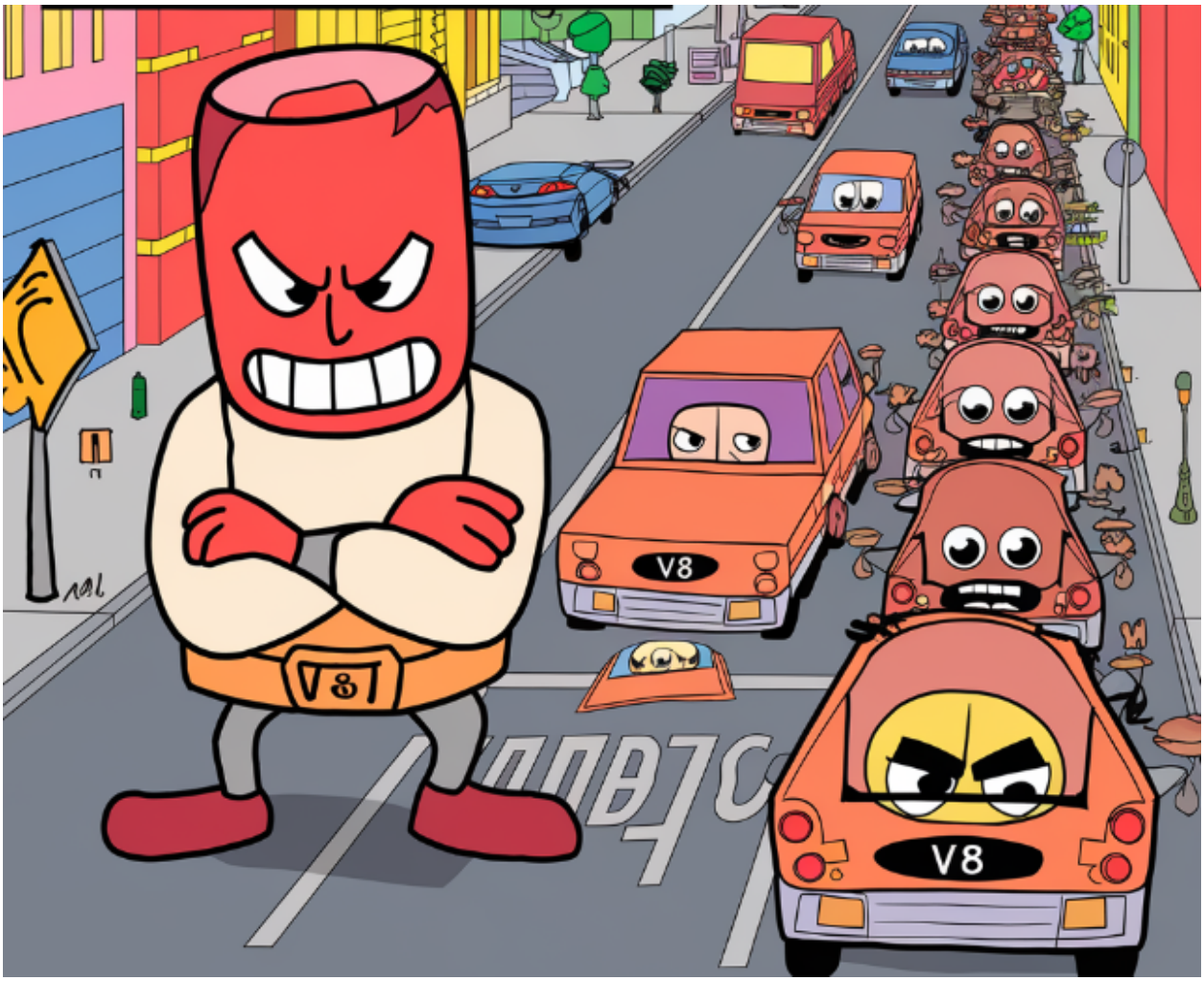
Let me explain something further about the code we discussed in our last episode.

