**Batch -** T5

**Practical No. -** 9

**Title –** Study and implementation of Node.js

**Student Name -** Sharaneshwar Bharat Punjal

**Student PRN -** 23520011

**Perform following problem statements using Node.Js**

**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.

**1. Install MySQL and mysql2 Package:**

If you haven’t installed the MySQL server, make sure it's running on your system and that you have a database ready with a table from which you want to retrieve data.

Install the mysql2 package by running:

npm install mysql2

**2. Create the Node.js Program:**

Here’s the code that connects to MySQL, retrieves data from a table, and displays it in the console.

js

Copy code

// Import MySQL module

const mysql = require('mysql2');

// Create a connection to the database

const connection = mysql.createConnection({

host: 'localhost', // Change if your MySQL is hosted elsewhere

user: 'root', // Replace with your MySQL username

password: '', // Replace with your MySQL password

database: 'your\_database' // Replace with your database name

});

// Connect to the database

connection.connect((err) => {

if (err) {

console.error('Error connecting to the database:', err.stack);

return;

}

console.log('Connected to MySQL as ID:', connection.threadId);

// Retrieve data from the database

const query = 'SELECT \* FROM your\_table'; // Replace 'your\_table' with your actual table name

connection.query(query, (error, results, fields) => {

if (error) {

console.error('Error executing the query:', error.stack);

return;

}

// Display results

console.log('Results:', results);

// Close the connection

connection.end();

});

});

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

* **What is middleware in Node.js, particularly in the context of Express.js?**

Middleware in Node.js (and particularly in Express.js) is a function that sits between a request and a response. It's essentially a series of functions that are executed when a request is received but before the response is sent. Middleware functions can modify the request or response objects, terminate the request-response cycle, or pass control to the next middleware function in the chain.

In Express.js, middleware functions have access to:

* The request object (req), which contains information about the HTTP request.
* The response object (res), which is used to send a response to the client.
* The next() function, which moves the execution to the next middleware function in the stack.

Middleware functions can perform a variety of tasks such as:

* Logging each request
* Authenticating users
* Parsing request bodies (e.g., JSON or URL-encoded data)
* Serving static files
* Error handling
* **How do you create custom middleware in Express.js?**

Creating custom middleware in Express.js is straightforward. You write a function that takes in three parameters: req (request), res (response), and next (to pass control to the next middleware).

Here’s an example of a custom logging middleware:

const express = require('express');

const app = express();

// Custom middleware function

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

console.log(`${req.method} ${req.url} at ${new Date().toISOString()}`);

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

};

// Use the middleware globally for all routes

app.use(loggerMiddleware);

// Example route

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

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

});

// Start the server

app.listen(3000, () => {

console.log('Server is running on port 3000');

});

In this example, the loggerMiddleware logs the HTTP method and URL of the request, along with the current timestamp, then calls next() to proceed to the next middleware or route handler.

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

In an Express.js application, middleware functions are executed in the order in which they are defined. Express follows a top-to-bottom execution flow, meaning that middleware functions are executed in the order they appear in the code.

Execution Flow:

* Order Matters: Middleware functions are called in the order they are declared using app.use() or app.METHOD() (e.g., app.get(), app.post()).
* next() Function: Each middleware can call next() to pass control to the next middleware in the chain. If next() is not called, the middleware will stop the request-response cycle, and no further middleware will run.
* Final Route Handler: After all middleware is processed, the route handler (which sends the response) is executed. If no route matches or an error occurs, Express moves to the error-handling middleware.

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

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

The fs (File System) module in Node.js provides various methods to interact with the file system, such as reading, writing, deleting, and renaming files. Here are examples of how to read and write files using this module.

Reading Files:

* fs.readFile(): Reads the contents of a file asynchronously.
* fs.readFileSync(): Reads the contents of a file synchronously.

Writing Files:

* fs.writeFile(): Writes data to a file asynchronously.
* fs.writeFileSync(): Writes data to a file synchronously.
* **What is the difference between fs.readFile() and fs.readFileSync()?**

fs.readFile() is asynchronous and non-blocking. It reads the file contents in the background and calls the provided callback function once the operation is completed. You need to handle the result inside the callback function. This is useful for handling multiple operations concurrently without blocking the event loop.

fs.readFileSync() is synchronous and blocking. It reads the file contents before continuing with the next line of code. The entire process is paused until the operation is finished. This can block the execution of other code, so it's recommended only for small scripts or when blocking is acceptable.

Key Differences:

* Blocking vs. Non-blocking: fs.readFileSync() blocks the code execution, while fs.readFile() allows other code to run while the file is being read.
* Use Case: Use fs.readFile() for larger applications that require asynchronous behavior. Use fs.readFileSync() for simpler scripts or when blocking behavior is acceptable.
* **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 the fs.existsSync() method or the newer fs.access() method.

fs.existsSync(): This is a synchronous method that returns true if the file or directory exists, otherwise false.

