Assignment 4.1

Advanced Node.js Concepts and Tools

1. Middleware

Definition: Middleware function are functions that have access to the request object ('req'), the response object ('res'), and the next middleware function in the application's request-response cycle.

Usage:

- Express Middleware: Express is a popular Node.js web framework. Middleware in Express can be used for various purposes like logging, authentication, error handling, etc.

Code →

```
const express = require('express');
const app = express();
// Example of a simple middleware
app.use((req, res, next) => {
  console.log('Time:', Date.now());
  next();
});
```

Types:

- Application-level middleware: Bound to an instance of 'express()'.
- Router-level middleware: Bound to an instance of 'express.Router()'.
- Error-handling middleware: Defined with 4 argument instead of 3: '(err, req, res, next)'.
- Built-in middleware: Provided by Express. eg: 'express.json()', 'express.urlencoded()'.
- Third-party middleware: Installed via npm, eg 'morgan', 'body-parser'.

2. Asynchronous Programming

Concept: Node.js is designed to be asynchronous and non-blocking, using an event-driven architecture.

Event Loop:

- Phases: Timers, I/O callbacks, idle/prepare, poll, check and close callbacks.
- Callbacks: Functions that are called after a certain task is completed.

Callback Hell:

- Nested callbacks can lead to deeply nested code, often referred to as "callback Hell".

Code →

```
fs.readFile('/path/to/file', (err, data) => {
    if (err) throw err;
    console.log(data);
});
```

Promises:

- Objects representing the eventual completion or failure of an asynchronous operation.

Code →

```
const fs = require('fs').promises;

fs.readFile('/path/to/file')
    .then(data => console.log(data))
    .catch(err => console.error(err));
```

Async / Await:

- Syntatic sugar over promises, allowing for cleaner and more readable asynchronous code.

Code →

```
async function readFile() {
    try {
        const data = await fs.readFile('/path/to/File');
        console.log(data);
    } catch (err) {
        console.error(err);
    }
}
```

3. Event Emitter

Concept: The 'events' module provides the 'EventEmitter' class, which is key to working with events in Node.js.

Basic Usage:

- Emit: Trigger an event.
- On: Listen for an event.

Code →

```
const EventEmitter = require('events');
const myEmitter = new EventEmitter();

myEmitter.on('event', () => {
    console.log('An event occurred!');
});
myEmitter.emit('event');
```

Custom Events:

- You can create and handle custom events.

Code →

```
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();

myEmitter.on('customEvent', (msg) => {
    console.log(msg);
});
myEmitter.emit('customEvent', 'Hello, this is a
    custom event!');
```

14. Streams:

Concept: Streams are objects that lets you read data from a source or write data to a destination in a continuous fashion.

Types:

- Readable: Streams from which data can be read.
- Writable: Streams to which data can be written.
- Duplex: Streams that are both readable and writable.
- Transform: Duplex streams that can be modify or transform the data as it is written and read.

Code =>

```
const fs = require('fs');

// Readable Stream
const readStream = fs.createReadStream('input.txt');
readStream.on('data', (chunk) => {
    console.log('Received ${chunk.length} bytes of data.');
});

// Writable Stream
const writeStream = fs.createWriteStream('output.txt');
writeStream.write('Hello, world!', 'utf8');
writeStream.end();
```

Pipe :

- A mechanism for connecting the output of one stream to the input of another.

code →

```
Const readStream = fs. CreateReadStream ('input.txt');
Const writeStream = fs. createWriteStream ('output.txt');

readStream. pipe (writeStream);
```

5. Buffer

Concept : Buffers are used to handle binary data directly in Node.js

Usage :

- Useful for dealing with binary data, such as file and network protocols.

Code →

```
const buf = Buffer.from ('hello', 'utf8');

Console.log (buf); // <Buffer 68 65 6c 6c 6f>

Console. log (buf. to string ()); // 'hello'
```

Operations :

- Create a buffer : 'Buffer .alloc (size)', 'Buffer .from (array)'.
- Write to buffer : 'buf. write (string)'.
- Read from buffer :

'buf. to String (encoding)'.

6. File System (fs)

Synchronous vs Asynchronous:

- Synchronous : Blocking operations.
- Asynchronous : Non-blocking operations, recommended for I/O operations.

Basic Operations:

- Read file:

code →

```
const fs = require('fs');

fs.readFile('path/to/file', 'utf8', (err, data) => {

  if (err) throw err;

  console.log(data);

});
```

- Write file:

code →

```
fs.writeFile('path/to/file', 'Hello, world!', (err) => {

  if (err) throw err;

  console.log('File has been saved!');

});
```

- Append file:

```
fs.appendFile('path/to/file', 'Data to append', (err) => {
  if (err) throw err;
  console.log('Data was appended to file!');
});
```

- Delete File:

Code →

```
fs.unlink('path/to/file', (err) => {
  if (err) throw err;
  console.log('File deleted!');
});
```

7. Child Processes

Concept: Node.js allows the creation of child processes to execute other applications or commands.

Methods:

- exec: Runs a command in a shell and buffers the o/p.

Code →

```
const { exec } = require('child_process');
exec('ls', (error, stdout, stderr) => {
  if (error) {
    console.error(`exec error: ${error}`);
    return;
  }
  console.log(`stdout: ${stdout}`);
  console.error(`stderr: ${stderr}`);
});
```

- Spawn : Launches a new process with a given command.

Code →

```
const {spawn} = require ('child_process');

const ls = spawn ('ls', ['-lh', '/usr']);

ls.stdout.on ('data', (data) => {
    console.log ('stdout : ${data}');
});

ls.stderr.on ('data', (data) => {
    console.error ('stderr : ${data}');
});

ls.on ('close', (code) => {
    console.log ('child process exited with code ${code}');
});
```

- fork: A special case of spawn to create new Node.js processes.

Code →

```
const {fork} = require ('child_process');

const forked = fork ('script.js');

forked.on ('message', (msg) => {
    console.log ('Message from child ${msg}');
});

forked.send ({ hello: 'world' });
```