JS xyz.js > multiplyFn

```
1   console.log("Hello world");
2
3   var a = 107;
4   var b = 20;
5
6   //synchronous
7   fs.readFileSync("./file.txt", "utf8"); //10ms
8   console.log("this will execute only after file read");
9
10  Https.get("https://dummyjson.com/products/1", (res) => {
11    console.log("fetched data success");
12  });
13
14  setTimeout(() => {
15    console.log("setTimeout called after 5 seconds");
16  }, 5000);
17
18  //Async function
19  fs.readFile("./file.txt", "utf8", (err, data) => {
20    console.log("File Data : ", data);
21  });
22
23  function multiplyFn(x, y) {
24    const result = a * b;
25    return result;
26  }
27
```

```
fs.readFileSync("./file.txt", "utf8");
```

This function reads the contents of a file synchronously, meaning it will actually block the main thread while it's running.



What does this mean?

- The **V8 engine** is responsible for executing JavaScript code, but it cannot offload this task to **Libuv** (which handles asynchronous operations like I/O tasks).
- Think of it like an ice cream shop where the owner insists on serving each customer one at a time, without moving on to the next until the current customer has been fully served. This is what happens when you use synchronous methods like `fs.readFileSync()`—the main thread is blocked until the file is completely read

Why is this important?

- As a developer, it's important to understand that while Node.js and the V8 engine give you the capability to block the main thread using synchronous methods, this is generally not recommended.
- **Synchronous methods** like `fs.readFileSync()` are still available in Node.js, but using them can cause performance issues because the code execution will be halted at that point (e.g., line 7) until the file reading operation is complete.

