Date:- 12/02/2025

Page 40

Module 03

Node.js.

1. Define Node.js and explain its role in modern web development.

→

- Node.js is an open-source, cross-platform JavaScript runtime environment that allows JavaScript to be executed on the server side.
- It's built on Google Chrome's V8 JavaScript engine, which makes it fast and efficient.
- In modern web development, Node.js plays a crucial role by enabling developers to build scalable and high-performance applications.
- It allows the use of JavaScript for both client-side and server-side development, which simplifies the development process.
- Its non-blocking, event-driven architecture is ideal for building real-time applications like chat apps, APIs and streaming services.

2. Describe how Node.js handles file access operations and why it preferred for I-O bound tasks.

→

- Node.js handles file access operations using it built-in fs (file Systems) module. It offers both asynchronous (non-blocking) and synchronous (blocking) methods for file operations.

- Asynchronous Example:-

```
const fs = require('fs');
fs.readFile('nitan.txt', 'utf8', (err, data) => {
    if (err) throw err;
    console.log(data);
});
```

- Synchronous Example:-

```
const data = fs.readFileSync('nitan.txt', 'utf8');
console.log(data);
```

- Preferred for I/O-bound tasks because Node.js's non-blocking nature allows it to handle multiple file operations simultaneously without slowing down performance, making it ideal for applications that require frequent file or data access.

3. Write a simple Node.js script that prints "Hello, world!" to the console.

→

```
const http = require('http');
const server = http.createServer((req, res) => {
    res.writeHead(200, {'Content-Type':
        'text/plain'});
    res.end('Hello, world!');
});

server.listen(3000, () =>
```

```
    console.log('Server running at http://
        localhost:3000/');
});
```

4. Implement a simple TCP server using Node.js that listens on port 3000 and logs "client connected" when a client connects.

→

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

const server = net.createServer((socket) =>{
    console.log('client connected');
});

server.listen(3000, () =>{
    console.log('Server listening on port
        3000');
});
```

5. Develop a Node.js script that recursively reads all files in a directory and its subdirectories, logging their names to the console.

→

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

function readFileRecursively(dir){
    fs.readdir(dir, (err, files) => {
```

```
    if (err) throw err;
    files.forEach(file => {
        const filePath = path.join(dir, file);
        fs.stat(filePath, (err, stats) => {
            if (err) throw err;
            if (stats.isDirectory()) {
                readFilesRecursively(filePath);
            } else {
                console.log(filePath);
            }
        });
    });
}
readFilesRecursively('./');
```

6. Explain the purpose of timers in Node.js and provide example of when they might be used.

→

- Timers in Node.js are used to schedule the execution of code after a specified amount of time or at regular intervals.
- The Primary timer functions are :-

i) setTimeout() :- Executes a functions once after a delay.

ii) setInterval() :- Execute a functions repeatedly at specified intervals.

iii) setImmediate() :- Execute a function immediately after the current event loop cycle.

iv) clearTimeout() & clearInterval() :-
cancel scheduled timers.

- Use Cases :-

- Delaying execution of code.
- Repeatedly performing tasks like checking for updates or refreshing data.
- scheduling background tasks.

7. Discuss the significance of events and the event loop in Node.js applications.

→ Events in Node.js :-

- Node.js is built on an event-driven architecture, which allows it to handle asynchronous operations efficiently.
- Events are emitted when something happens and event listeners respond to those events. The EventEmitter class from the events module is used to handle events in Node.js.
- Example :-

```
const EventEmitter = require('events');
const emitter = new EventEmitter();
emitter.on('greet', () => {
    console.log('Hello from the event!');
});
emitter.emit('greet');
```

→ Event Loop in Node.js :-

- The event loop is the mechanism that allows Node.js to perform non-blocking I/O operation
- It continuously checks the event queue and executes the corresponding callbacks when events occur.
- Significance :-
  - Handles asynchronous operations like file I/O, network requests and timers.
  - Allows Node.js to perform efficiently, even with high concurrency.
  - Prevents blocking of code, leading to better performance and responsiveness.

