**Batch- T4**

**Practical No. 9**

**Title of Assignment: Study and implementation of node.js**

**Student Name: Daivik Karbhari**

**Student PRN: 22510111**

**Problem Statement:**

**Problem Statement 1: Database Connectivity using SQL or Oracle**

Write a Node.js program that connects to an Oracle/SQL database, retrieves data from a table, and displays the results.

**Problem Statement 2: Middleware (Express.js)**

**1. What is Middleware in Node.js (particularly in the context of Express.js)?**

In the context of Express.js, middleware refers to functions that have access to the request (req), response (res), and the next middleware function in the request-response cycle. Middleware functions can perform various operations, such as:

• Executing code.

• Modifying the req or res objects.

• Ending the request-response cycle by sending a response.

• Calling the next() function to pass control to the next middleware function.

Middleware can be used to handle things like authentication, logging, data validation, error handling, and more.

**2. How do you create custom middleware in Express.js?**

You can create custom middleware in Express.js by defining a function that takes three parameters: req, res, and next. Inside this function, you can perform any logic, modify the request or response objects, and either terminate the request-response cycle or pass the control to the next middleware using the next() function.

Here's an example of a custom middleware that logs the request method and URL:

const express = require('express');

const app = express();

// Custom middleware function

const logger = (req, res, next) => {

console.log(`${req.method} ${req.url}`);

next(); // Pass control to the next middleware or route handler

};

// Use the middleware globally

app.use(logger);

// Sample route

app.get('/', (req, res) => {

res.send('Home Page');

});

app.listen(3000, () => {

console.log('Server is running on http://localhost:3000');

});

**3. How is middleware executed in order in an Express.js application?**

In Express.js, middleware functions are executed in the order they are defined. This is important because middleware can either:

• Modify the request or response objects.

• Terminate the request-response cycle (e.g., by sending a response).

• Pass control to the next middleware or route handler using the next() function.

The sequence in which middleware is executed follows a top-down approach. Once a middleware function is executed, the next() function is called to pass control to the next middleware in the chain. If next() is not called, the request-response cycle will be terminated, and the next middleware or route handler will not be executed.

Execution Flow:

1. Middleware functions are executed in the order they are registered using app.use() or directly in the route.

2. If next() is called, the next middleware or route handler will be executed.

3. If next() is not called, the response is sent, and the cycle stops.

Example:

const express = require('express');

const app = express();

// First middleware

app.use((req, res, next) => {

console.log('Middleware 1');

next(); // Pass control to the next middleware

});

// Second middleware

app.use((req, res, next) => {

console.log('Middleware 2');

next(); // Pass control to the next middleware or route handler

});

// Route handler

app.get('/', (req, res) => {

console.log('Route handler');

res.send('Hello, World!');

});

app.listen(3000, () => {

console.log('Server running on http://localhost:3000');

});

Execution order:

1. The request enters Middleware 1 (console.log('Middleware 1')).

2. next() is called, so Middleware 2 is executed (console.log('Middleware 2')).

3. next() is called again, so the route handler is executed (console.log('Route handler')).

4. The response 'Hello, World!' is sent.

If any middleware function does not call next(), the request-response cycle will stop at that point, and no further middleware or route handlers will be executed.

**Problem Statement 3: File System (fs) Module**

**1. How do you read and write files using the fs module in Node.js?**

In Node.js, the fs (File System) module allows you to interact with the file system to perform operations like reading and writing files.

To **read** a file asynchronously:

const fs = require('fs');

fs.readFile('example.txt', 'utf8', (err, data) => {

if (err) {

console.error(err);

return;

}

console.log(data); // Output the content of the file

});

To **write** to a file asynchronously:

const fs = require('fs');

fs.writeFile('example.txt', 'Hello, World!', (err) => {

if (err) {

console.error(err);

return;

}

console.log('File has been written');

});

Both operations above are asynchronous, meaning the file operations are handled in the background, and the program can continue running while waiting for the results.

**2. What is the difference between fs.readFile() and fs.readFileSync()?**

* **fs.readFile()**: This is an **asynchronous** function. When you use it, Node.js doesn’t block the execution of subsequent code. Instead, you pass a callback function, which will be executed once the file is read. This is suitable for applications where you want to maintain a non-blocking, asynchronous flow.

Example:

fs.readFile('example.txt', 'utf8', (err, data) => {

if (err) throw err;

console.log(data);

});

console.log('This line will run before the file content is printed.');

* **fs.readFileSync()**: This is a **synchronous** function. It blocks the execution of subsequent code until the file has been read. This can be simpler to use, but it can make your application less performant because other tasks are blocked until the file is fully read.

Example:

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

console.log(data);

console.log('This line runs after the file content is printed.');

**Key difference**:

* fs.readFile() (asynchronous) doesn’t block the program while the file is being read.
* fs.readFileSync() (synchronous) blocks the program until the file is fully read.

**3. How can you check if a file or directory exists in Node.js?**

You can check if a file or directory exists using fs.existsSync() or fs.access():

**Synchronous** way using fs.existsSync():

const fs = require('fs');

const fileExists = fs.existsSync('example.txt');

console.log(fileExists ? 'File exists' : 'File does not exist');

**Asynchronous** way using fs.access():

const fs = require('fs');

fs.access('example.txt', fs.constants.F\_OK, (err) => {

console.log(err ? 'File does not exist' : 'File exists');

});

fs.constants.F\_OK checks for file existence, but other constants like R\_OK and W\_OK can be used to check for read or write permissions.

**4. How do you handle file operations in an asynchronous manner?**

In Node.js, you handle file operations asynchronously by using the **callbacks** or **promises** approach.

**1. Using callbacks:**

Most file operations in the fs module have asynchronous versions where you pass a callback to handle errors or the results.

Example of reading a file asynchronously using a callback:

const fs = require('fs');

fs.readFile('example.txt', 'utf8', (err, data) => {

if (err) {

console.error('Error reading file:', err);

return;

}

console.log('File content:', data);

});

Example of writing to a file asynchronously using a callback:

fs.writeFile('example.txt', 'Hello, World!', (err) => {

if (err) {

console.error('Error writing file:', err);

return;

}

console.log('File successfully written.');

});

**2. Using Promises (fs.promises) or async/await:**

The fs module also provides promise-based versions of file operations, which allow you to use async/await for cleaner code.

To read a file using promises:

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

async function readFileAsync() {

try {

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

console.log('File content:', data);

} catch (err) {

console.error('Error reading file:', err);

}

}

readFileAsync();

To write to a file using promises:

async function writeFileAsync() {

try {

await fs.writeFile('example.txt', 'Hello, World!');

console.log('File successfully written.');

} catch (err) {

console.error('Error writing file:', err);

}

}

writeFileAsync();

Using **promises** or **async/await** helps in avoiding "callback hell" and makes the code more readable, especially when chaining multiple asynchronous file operations.

**Problem Statement 4: File Upload and Download API**

* Develop a file upload and download API using Node.js and Express. The API should allow users to upload files (e.g., images, documents) and download them later.
* Create an API to upload files to the server.
* Implement routes to retrieve and download files.
* Ensure proper error handling (e.g., file size limits, invalid file formats).
* Implement file versioning to allow multiple uploads of the same file name without overwriting.

**Problem Statement 5: Real-time Chat Application with Socket.io**