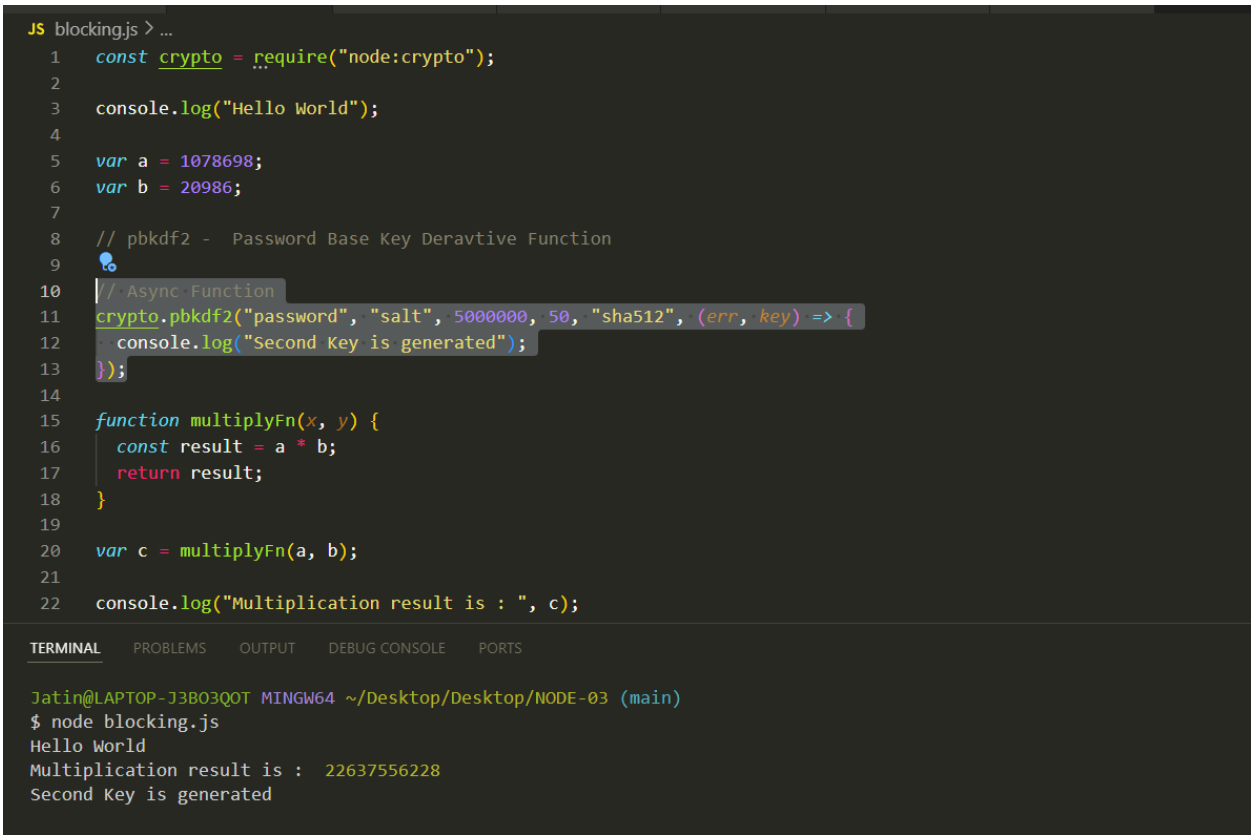
Best Practice

- **Avoid using synchronous methods** in production code whenever possible, especially in performance-critical applications, because they can slow down your application by blocking the event loop.
- Instead, use asynchronous methods like `fs.readFile()` that allow other operations to continue while the file is being read, keeping the application responsive.

Let me show you another example of blocking code in Node.js.

Introducing the `crypto` Module

Node.js has a core library known as `crypto`, which is used for cryptographic operations like generating secure keys, hashing passwords, and more.



What is `crypto`?

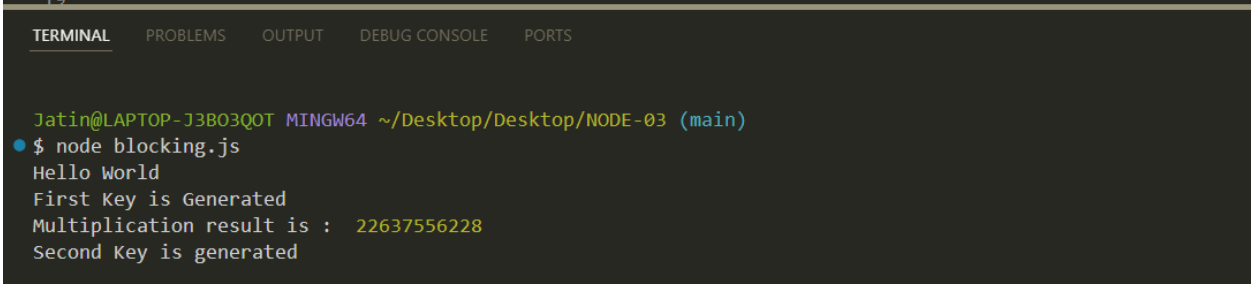
- The `crypto` module is one of the core modules provided by Node.js, similar to other core modules like `https`, `fs` (file system), and `zlib` (used for compressing files).
- These core modules are built into Node.js, so when you write `require('crypto')`, you're importing a module that is already present in Node.js.
- You can also import it using `require('node:crypto')` to explicitly indicate that it's a core module, but this is optional.

Example of Blocking Code with `crypto`

One of the functions provided by the `crypto` module is `pbkdf2Sync`, which stands for **Password-Based Key Derivation**

Function 2. This function is used to generate a cryptographic key from a password and salt, and it operates synchronously.

