Web Technologies – Module 02:

What is 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.

Blocking Code Example

Create a text file named **input.txt** with the following content −

```
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

Create a js file named **main.js** with the following code −

```
var fs = require("fs");
var data = fs.readFileSync('input.txt');

console.log(data.toString());
console.log("Program Ended");
```

Now run the main.js to see the result −

```
$ node main.js
```

Verify the Output.

```
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
Program Ended
```

Non-Blocking Code Example

Create a text file named input.txt with the following content.

```
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

Update main.js to have the following code −

```
var fs = require("fs");

fs.readFile('input.txt', function (err, data) {
   if (err) return console.error(err);
   console.log(data.toString());
});

console.log("Program Ended");
```

Now run the main.js to see the result −

```
$ node main.js
```

Verify the Output.

```
Program Ended
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

These two examples explain the concept of blocking and non-blocking calls.

- The first example shows that the program blocks until it reads the file and then only it proceeds to end the program.
- The second example shows that the program does not wait for file reading and proceeds to print "Program Ended" and at the same time, the program without blocking continues reading the file.

Thus, a blocking program executes very much in sequence. From the programming point of view, it is easier to implement the logic but non-blocking programs do not execute in sequence. In case a program needs to use any data to be processed, it should be kept within the same block to make it sequential execution.

Node.js EventEmitter

Node.js allows us to create and handle custom events easily by using events module. Event module includes EventEmitter class which can be used to raise and handle custom events.

The following example demonstrates EventEmitter class for raising and handling a custom event.

Example: Raise and Handle Node.js events

Copy

```
// get the reference of EventEmitter class of events module
var events = require('events');

//create an object of EventEmitter class by using above reference
var em = new events.EventEmitter();

//Subscribe for FirstEvent
em.on('FirstEvent', function (data) {
    console.log('First subscriber: ' + data);
});

// Raising FirstEvent
em.emit('FirstEvent', 'This is my first Node.js event emitter example.');
```

In the above example, we first import the 'events' module and then create an object of EventEmitter class. We then specify event handler function using on() function. The on() method requires name of the event to handle and callback function which is called when an event is raised.

The emit() function raises the specified event. First parameter is name of the event as a string and then arguments. An event can be emitted with zero or more arguments. You can specify any name for a custom event in the emit() function.

You can also use addListener() methods to subscribe for an event as shown below.

Example: EventEmitter

```
var emitter = require('events').EventEmitter;
```

```
var em = new emitter();
//Subscribe FirstEvent
em.addListener('FirstEvent', function (data) {
    console.log('First subscriber: ' + data);
});

//Subscribe SecondEvent
em.on('SecondEvent', function (data) {
    console.log('First subscriber: ' + data);
});

// Raising FirstEvent
em.emit('FirstEvent', 'This is my first Node.js event emitter example.');

// Raising SecondEvent
em.emit('SecondEvent', 'This is my second Node.js event emitter example.');
```

EventEmitter Methods	Description
emitter.addListener(event, listener)	Adds a listener to the end of the listeners array for the specified event. No checks are made to see if the listener has already been added.
emitter.on(event, listener)	Adds a listener to the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. It can also be called as an alias of emitter.addListener()
emitter.once(event, listener)	Adds a one time listener for the event. This listener is invoked only the next time the event is fired, after which it is removed.
emitter.removeListener(event, listener)	Removes a listener from the listener array for the specified event. Caution: changes array indices in the listener array behind the listener.
emitter.removeAllListeners([event])	Removes all listeners, or those of the specified event.
emitter.setMaxListeners(n)	By default EventEmitters will print a warning if more than 10 listeners are added for a particular event.
emitter.getMaxListeners()	Returns the current maximum listener value for the emitter which is either set by emitter.setMaxListeners(n) or defaults to EventEmitter.defaultMaxListeners.
emitter.listeners(event)	Returns a copy of the array of listeners for the specified event.
emitter.emit(event[, arg1][, arg2][, ...])	Raise the specified events with the supplied arguments.
emitter.listenerCount(type)	Returns the number of listeners listening to the type of event.

Common Patterns for EventEmitters

There are two common patterns that can be used to raise and bind an event using EventEmitter class in Node.js.

1. Return EventEmitter from a function
2. Extend the EventEmitter class

Return EventEmitter from a function

In this pattern, a constructor function returns an EventEmitter object, which was used to emit events inside a function. This EventEmitter object can be used to subscribe for the events. Consider the following example.

Example: Return EventEmitter from a function

