

IP PRACTICAL NO. 11

Date of Performance: 25 – 10 – 2022

Software Requirement: Visual Studio Code, NotePad, NotePad++, NodeJS

Aim: To implement Callback, Event loop, Event emitter, Buffers & Stream in NodeJS

Objectives: The aim of this experiment is that the students will be able

- Implement Callback, Event Loop and Event Emitter
- Understand how buffers and streams work

Outcomes: After study of this experiment, the students will be able

- Implement Callback, Event Loop and Event Emitter using NodeJS • Create buffers and streams

Prerequisite: Basic knowledge of HTML, CSS, JavaScript & React required.

Theory:

Callback

Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All the APIs of Node are written in such a way that they support callbacks.

For example, a function to read a file may start reading file and return the control to the execution environment immediately so that the next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as a parameter. So, there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process a high number of requests without waiting for any function to return results.

Event Loop

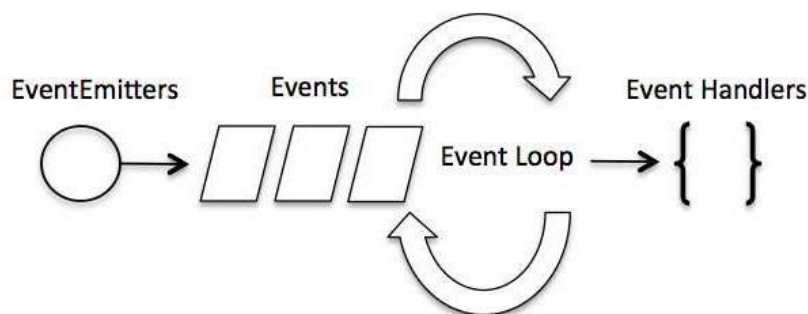
The event loop allows Node.js to perform non-blocking I/O operations despite the fact that JavaScript is single-threaded. It is done by assigning operations to the operating system whenever and wherever possible. When one of these operations is completed, the kernel tells Node.js and the respective callback assigned to that operation is added to the event queue which will eventually be executed.

Features of Event Loop:

- Event loop is an endless loop, which waits for tasks, executes them and then sleeps until it receives more tasks.
- The event loop executes tasks from the event queue only when the call stack is empty i.e., there is no ongoing task.
- The event loop allows us to use callbacks and promises.
- The event loop executes the tasks starting from the oldest first.

Event Emitter

Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast. As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur. In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.



Although events look quite similar to callbacks, the difference lies in the fact that callback functions are called when an asynchronous function returns its result, whereas event handling works on the observer pattern.

Buffers

Pure JavaScript is Unicode friendly, but it is not so for binary data. While dealing with TCP streams or the file system, it's necessary to handle octet streams. Node provides Buffer class which provides instances to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.

Buffer class is a global class that can be accessed in an application without importing the buffer module.

Streams

Streams are objects that let you read data from a source or write data to a destination in continuous fashion. In Node.js, there are four types of streams –

- Readable – Stream which is used for read operation.
- Writable – Stream which is used for write operation.
- Duplex – Stream which can be used for both read and write operation.
- Transform – A type of duplex stream where the output is computed based on input.

Each type of Stream is an EventEmitter instance and throws several events at different instance of times. For example, some of the commonly used events are –

- data – This event is fired when there is data is available to read.
- end – This event is fired when there is no more data to read.
- error – This event is fired when there is any error receiving or writing data.
- finish – This event is fired when all the data has been flushed to underlying system.

Piping the Streams: Piping is a mechanism where we provide the output of one stream as the input to another stream.

Chaining the Streams: Chaining is a mechanism to connect the output of one stream to another stream and create a chain of multiple stream operations.

Problem Statement:

Implement Callback, Event loop, Event emitter, Buffers & Stream in NodeJS

Source Code:

blocking.js:

```
var fs = require("fs");
var data = fs.readFileSync('input');
console.log(data.toString());
console.log("PRAC 12- Q1- BLOCKING CODE EXECUTED SUCCESSFULLY!!!!");
```

non-blocking.js:

```
var fs = require("fs");
fs.readFile('input', function (err, data) {
  if (err) return console.error(err);
  console.log(data.toString());
});
console.log("Q1 OF PRAC12- NON BLOCKING EXECUTED SUCCESSFULLY!!");
```

Output:**Blocking**

```
PS C:\Users\husna\OneDrive\Desktop\node> node q1.js
This is Husna Shaikh, Roll no 612050. I am currently studying in the third year of I.T.
PRAC 12- Q1- BLOCKING CODE EXECUTED SUCCESSFULLY!!!
```

Non-blocking

```
PS C:\Users\husna\OneDrive\Desktop\node> node q2.js
Q1 OF PRAC12- NON BLOCKING EXECUTED SUCCESSFULLY!!
This is Husna Shaikh, Roll no 612050. I am currently studying in the third year of I.T.
```

Source Code: Event Loop

```
var events = require('events');
// Create an EventEmitter object
var EventEmitter = new events.EventEmitter();
// Create an event handler as follows
var connectHandler = function connected() {
  console.log('Husna, your connection was successfully established.');
```

```
// Fire the data_received event
  EventEmitter.emit('data_received');
}
// Bind the connection event with the handler
EventEmitter.on('connection', connectHandler);
// Bind the data_received event with the anonymous function
EventEmitter.on('data_received', function() {
  console.log('Husna, your data was received after successful establishment of
the connection.');
```

```
});
// Fire the connection event
EventEmitter.emit('connection');
console.log("Your connection was fired/emitted.");
```