8. Implement a Node.js script to read the contents of a text file and display them in the console.

→

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

//Asynchronous read
fs.readFile('input.txt', function(err, data) {
    if (err) {
        return console.error(err);
    }
    console.log("Asynchronous read: " +
                data.toString());
});
```

→

```
//Synchronous read
var data = fs.readFileSync('input.txt');
  console.log("Synchronous read : " +
        data.toString());
console.log("Program Ended!");
```

9. Ai create a basic REST API using Node.js that contents of a responds with "Hello -API!" when accessed via an HTTP GET request.

→

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

const server = http.createServer((req, res)=>
  if(req.method === "GET" &&
     req.url === '/api'){
    res.writeHead(200, {'content-type':
                  'text/plain'});
    res.end('Hello, API!');
  }else{
    res.writeHead(404, {'content-Type':
                  'text/plain'});
    res.end('Not Found');
  }
});

server.listen(3000, () => {
  console.log('API running at http://
     localhost:3000/api');
});
```

10. Write a Node.js script that uses setTimeout to print "Delayed Message" after 2 seconds.

→

```
const util = require('util');
console.log('Timer started, waiting for 2
                    Seconds ... ');
setTimeout(() => {
    console.log('Delayed Message : ');
}, 2000);

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

11. Create Node.js application that use the async/await syntax to perform asynchronous file I/O operations, such as reading and writing files, in a synchronous-like manner.

→

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

async function performFileOperations() {
  try {
    await fs.writeFile('example.txt',
        'Hello, this is an async/await
          example!');
    console.log('File written successfully.');

    const data = await fs.readFile('example.
        txt', 'utf8');
    console.log('File content:', data);
```

```
    }
    catch (err) {
        console.error('Error occurred:',
            err);
    }
}

performFileOperation();
```

12. Explain the difference between synchronous and asynchronous programming in Node.js, providing examples of each.

→

Synchronous	Asynchronous
- Blocks execution of code.	- Non-blocking, allows parallel execution.
- Slower for I/O-bound tasks.	- Faster and more efficient for I/O-bound tasks.
- Easier to understand for beginners.	- Requires callbacks, promises or async/await.

→ Example:-

- Synchronous:-

```
const fs = require('fs');
const data = fs.readFileSync('notes.txt',
                'utf8');
```

```
console.log(data);
console.log('this will print after reading
            the file');
```

- Asynchronous:-

```
const fs = require('fs');
fs.readFile('myFav.txt', 'utf8', (err, data) => {
    if (err) throw err;
    console.log(data);
});
console.log('this will print before reading
            the file');
```

13. Demonstrate the concept of a REST API and how Node.js ~~and the differences~~ can be used to create and consume RESTful services with the help of an examples.

→

```
const express = require('express');
const app = express();
const port = 3000;

app.use(express.json());

app.get('/api/message', (req, res) => {
    res.send('Hello, REST API!');
});

app.post('/api/message', (req, res) => {
    const message = req.body.message;
```

```
    res.send('Received message:
        ${message}');
});

app.listen(port, () => {
    console.log('Server is running on
        http://localhost:${port}');
});
```

14. Describe how TCP and HTTP servers can be created in Node.js and the differences between them in terms of communication protocols and use cases.

→

1) **TCP Server**:-

- TCP (Transmission Control Protocol) is a connection-oriented protocol that ensures reliable, ordered communication between devices.
- used for low-level data transmission (e.g., chat applications, file transfers).

- **Creating a TCP Server**:-

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

const server = net.createServer((socket) => {
    console.log('client connected.');
    socket.write('welcome to the TCP server!');
```

```
socket.on('data', (data) => {
    console.log('Received : ${data}');
});

socket.on('end', () => {
    console.log('client disconnected.');
});
});

server.listen(3000, () => {
    console.log('TCP Server running on port
                3000');
});
```

2. HTTP server :-

- HTTP (Hypertext Transfer Protocol) is stateless communication, commonly used for web applications.
- Used for serving web content or creating RESTful APIs.

- Creating an HTTP server :-

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

const server = http.createServer((req, res) => {
    res.writeHead(200, {'content-Type':
                        'text/plain'});
    res.end('Hello from HTTP Server!');
});
```

```
server.listen(3000, () => {
    console.log('HTTP server running on
                port 3000');
});
```

15. Build a RESTful API using Node.js and Express that performs CRUD operations (Create, Read, Update, Delete) on a mock database (e.g., an array of objects).

