**Batch- T4**

**Practical No. 8**

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

**Student Name: Daivik Karbhari**

**Student PRN: 22510111**

**Problem Statement:**

1. **Introduction to Node.js**

* **What is Node.js, and how does it differ from traditional server-side platforms like Apache or PHP?**

**Node.js** is a runtime environment that allows you to execute JavaScript code on the server side. Unlike traditional server-side platforms like Apache or PHP, which are designed for handling synchronous and blocking operations, Node.js excels at handling asynchronous, non-blocking operations with a single-threaded event loop. This makes it ideal for building scalable applications that need to handle a large number of I/O operations, such as web servers or real-time applications.

* **Node.js** uses JavaScript as its primary language.
* It uses a non-blocking, event-driven architecture, which is different from PHP's traditional blocking execution model.
* **Apache** and PHP are generally multithreaded, whereas Node.js handles multiple requests on a single thread using the event loop.
* **What is the purpose of the V8 engine in Node.js?**

The **V8 engine** is a JavaScript engine developed by Google for Chrome, and it is used in **Node.js** to compile and execute JavaScript code. V8 compiles JavaScript into native machine code, which improves the performance of JavaScript execution. In Node.js, V8 ensures that JavaScript code is executed quickly and efficiently, allowing for fast server-side operations.

* **Explain the single-threaded, event-driven architecture of Node.js.**

Node.js operates on a single-threaded, event-driven architecture. This means:

* Node.js processes all requests on a single thread.
* Instead of waiting for an operation to complete (like reading a file or querying a database), Node.js uses an event loop to handle multiple requests asynchronously. Once an operation is complete, a callback function is triggered.

This architecture is ideal for I/O-heavy applications as it can handle many requests simultaneously without the need for multiple threads.

* **Why is Node.js considered non-blocking?**

Node.js is considered **non-blocking** because it uses asynchronous I/O operations. This means that when Node.js performs tasks such as reading from a file or querying a database, it doesn't wait for the task to complete before moving on to handle other operations. The event loop handles callbacks once the operation is done. This leads to better performance in applications that have a lot of I/O operations, as Node.js can process other requests while waiting for tasks to complete.

* **What is npm, and how is it used in Node.js?**

**npm** (Node Package Manager) is the default package manager for Node.js. It allows developers to:

* Download and manage libraries or frameworks (known as packages) from a centralized repository.
* Manage project dependencies and versions.
* Share code with the community.

In a Node.js project, npm can be used to install, update, and uninstall packages, as well as run scripts like build, test, etc.

* **What is a module in Node.js? How do you export and import modules?**

A **module** in Node.js is a reusable piece of code that is encapsulated in a file. Modules help in organizing and reusing code across different parts of a Node.js application.

To **export** a module, you use the module.exports or exports object:

// myModule.js

function greet(name) {

return `Hello, ${name}`;

}

module.exports = greet;

To import a module, you use the require() function:

// app.js

const greet = require('./myModule');

console.log(greet('John'));

* **What is the difference between require() and import in Node.js?**

\*\*require()\*\*: Used in CommonJS modules, which is the older module system in Node.js. It loads modules synchronously.

Example:

**const module = require('./module.js');**

\*\*import\*\*: Part of ECMAScript Modules (ESM) and allows for asynchronous loading of modules. It is now supported in modern Node.js versions (with a .mjs file extension or using the "type": "module" in package.json).

Example:

**import module from './module.js';**

* **How can you create a custom module in Node.js?**

You can create a custom module by writing JavaScript functions or objects and exporting them using module.exports. Here’s an example:

**// calculator.js**

**function add(a, b) {**

**return a + b;**

**}**

**function subtract(a, b) {**

**return a - b;**

**}**

**module.exports = { add, subtract };**

Now you can import and use this module:

**// app.js**

**const calculator = require('./calculator');**

**console.log(calculator.add(5, 3)); // Output: 8**

**console.log(calculator.subtract(5, 3)); // Output: 2**

* **What is the role of the package.json file in a Node.js project?**

The \*\*package.json\*\* file holds metadata for your project. It includes:

* Project details (name, version, description).
* Dependencies required to run the project.
* Scripts for build, test, or start processes.
* Configurations for npm and project settings.

