Name - Sandarbh Singhal Role - full stack Web Dclopment

Advanced concept of Node.js Cheatsheet

Middleware

- Define - Middleware functions 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 - Middlware can execute code, modify the request and response objects, end the request-response cycle, and call the next middleware functions
- Types of Middleware
  → Application-level middleware : Bound to an instance of ~~'se~~ 'express()';
  → Router-level Middleware : Bound to an instance of 'express.Router()'.

  Ex

```
const express = require('express');
const app = express();
app.use((req, res, next) => {
    // function definition
});
const router = express.Router();
router.use((req, res, next) => {
    // function definition.
});
```

  → Error-handling middleware : Defined with four arguement. ((err, req, res, next) => { })

→ Builin Middleware - Provided By express. eg → 'express.json()'.

→ Third Party Middleware - Installed via npm eg → 'morgan', 'cors',

Asynchronous Programming

- Callback functions - functions passed as arguements to other function and executed after the completion of a given task.

Eg →
```
fs.readFile('file.txt', (err, data) => {
        if (err) throw err;
        console.log(data);
    });
```

- Promises : Objects representing the eventual completion of an asynchronous operation and its resulting value.

Eg →
```
const promise = new Promise ((resolve, reject) => {
        if (success){
            resolve(result);
        } else {
            reject(error);
        }
    });
promise.then(result => {
        console.log(result);
  }).catch(error => {
        console.error(error);
  });
```

- Async / Await : Syntactic sugar built or promises, making asynchronous code look synchronous.

Eg →
```
async function fetchData () {
        try {
            const response = await fetch ("https://api.example.com");
            const data = await response.json();
            console.log(data);
        } catch (error){
            console.error ('Error:', error);
        }
}
```

Event Loop

- Definition - The event loop is what allows Node.js to perform non-blocking I/O operations.
- Phases of Event Loop:
  - Timers: Executes callbacks scheduled by 'setTimeout()' and 'setInterval()'.
  - Pending Callbacks: Executes I/O callbacks deferred to the next loop iteration.
  - Idle, Prepare: Only used internally.
  - Poll: Retrieves new I/o events; executes I/O callbacks.
  - Check: Executes 'setImmediate()' callbacks
  - Close Callbacks: Executes close event callbacks.

Streams

- Definition: Objects that let you read data from a source or write data to a destination in a continuous fashion.
- Types
  - Readable Streams: Stream from which data can be read.
  - Writable Streams: Stream to which data can be written.
  - Duplex Streams: Stream that is both readable and writable.
  - Transform Streams: Duplex streams where the output is commnected based on the input.

Buffers

- Definition - Buffers are used to handle binary data in Node.js.

Usage :
```
const buffer = Buffer.from('Hello');
console.log(buffer.toString());
console.log(buffer[0]);
```

Cluster Module

- Definition: Enable the creation of child processes that share the same server port.

- Usage ::

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;
if(cluster.isMaster){
    for(let i=0; i<numCPUs; i++){
        cluster.fork();
    }
    cluster.on('exit', (worker, code, signal) => {
        console.log(`Worker ${worker.process.pid} died`);
    });
} else {
    http.createServer((req, res) => {
        res.writeHead(200);
        res.end('Hello World\n');
    }).listen(8000);
}
```