```
var emitter = require('events').EventEmitter;

function LoopProcessor(num) {
    var e = new emitter();

    setTimeout(function () {

        for (var i = 1; i <= num; i++) {
            e.emit('BeforeProcess', i);

            console.log('Processing number:' + i);

            e.emit('AfterProcess', i);
        }
    }
    , 2000)

    return e;
}
var lp = LoopProcessor(3);

lp.on('BeforeProcess', function (data) {
    console.log('About to start the process for ' + data);
});

lp.on('AfterProcess', function (data) {
    console.log('Completed processing ' + data);
});
```

Output:

```
About to start the process for 1
```

```
Processing number:1
```

```
Completed processing 1
```

```
About to start the process for 2
```

```
Processing number:2
```

```
Completed processing 2
```

```
About to start the process for 3
```

```
Processing number:3
```

```
Completed processing 3
```

In the above LoopProcessor() function, first we create an object of EventEmitter class and then use it to emit 'BeforeProcess' and 'AfterProcess' events. Finally, we return an object of EventEmitter from the function. So now, we can use the return value of LoopProcessor function to bind these events using on() or addListener() function.

Extend EventEmitter Class

In this pattern, we can extend the constructor function from EventEmitter class to emit the events.

Example: Extend EventEmitter Class

Copy

```
var emitter = require('events').EventEmitter;

var util = require('util');

function LoopProcessor(num) {
    var me = this;

    setTimeout(function () {

        for (var i = 1; i <= num; i++) {
            me.emit('BeforeProcess', i);

            console.log('Processing number:' + i);

            me.emit('AfterProcess', i);
        }
    }
    , 2000)

    return this;
}

util.inherits(LoopProcessor, emitter)

var lp = new LoopProcessor(3);

lp.on('BeforeProcess', function (data) {
```

```
    console.log('About to start the process for ' + data);
});

lp.on('AfterProcess', function (data) {
    console.log('Completed processing ' + data);
});
```

Output:

```
About to start the process for 1
Processing number:1
Completed processing 1
About to start the process for 2
Processing number:2
Completed processing 2
About to start the process for 3
Processing number:3
Completed processing 3
```

In the above example, we have extended LoopProcessor constructor function with EventEmitter class using util.inherits() method of utility module. So, you can use EventEmitter's methods with LoopProcessor object to handle its own events.

In this way, you can use EventEmitter class to raise and handle custom events in Node.js.

Node.js | Event Loop

Node.js is a single-threaded event-driven platform that is capable of running non-blocking, asynchronously programming. These functionalities of Node.js make it memory efficient. 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. Most operating systems are multi-threaded and hence can handle multiple operations executing in the background. 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. This will be explained further in detail later in this topic.

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.

Example:

```
console.log("This is the first statement");

setTimeout(function(){

    console.log("This is the second statement");

}, 1000);