Example:

const fs = require('fs');

if (fs.existsSync('example.txt')) {

console.log('File exists');

} else {

console.log('File does not exist');

}

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

Node.js encourages asynchronous programming, especially when dealing with I/O operations like reading and writing files. To handle file operations asynchronously, you can use callbacks, promises, or async/await.

Using Callbacks:

The fs module's asynchronous methods like fs.readFile() or fs.writeFile() take a callback as the last argument, which is executed once the file operation is complete.

Example:

const fs = require('fs');

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

if (err) {

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

return;

}

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

});

Using Promises:

The fs.promises API provides promise-based alternatives to the callback-based fs methods, making it easier to chain operations or handle them with async/await.

Example:

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

fs.readFile('example.txt', 'utf8')

.then(data => {

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

})

.catch(err => {

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

});

Using async/await:

You can wrap the promise-based methods inside an async function to handle file operations using await. This gives a more synchronous-looking code structure, while still remaining asynchronous.

Example:

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

async function readFile() {

try {

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

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

} catch (err) {

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

}

}

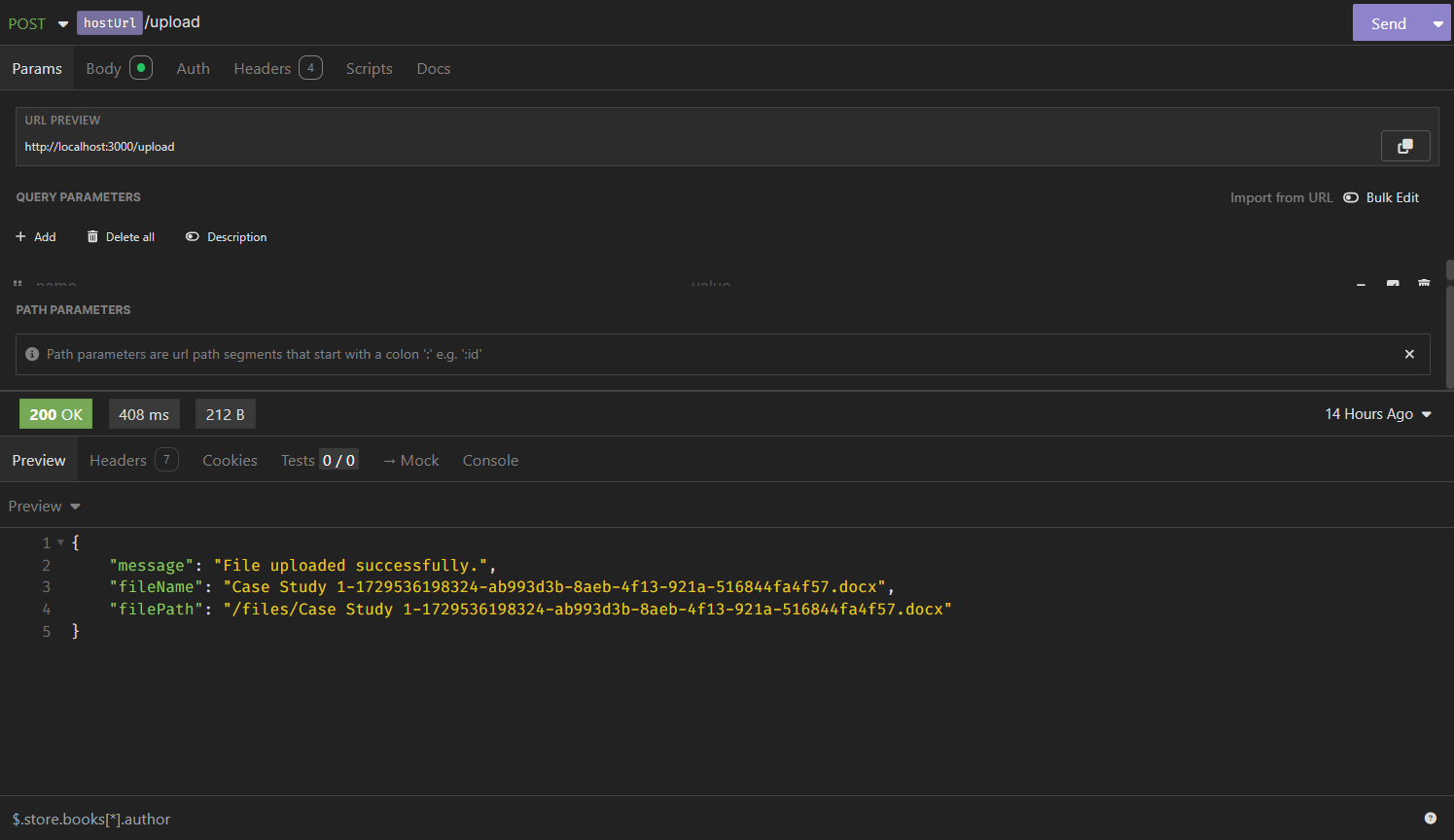
readFile();