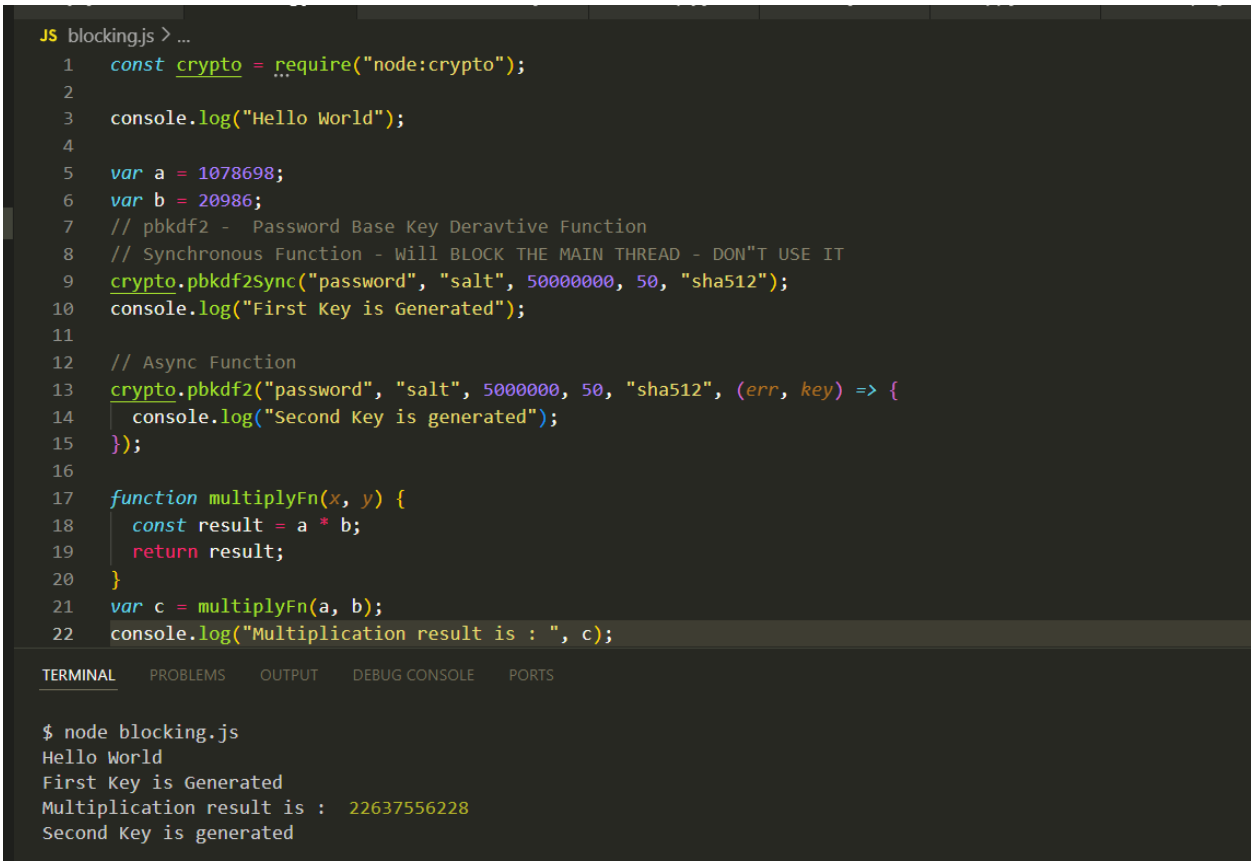
```
// Synchronous Function - Will BLOCK THE MAIN THREAD - DON"T USE IT

crypto.pbkdf2Sync("password", "salt", 50000000, 50, "sha512");
console.log("First Key is Generated");

// Async Function
crypto.pbkdf2("password", "salt", 5000000, 50, "sha512", (err, key) => {
  console.log("Second Key is generated");
});
```



The first key is generated first because it's synchronous, while the second key is generated afterward because it's asynchronous

Here's what this function does:

- **Password and Salt**: You provide a password and a salt value, which are combined to create a cryptographic key.
- **Iterations**: You specify the number of iterations (e.g., 50,000) to increase the complexity of the key, making it harder to crack.
- **Digest Algorithm**: You choose a digest algorithm, like `sha512`, which determines how the key is hashed.
- **Key Length**: You define the length of the key (e.g., 50 bytes).
- **Callback**: In the asynchronous version (`pbkdf2`), a callback is provided to handle the result once the key is generated.



