Reg no: 73772214103

Name: Ackshaya R

Class: CSE – 'A' 3rd yr

Subject code:60 CS E12

Subject name: Node.js and React.js

ASSIGNMENT 01

01. Write blocking and non-blocking code with sample example**.**

Blocking Code:

- Blocking code in Node.js means that each operation must finish completely before the next one starts.
- This approach can lead to performance issues, especially with I/O-bound tasks like reading from the disk or fetching data from an API, as the entire system might halt and wait for one task to finish.

Example code for Blocking code:

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

console.log("Start Reading File...");

const data = fs.readFileSync('example.txt', 'utf8');

console.log("File Data:", data);

console.log("End of Program.");
```

Non-Blocking Code:

- Non-blocking code, on the other hand, allows multiple operations to run simultaneously without waiting for each other to complete.
- Node.js uses an event-driven architecture that works well with non-blocking code, making it efficient for handling a large number of concurrent operations

Example code for Non-Blocking code:

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

console.log("Start Reading File...");
fs.readFile('example.txt', 'utf8', (err, data) => {
    if (err) throw err;
    console.log("File Data:", data);
  });
  console.log("End of Program.");
```

Pros and Cons:

Blocking Code	**Non-Blocking Code**
Simpler to write and understand.	More efficient for I/O-heavy tasks.
Can lead to poor performance.	Scales better under high load.
Blocks the execution of further code.	Code execution continues without waiting.

---

02. Write file system module with sample coding

- The fs module in Node.js is crucial for interacting with the file system. You can create, read, write, delete, or modify files and directories using various methods provided by the fs module.
- It provides both synchronous (blocking) and asynchronous (non-blocking) methods.

Common Operations with the fs Module:

1. <u>Creating/Opening a File:</u>

```
const fs = require('fs');
fs.writeFileSync('newfile.txt', 'Hello World!', (err) => {
    if (err) throw err;
    console.log('File created successfully!');
});
```

2. Reading from a File:

```
fs.readFile('newfile.txt', 'utf8', (err, data) => {
   if (err) throw err;
   console.log('File Content:', data);
});
```

3. Appending to a File:

```
fs.appendFile('newfile.txt', ' Additional data!', (err) => {
   if (err) throw err;
   console.log('Content added successfully!');
});
```

4. Renaming a File:

```
fs.rename('newfile.txt', 'renamedfile.txt', (err) => {
   if (err) throw err;
   console.log('File renamed successfully!');
});
```

5. Deleting a File:

```
fs.unlink('renamedfile.txt', (err) => {
   if (err) throw err;
   console.log('File deleted successfully!');
});
```

Understanding Synchronous vs. Asynchronous File Operations:

- Synchronous operations are simpler but can degrade performance, especially for large-scale applications.
- Asynchronous methods return control to the event loop immediately, making them better for scaling applications

03. Develop the REPL program to find odd or even number.

- A REPL (Read-Eval-Print-Loop) is an interactive shell where the user inputs code, the system evaluates it, and the output is printed
- In Node.js, we can implement a simple REPL to check whether a number is odd or even.

  .

Example code:

```
const readline = require('readline');
const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});
function checkOddOrEven() {
    rl.question('Enter a number: ', (input) => {
        const number = parseInt(input);
        if (isNaN(number)) {
            console.log('Please enter a valid number.');
        } else {
            if (number % 2 === 0) {
                console.log(`${number} is an Even Number.`);
            } else {
                console.log(`${number} is an Odd Number.`);
            }
        }
        checkOddOrEven();
    });
}
checkOddOrEven();
```

04. Develop DNS module with sample coding

- The DNS module in Node.js is primarily used for name resolution, i.e., converting domain names into IP addresses (forward lookup) or resolving IP addresses into domain names (reverse lookup).
- It also provides the capability to resolve different types of DNS records (e.g., A, MX, TXT, SRV) and handle both IPv4 and IPv6.
- The DNS module uses libuv (Node.js' asynchronous I/O library) for non-blocking DNS operations, which means that DNS requests do not block other operations in your application.

Common Methods in the DNS Module:

1. **dns.lookup()**:
   - Resolves a domain name into an IP address, similar to the system's native DNS lookup (which uses /etc/hosts on Unix systems or DNS configurations).
2. **dns.resolve()**:
   - Performs DNS resolution without using the underlying operating system's facilities. Can retrieve various DNS records like A, AAAA, CNAME, etc.
3. **dns.resolve4()**:
   - Specifically resolves only IPv4 addresses for a given domain.
4. **dns.resolve6()**:
   - Specifically resolves only IPv6 addresses for a given domain.
5. **dns.reverse()**:
   - Resolves an IP address into a domain name (reverse lookup).
6. **dns.resolveMx()**:
   - Resolves the MX (Mail Exchange) records for a domain.
7. **dns.resolveTxt()**:
   - Retrieves the TXT records for a domain.

Example:

```
const dns = require('dns');
dns.lookup('www.google.com', (err, address, family) => {
  if (err) throw err;
  console.log('IP Address:', address);
  console.log('Address Family:', family);
});
dns.resolve4('www.google.com', (err, addresses) => {
  if (err) throw err;
  console.log('IP Addresses:', addresses);
});
```

Handling Errors in DNS Lookups:

Each DNS method provides an err object as the first parameter in the callback function. This allows you to handle potential errors in DNS queries, such as:

- Invalid domain names.
- Network issues preventing DNS resolution.
- Timeouts during the lookup process.

05. Develop TCP server and client program

- TCP (Transmission Control Protocol) ensures reliable communication between the server and client.
- Node.js has built-in support for TCP servers and clients using the net module.

TCP Server Example:

```
const net = require('net');
const server = net.createServer((socket) => {
   console.log('Client connected');
   socket.on('data', (data) => {
      console.log('Received from client:', data.toString());
      socket.write('Hello from server!');
   });
   socket.on('end', () => {
      console.log('Client disconnected');
   });
});
server.listen(8080, () => {
   console.log('Server is listening on port 8080');
});
```

TCP Client Example:

```
const net = require('net');
const client = net.createConnection({ port: 8080 }, () => {
   console.log('Connected to server');
   client.write('Hello from client!');
});
client.on('data', (data) => {
   console.log('Received from server:', data.toString());
   client.end(); // Close the connection after receiving data
});
```

```
 client.on('end', () => {
    console.log('Disconnected from server');
});
```

In this example:

- The **server** listens on port 8080 and sends a greeting message to the client when a connection is established.
- The **client** connects to the server and exchanges messages