→

```
const express = require('express');
const app = express();
const port = 3000;

app.use(express.json());

let users = [
    {id: 1, name: 'Alice'},
    {id: 2, name: 'Bob'}
];

// create
app.post('/users', (req, res) => {
    const newUser = {id: users.length + 1,
                     ...req.body};
    users.push(newUser);
    res.status(201).json(newUser);
});
```

→

```
//read
app.get('/users', (req, res) => {
    res.json(users);
});

//update
app.put('/users/:id', (req, res) => {
    const userId = parseInt(req.params.
                            id);
    const user = users.find(u => u.id ===
                            userId);
    if(!user) {
        return res.status(404).send('User
                not found');
    }
    user.name = req.body.name;
    res.json(user);
});

//delete
app.delete('/users/:id', (req, res) => {
    const userId = parseInt(req.params.
                            id);
    users = users.filter(u => u.id !==
                            userId);
    res.status(204).send();
});

app.listen(port, () => {
    console.log('REST API is running at
        http://localhost:${port}');
});
```

16. Build a TCP chat server using Node.js & Express that allows multiple client to connect, send message and receive messages from other connected clients.

→

```
const net = require('net');
let clients = [];

const server = net.createServer((socket) => {
    console.log('A new client has connected.');

    clients.push(socket);
    socket.write('welcome to the chat server!\n');
    socket.on('data', (data) => {
        const message = data.toString().trim();
        console.log('Received:', message);

        clients.forEach((client) => {
            if (client !== socket) {
                client.write('client says: ${message}\n');
            }
        });
    });

    socket.on('end', () => {
        console.log('A client has disconnected.');
        clients = clients.filter((client) => {
            client !== socket;
        });
```

```
socket.on('error', (err) => {
    console.log('Error:', err.message);
});
});

server.listen(3000, () => {
    console.log('TCP chat server running
                on port 3000');
});
```

17. Build a RESTful API using Node.js and MongoDB (or other Database of your choice) for managing user accounts, including features such as user registration, authentications and profile updates.

→

a. Setup and Install:-

npm install express mongoose bcryptjs jsonwebtoken.

b. Connecting to MongoDB:-

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

mongoose.connect('mongodb://localhost:
    27017/userDB', {
    useNewUrlParser: true,
    useUnifiedTopology: true
});
.then(() => console.log('MongoDB connected'))
.catch(err => console.error('MongoDB
    connection error:', err));
```

c. Creating a User Model :-

```
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');

const userSchema = new mongoose.Schema({
    username: { type: String, required: true,
                unique: true },
    password: { type: String, required: true,
                unique: true },
    email: { type, String, required: true,
             unique: true }
});

userSchema.pre('save', async function
    (next) {
    if (!this.isModified('password'));
    this.password = await bcrypt.hash
                (this.password, 10);
    next();
});

const User = mongoose.model('User',
                userSchema);
module.exports = User;
```

d. Creating the RESTful API :-

```
const express = require('express');
const User = require('./models/User');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
```

```
const app = express();
app.use(express.json());
const secretKey = 'your-secret-key';

app.post('/register', async (req, res) => {
  const {username, password, email} = req.body;
  try {
    const newUser = new User({username, password, email});
    await newUser.save();
    res.status(201).send('User registered successfully!');
  } catch (err) {
    res.status(400).send('Error registering user: ' + err.message);
  }
});

app.post('/login', async (req, res) => {
  const {username, password} = req.body;
  const user = await User.findOne({username});

  if(!user) return res.status(400).send('User not found.');

  const isMatch = await bcrypt.compare(password, user.password);
  if (!isMatch) return res.status(400).send('Invalid credentials');
```

```
    const token = jwt.sign({id: user._id},
        secretKey, {expiresIn: '1h'});
    res.json({token});
});

app.get('/profile', authenticateToken,
    async(req, res) => {
    const user = await User.findById
        (req.user.id).select('-password');
    res.json(user);
});

function authenticateToken(req, res, next)
    const token = req.headers['authori-
        zation'];
    if(!token) return res.sendStatus(403);
    jwt.verify(token, secretKey, (err, user) =>
    {
        if(err) return res.sendStatus(403);
        req.user = user;
        next();
    });
}

app.listen(3000, () => {
    console.log('RESTful API running on
        http://localhost:3000');
});
```