**Batch: T7**

**Practical No: 9**

**Title of Assignment: Study and Implementation of Nodejs**

**Student Name: Shreeyash Shripad Dongarkar**

**Student PRN: 22510025**

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

**1) 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 context of Express.js?**

=> In the context of **Express.js**, **middleware** refers to 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. Middleware functions can perform tasks such as:

* Executing code
* Modifying the req and res objects
* Ending the request-response cycle
* Calling the next middleware function

If a middleware function doesn't end the request-response cycle (e.g., by sending a response), it must call next() to pass control to the next middleware function. Otherwise, the request will hang.

**Key Features of Middleware in Express.js**

1. **Executed in Order**: Middleware is executed in the order it is defined in your Express app.
2. **Chained Execution**: Middleware functions can pass control to the next function by calling next(). Without calling next(), the request won't move forward in the chain.
3. **Multiple Uses**: Middleware can be used to handle a wide range of tasks like:
   1. Request Logging
   2. Request Parsing (JSON, form data)
   3. Authentication
   4. Error Handling

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

=> Creating custom middleware in Express.js is straightforward. You simply define a function that takes three arguments: req, res, and next. This function can perform any tasks you want—like logging requests, modifying the request or response objects, or handling errors—and must call next() to pass control to the next middleware in the stack (unless you end the request-response cycle by sending a response).

**Steps to Create Custom Middleware**

1. **Define the Middleware Function**: Create a function that takes req, res, and next as parameters.
2. **Implement the Logic**: Write the code for the logic you want to execute.
3. **Call next()**: If you want the request to continue to the next middleware, call next(). If you’re ending the response, you don’t need to call next().
4. **Use the Middleware in Your Express App**: Register the middleware using app.use() or with specific routes.

**3) Explain how middleware is executed in order in an Express.js application.**

=> In an Express.js application, middleware is executed in the order in which it is defined. This sequential execution allows developers to control the flow of requests through the application and perform various tasks at different stages of the request-response lifecycle.

**How Middleware Execution Works**

1. **Order of Definition**: Middleware functions are registered using app.use() or specific route handlers (app.get(), app.post(), etc.). The order in which these middleware functions are defined in the code determines the order of their execution.
2. **Request Lifecycle**: When a request is received by the Express server, it goes through the following steps:
   * The server checks the incoming request against the defined middleware in the order they were registered.
   * Each middleware function is called in sequence until one of the following conditions is met:
     + The middleware calls next(), which passes control to the next middleware function.
     + The middleware ends the request-response cycle by sending a response to the client.
3. **Conditional Execution**: Middleware can be conditionally executed based on the request method or URL. This allows you to have specific middleware functions handle certain routes or request types.
4. **Error Handling**: If an error occurs in any middleware, it can be passed to the next middleware in the stack by calling next(err), where err is the error object. This is typically caught by an error-handling middleware defined later in the stack.

**Problem Statement 3: File System 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 for reading, writing, and managing files. You can perform both asynchronous and synchronous file operations using the fs module.

1) Reading Files:

- We can use fs.readFile() to read a file asynchronously meaning the code doesn’t block execution while waiting for the file to be read.

- If fs.readFileSync() used then it blocks execution until the file is fully read.

2) Writing Files:

- We can use fs.writeFile() to write to a file asynchronously meaning the code doesn’t block execution while waiting for the file to be written.

- If fs.writeFileSync() used then it blocks execution until the file is fully written.

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

=> The main difference between fs.readFile() and fs.readFileSync() in Node.js is **how they handle reading files** with respect to **asynchronous vs. synchronous** execution.

1) fs.readFile() (Asynchronous)

- This is asynchronous means when this is used it does not block execution of the program. When you call it, Node.js continues executing subsequent code while the file is being read in the background. Once the file reading is complete, a callback function is invoked with the result.

2) fs.readFileSync() (Synchronous)

- This is synchronous and blocks the execution of the rest of the program until the file is fully read. The code following the fs.readFileSync() call will not be executed until the file reading operation is complete. The function returns the file content or throws an error, which can be handled with a try-catch block.

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

=> In Node.js, you can check if a file or directory exists using a few different methods provided by the fs (File System) module. The most common ones are fs.existsSync() for synchronous checks and fs.access() for asynchronous checks. fs.existsSync() is a simple and straightforward way to check if a file or directory exists. It blocks execution until it completes, which makes it useful in cases where blocking is acceptable. fs.access() checks the existence of a file or directory asynchronously. This method does not block the event loop, making it a better choice for production applications where non-blocking behavior is preferred.

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

=> In Node.js, asynchronous file operations are a common pattern for handling I/O tasks, ensuring non-blocking, efficient code execution. The fs (File System) module provides several asynchronous methods for file operations like reading, writing, and appending files. These methods rely on callbacks, promises, or async/await patterns to handle the completion of the file operation without blocking the event loop.

1) Using callbacks for Asynchronous File Operations: The typical format for the callback is callback(err, result), where err contains any error encountered and result contains the operation's result (like file data).

2) Using Promises for Asynchronous File Operations: To avoid "callback hell" (i.e., deeply nested callbacks), Node.js introduced fs.promises, which provides a promise-based API for handling file operations.

3) Using async/await for Asynchronous File Operations: The async/await syntax, built on top of promises, makes asynchronous code look more like synchronous code, making it more readable and easier to manage.