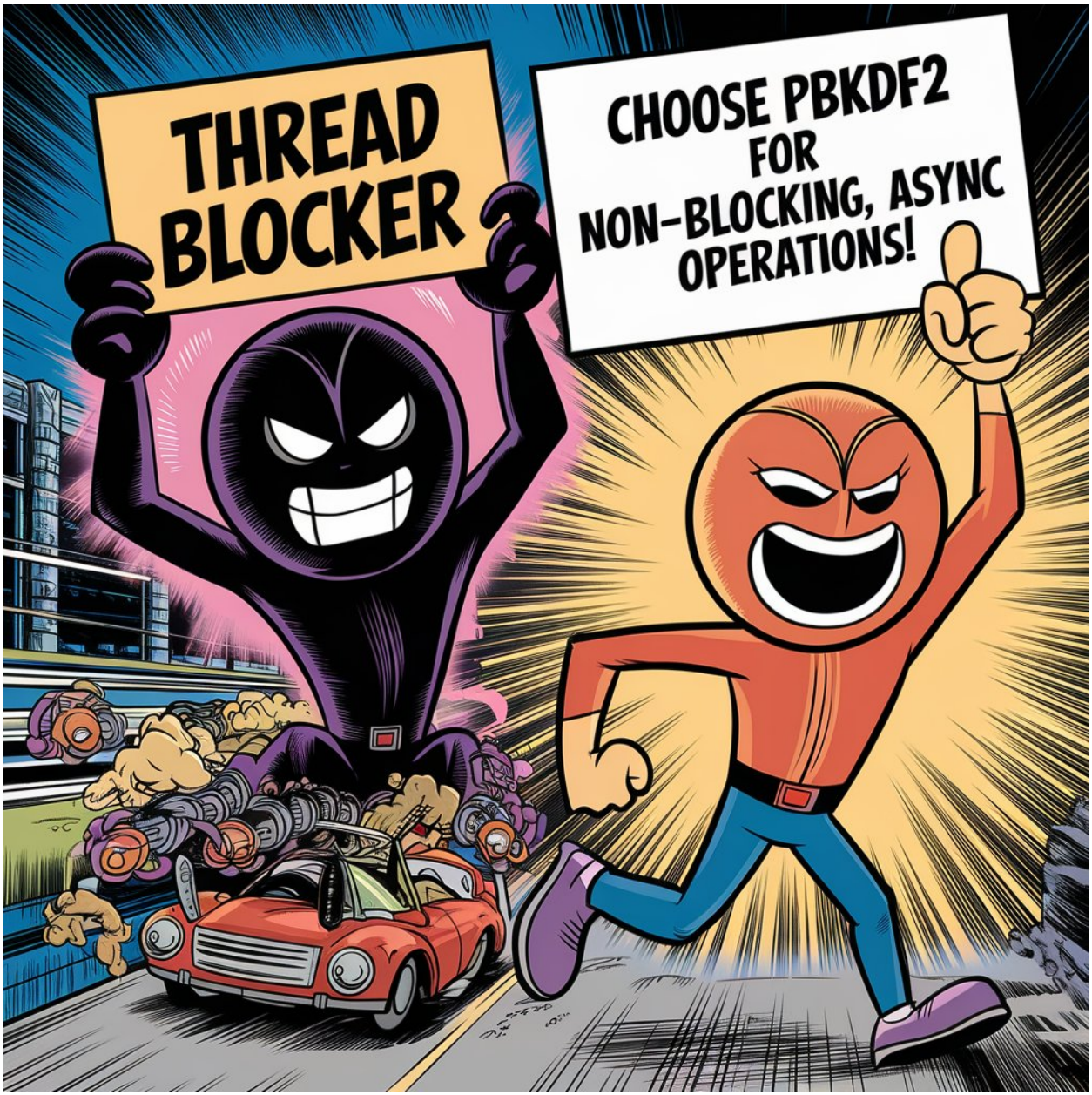
Important Note:

- When you see `Sync` at the end of a function name (like `pbkdf2Sync`), it means that the function is **synchronous** and will block the main thread while it's running. This is something you should be cautious about, especially in performance-sensitive applications.