Output:

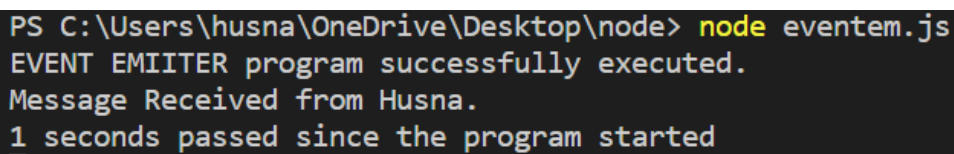
```
PS C:\Users\husna\OneDrive\Desktop\node> node eventloop.js
Husna, your connection was successfully established.
Husna, your data was received after successful establishment of the connection.
Your connection was fired/emitted.
```

Source Code: Event Emitter

```
// const { EventEmitter } = require('events');
//creating event emitters
const timerEventEmitter = new EventEmitter();
timerEventEmitter.emit("update");
let currentTime = 0;
```

```
// This will trigger the update event each passing second
setInterval(() => {
  currentTime++;
  timerEventEmitter.emit('update', currentTime);
}, 1000);
//subscribing to once event listener to the event emitter
timerEventEmitter.once('update', (time) => {
  console.log('Message Received from Husna. ');
  console.log(`${time} seconds passed since the program started`);
});
console.log('EVENT EMIITER program successfully executed. ')
```

Output:

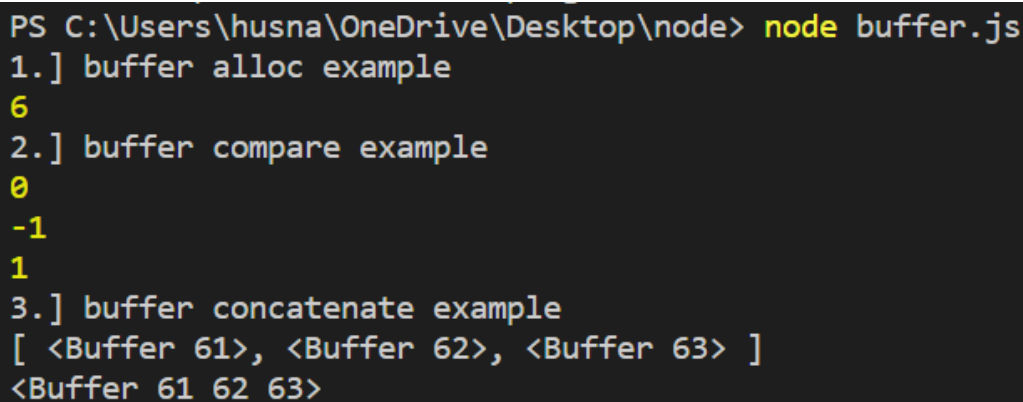


```
PS C:\Users\husna\OneDrive\Desktop\node> node eventem.js
EVENT EMIITER program successfully executed.
Message Received from Husna.
1 seconds passed since the program started
```

Source Code: Buffers

```
//1.] buffer alloc
console.log('1.] buffer alloc example');
var buf = Buffer.alloc(6);
//check the length of buffer created
var buffLen = Buffer.byteLength(buf);
//print buffer length
console.log(buffLen);
console.log('2.] buffer compare example');
//2.] buffer compare
var buf1 = Buffer.from('html');
var buf2 = Buffer.from('html');
var a = Buffer.compare(buf1, buf2);
//This will return 0
console.log(a);
var buf1 = Buffer.from('h');
var buf2 = Buffer.from('t');
var a = Buffer.compare(buf1, buf2);
//This will return -1
console.log(a);
var buf1 = Buffer.from('t');
var buf2 = Buffer.from('h');
var a = Buffer.compare(buf1, buf2);
//This will return 1
console.log(a);
//3.] buffer concatenate
console.log('3.] buffer concatenate example');
```

```
var buffer1 = Buffer.from('a');
var buffer2 = Buffer.from('b');
var buffer3 = Buffer.from('c');
var arr = [buffer1, buffer2, buffer3];
console.log(arr);
//concatenate buffer with Buffer.concat method
var buf = Buffer.concat(arr);
console.log(buf);
```

Output:

```
PS C:\Users\husna\OneDrive\Desktop\node> node buffer.js
1.] buffer alloc example
6
2.] buffer compare example
0
-1
1
3.] buffer concatenate example
[ <Buffer 61>, <Buffer 62>, <Buffer 63> ]
<Buffer 61 62 63>
```

Source Code: Streams

```
var fs = require("fs");
var data = '';
// Create a readable stream
//Reading from a Stream
var readerStream =
fs.createReadStream('input');
// Set the encoding to be utf8.
readerStream.setEncoding('UTF8');
// Handle stream events --> data, end, and error
readerStream.on('data', function(chunk) {
data += chunk;
});
readerStream.on('end',function() {
console.log(data);
});
readerStream.on('error', function(err) {
console.log(err.stack);
});
console.log("Implementation of STREAMS concept for practical 12 done
successfully!");
```

Output:

```
PS C:\Users\husna\OneDrive\Desktop\node> node stream.js
Implementation of STREAMS concept for practical 12 done successfully!
This is Husna Shaikh, Roll no 612050. I am currently studying in the third year of I.T.
```

Conclusion:

From this experiment we understand that React is a very powerful JavaScript Library and is used at a large scale. We have learned how to implement Reactstrap features along with HTML, CSS & JavaScript to create a form and validate its different fields such as name, number, email, etc. We also learned how to create responsive UI using Material-UI.