console.log("This is the third statement");
```

Output:

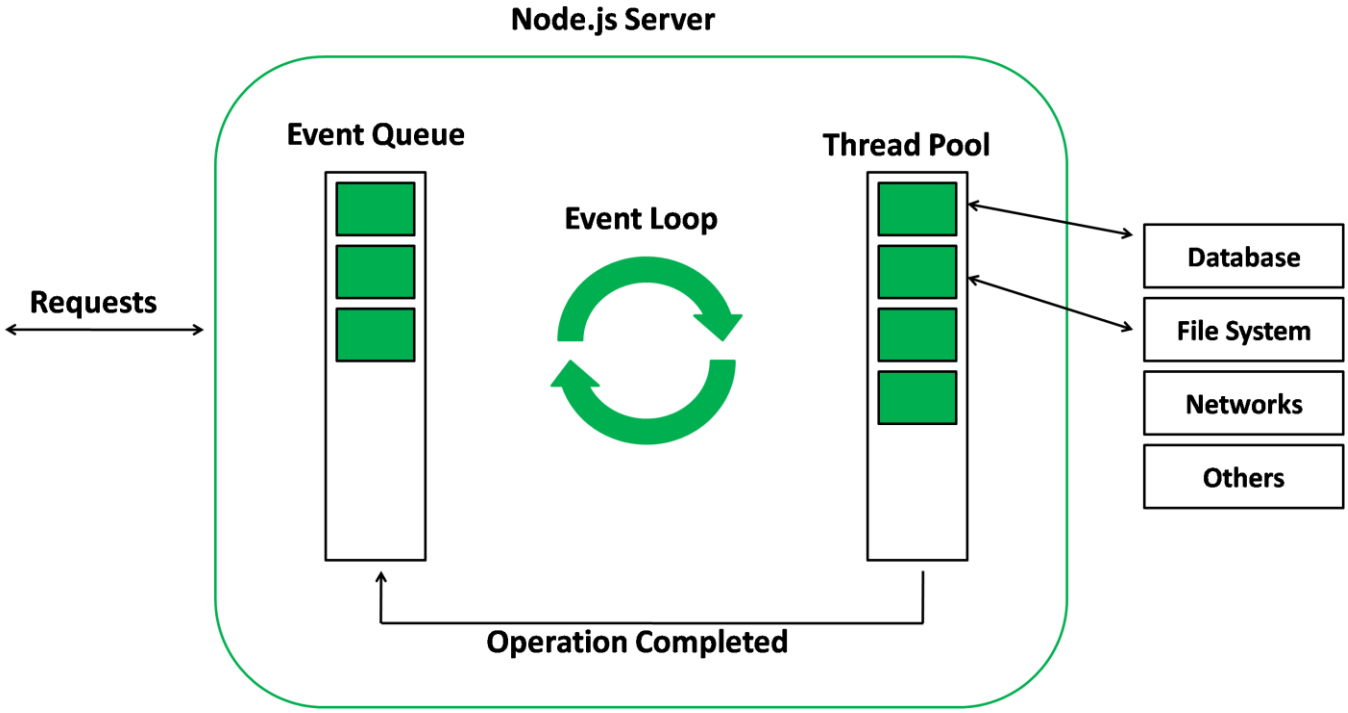
```
This is the first statement
This is the third statement
This is the second statement
```

Explanation: In the above example, the first console log statement is pushed to the call stack and “This is the first statement” is logged on the console and the task is popped from the stack. Next, the setTimeout is pushed to the queue and the task is sent to the Operating system and the timer is set for the task. This task is then popped from the stack. Next, the third console log statement is pushed to the call stack and “This is the third statement” is logged on the console and the task is popped from the stack.
When the timer set by setTimeout function (in this case 1000 ms) runs out, the callback is sent to the event queue. The event loop on finding the call stack empty takes the task at the top of the event queue and sends it to the call stack. The callback function for setTimeout function runs the instruction and “This is the second statement” is logged on the console and the task is popped from the stack.

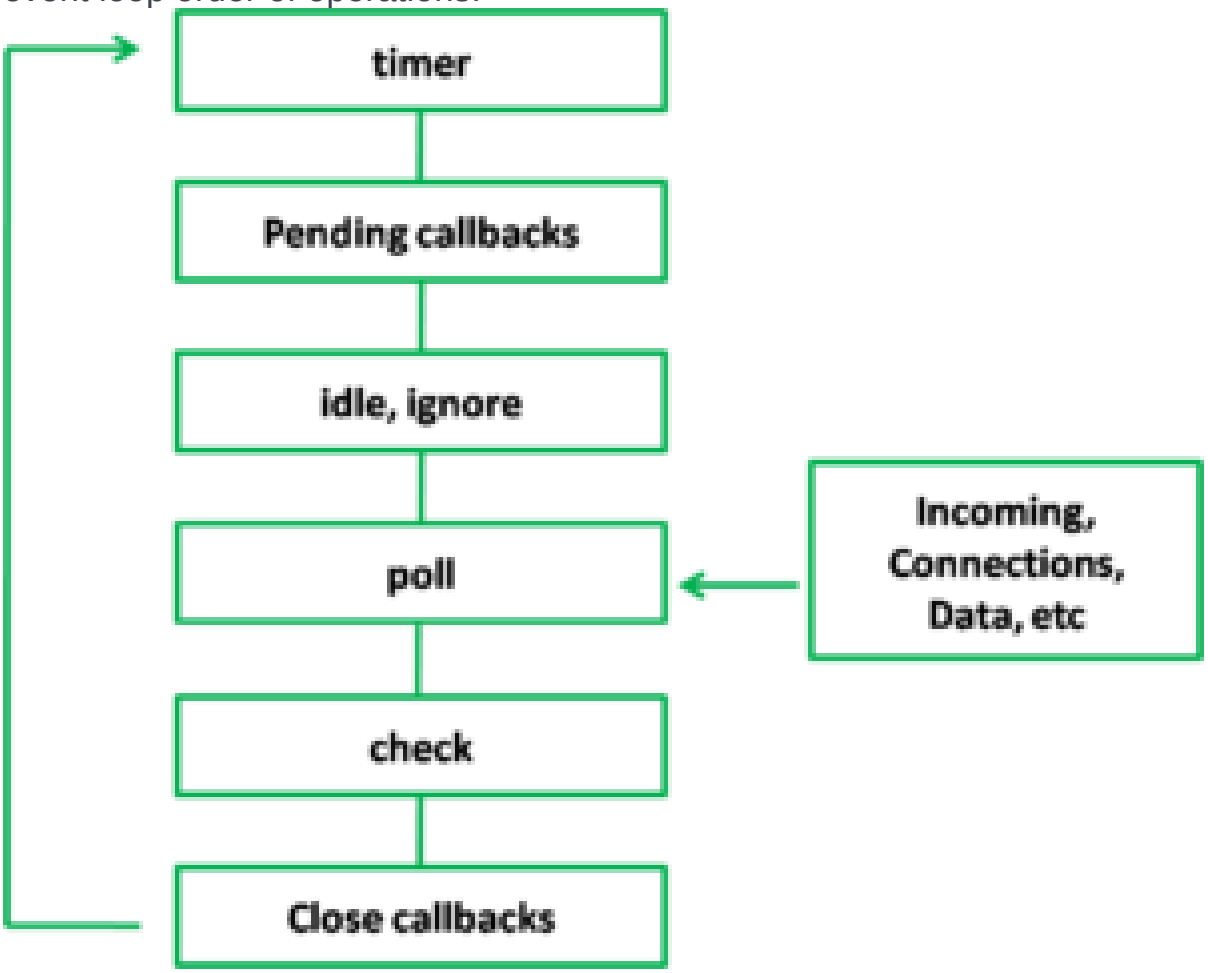
Working of the Event loop: When Node.js starts, it initializes the event loop, processes the provided input script which may make async API calls, schedule timers, then begins processing the event loop. In the previous example, the initial input script consisted of console.log() statements and a setTimeout() function which schedules a timer.
When using Node.js, a special library module called libuv is used to perform async operations. This library is also used, together with the back logic of Node, to manage a special thread pool called the libuv thread pool.This thread pool is composed of four threads used to delegate operations that are too heavy for the event loop. I/O operations, Opening and closing connections, setTimeouts are the example of such operations.

When the thread pool completes a task, a callback function is called which handles the error(if any) or does some other operation. This callback function is sent to the event queue. When the call stack is empty, the event goes through the event queue and sends the callback to the call stack.

The following diagram is a proper representation of the event loop in a Node.js server:



Phases of the Event loop: The following diagram shows a simplified overview of the event loop order of operations:



- **Timers:** Callbacks scheduled by setTimeout() or setInterval() are executed in this phase.

- **Pending Callbacks:** I/O callbacks deferred to the next loop iteration are executed here.
- **Idle, Prepare:** Used internally only.
- **Poll:** Retrieves new I/O events.
- **Check:** It invokes setIntermediate() callbacks.
- **Close Callbacks:** It handles some close callbacks. Eg: socket.on('close', …)

Reference: https://www.geeksforgeeks.org/node-js-events/