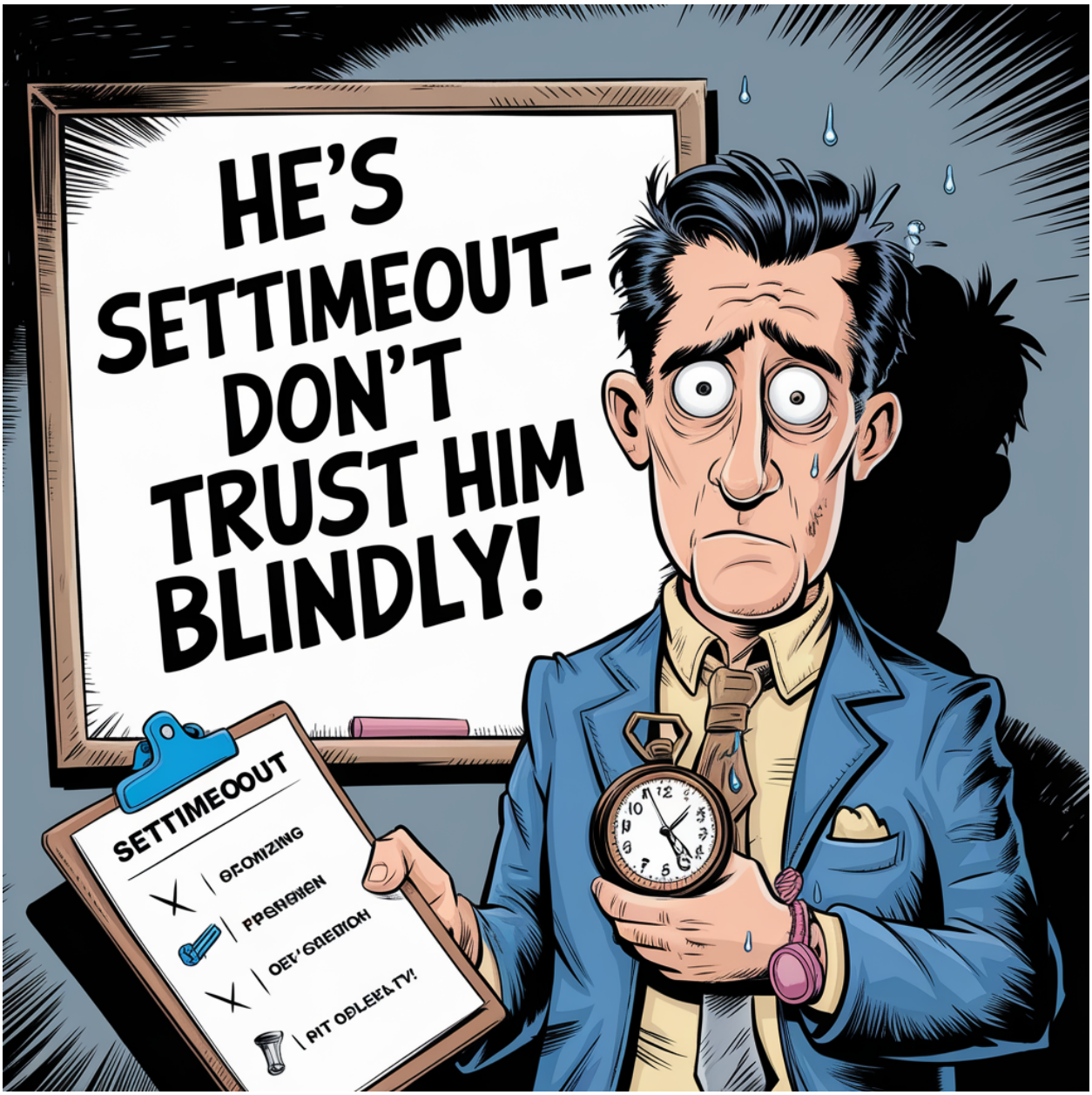
Why Does This Matter?