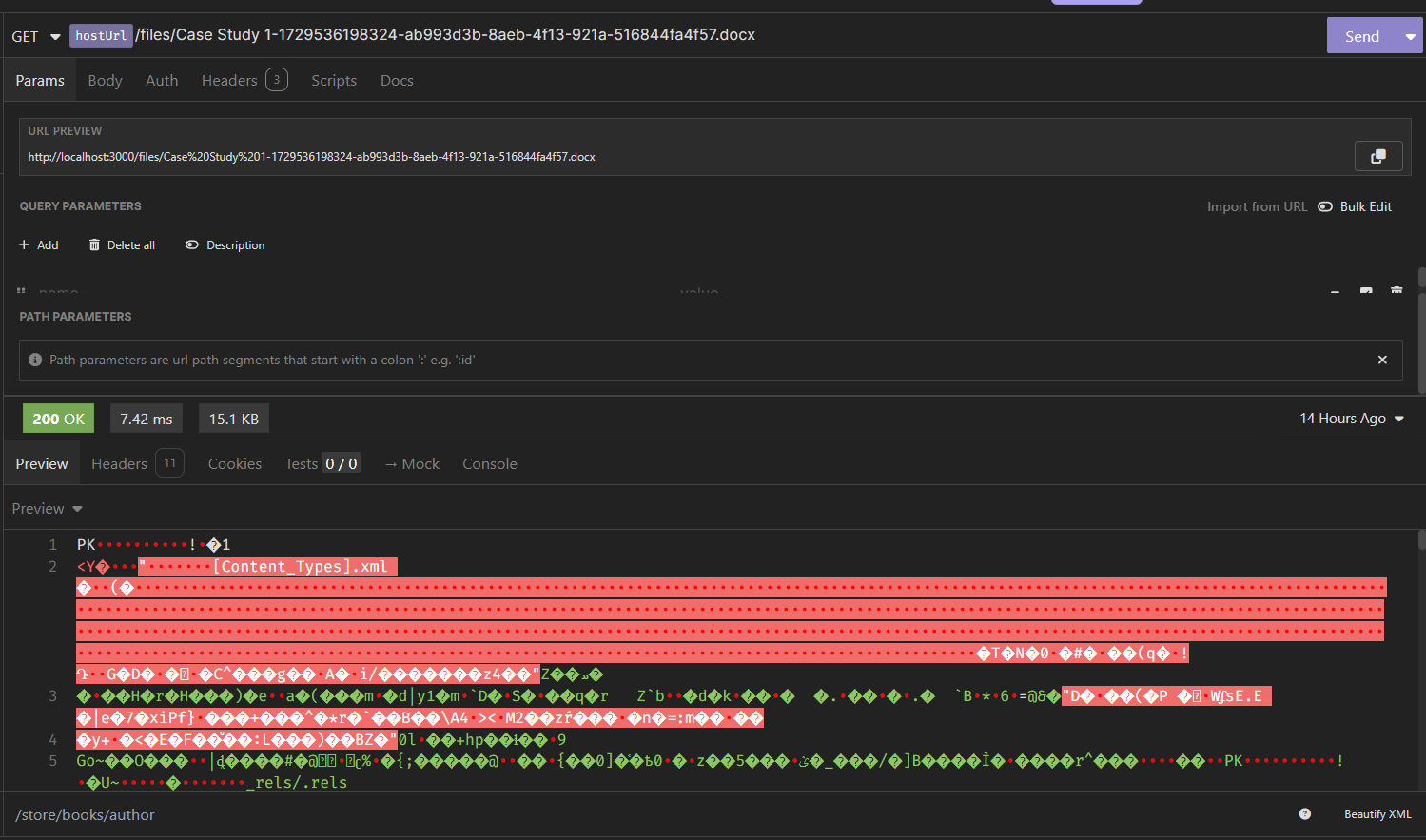
This method is often preferred because it results in cleaner, more readable code, especially when dealing with multiple asynchronous operations in sequence.

**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.

**Screenshots –**

****

****

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

* Create a real-time chat application using Node.js, Express, and Socket.io that allows multiple users to join and communicate in a chat room.
* Set up a Node.js server with Socket.io for real-time bi-directional communication.
* Implement event listeners to handle user connections, disconnections, and message broadcasting to all connected users.

**Screenshots –**