- **Blocking Behavior**: The synchronous version of `pbkdf2` (`pbkdf2Sync`) will block the event loop, preventing any other code from executing until the key generation is complete. This can cause your application to become unresponsive if used inappropriately.

- **Asynchronous Alternative**: Node.js also provides an asynchronous version (`pbkdf2` without the `Sync` suffix), which offloads the operation to **Libuv**. This allows the event loop to continue processing other tasks while the key is being generated.



New Interesting Concept: `setTimeout(0)`

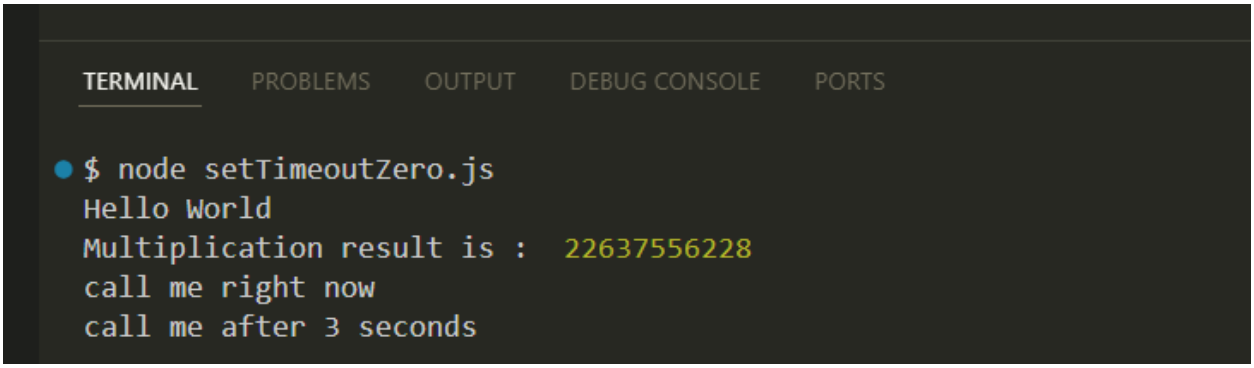


Interview Tip: This can be a tricky output question!

Q: What will be the output of the following code?

```js
console.log("Hello World");

var a = 1078698;
var b = 20986;


setTimeout(() => {
  console.log("call me right now ");
}, 0); // Trust issues with setTimeout

setTimeout(() => {
  console.log("call me after 3 seconds");
}, 3000);

function multiplyFn(x, y) {
  const result = a * b;
  return result;
}

var c = multiplyFn(a, b);

console.log("Multiplication result is : ", c);

```

Answer:



Why is `setTimeout(0)` Executed After the Multiplication Result?

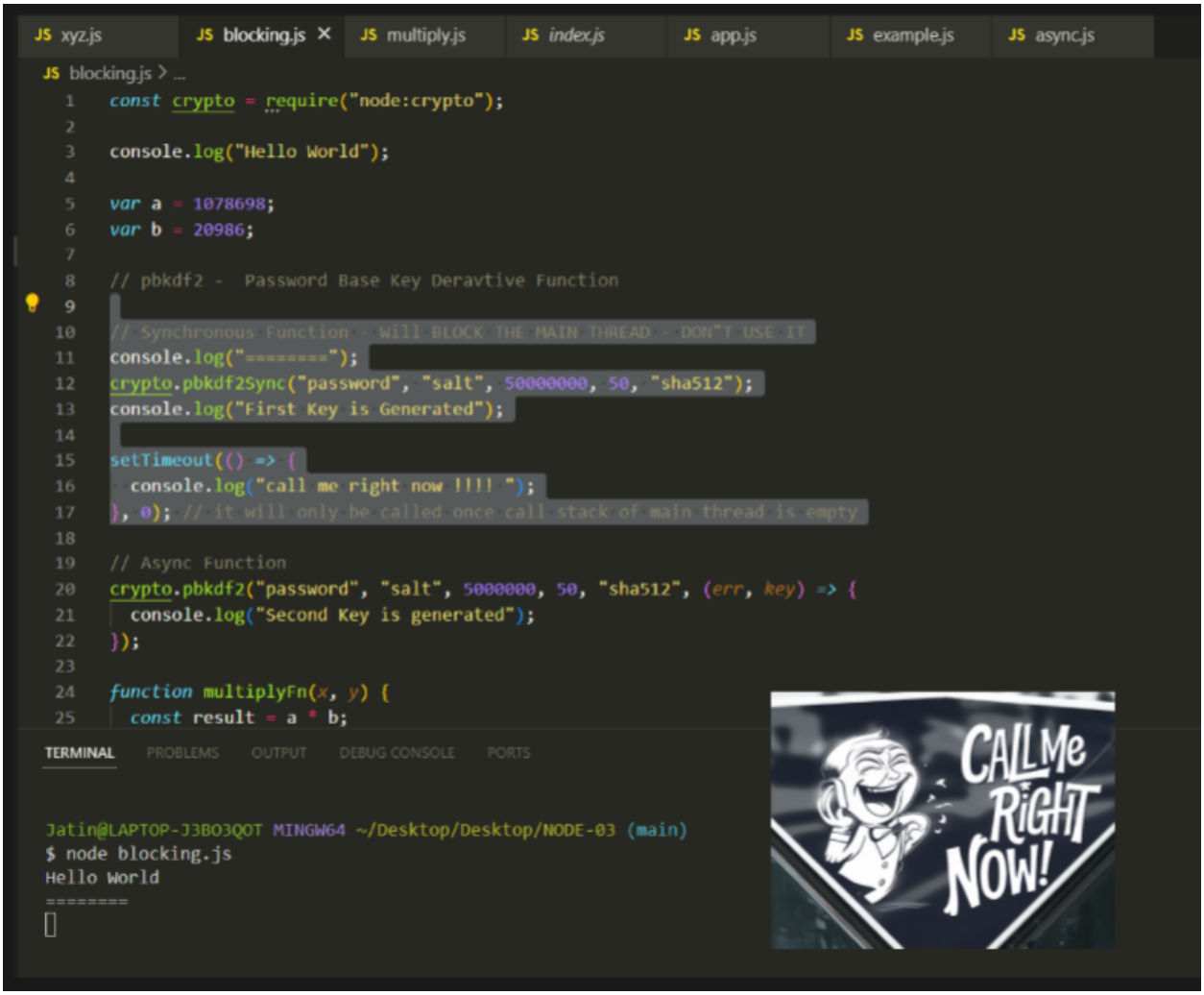
You might wonder why the `setTimeout(0)` callback is executed after other code, like the multiplication result, even though we set the delay to 0 milliseconds.

The Reason:

- **Asynchronous Operation**: `setTimeout` is an asynchronous function, meaning it doesn't block the execution of the code. When you call `setTimeout`, the

callback function is passed to **Libuv** (Node.js's underlying library), which manages asynchronous operations.

- **Event Loop and Call Stack**: The callback function from `setTimeout(0)` is added to the event queue. However, it won't be executed until the current **call stack** is empty. This means that even if you specify a 0-millisecond delay, the callback will only execute after the global execution context is done.
- **Trust Issues with `setTimeout(0)`**: When you ask the code to run after 0 milliseconds, it doesn't necessarily run immediately after that time. It runs **only when the call stack is empty**. This introduces some "terms and conditions," meaning that the actual execution timing is dependent on the state of the call stack.
-




YOU ACTUALLY LISTENED TO THE END?
WELL THANK YOU FOR YOUR ATTENTION!
makeameme.org