Example of a package.json:

**{**

**"name": "my-project",**

**"version": "1.0.0",**

**"description": "A sample Node.js project",**

**"scripts": {**

**"start": "node app.js",**

**"test": "mocha"**

**},**

**"dependencies": {**

**"express": "^4.17.1"**

**}**

**}**

* **How do you install a package globally and locally using npm?**

To install a package locally (inside the current project):

npm install <package-name>

To install a package globally (available across all projects):

npm install -g <package-name>

* **What is the difference between asynchronous and synchronous programming in Node.js?**

**Synchronous programming**: Tasks are executed one after another. Each task waits for the previous one to finish before starting.

Example:

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

**const data = fs.readFileSync('file.txt', 'utf8'); // Blocking**

**console.log(data);**

**Asynchronous programming**: Tasks are executed concurrently, and callbacks are used when a task is complete. It doesn't block other tasks.

Example:

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

**if (err) throw err;**

**console.log(data); // Non-blocking**

**});**

* **How do you create an HTTP server in Node.js?**

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

*const* server = http.createServer((*req*, *res*) *=>* {

*res*.statusCode = 200;

*res*.setHeader('Content-Type', 'text/plain');

*res*.end('Hello, World\n');

});

server.listen(3000, () *=>* {

  console.log('Server running at http://3000/')

})

* **What is the difference between http.createServer() and using frameworks like Express.js?**

**http.createServer()**: You create and manage the server manually, including handling routes and HTTP methods. It's lower-level and requires more setup.

**Express.js**: A web framework built on top of Node.js that simplifies the process of setting up routes, middleware, and HTTP methods. It provides a cleaner, higher-level abstraction for building web applications.

* **How do you handle GET and POST requests in Node.js?**

You can handle GET and POST requests using the http module or a framework like **Express.js**.

**Using http.createServer():**

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

*const* url = require('url');

http.createServer((*req*, *res*) *=>* {

*const* method = *req*.method;

  if (method === 'GET') {

    // Handle GET request

*const* queryObject = url.parse(*req*.url, true).query;

*res*.end('Received GET request with query: ' + JSON.stringify(queryObject));

  } else if (method === 'POST') {

    // Handle POST request

*let* body = '';

*req*.on('data', *chunk* *=>* {

      body += *chunk*.toString();

    });

*req*.on('end', () *=>* {

*res*.end('Received POST request with data: ' + body);

    });

  }

}).listen(3000);

Using Express:

*const* express = require('express');

*const* app = express();

app.use(express.json()); // For parsing application/json

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

*res*.send('Received GET request');

});

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

*res*.send('Received POST request with data: ' + JSON.stringify(*req*.body));

});

app.listen(3000, () *=>* {

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

});

**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: Database Connectivity**

**1. How do you connect to a MongoDB database from a Node.js application?**

To connect to a **MongoDB** database from a **Node.js** application, you can use the official **MongoDB Node.js driver** or an Object Data Modeling (ODM) library like **Mongoose**.

Here’s how to connect using both:

**Using MongoDB Node.js driver:**

1. Install the MongoDB driver via npm:

bash

Copy code

npm install mongodb

1. Connect to the database:

js

Copy code

const { MongoClient } = require('mongodb');

// Connection URL

const url = 'mongodb://localhost:27017';

const client = new MongoClient(url);

// Database Name

const dbName = 'myDatabase';

async function connectToDatabase() {

try {

// Connect to the MongoDB server

await client.connect();

console.log('Connected successfully to MongoDB');

// Select the database

const db = client.db(dbName);

// Perform database operations here

} catch (err) {

console.error('Failed to connect to MongoDB', err);

} finally {

await client.close(); // Close the connection when done

}

}

connectToDatabase();

**Using Mongoose (ODM library for MongoDB):**

1. Install **Mongoose**:

bash

Copy code

npm install mongoose

1. Connect using Mongoose:

js

Copy code

const mongoose = require('mongoose');

const url = 'mongodb://localhost:27017/myDatabase';

async function connectToDatabase() {

try {

// Connect to the MongoDB database using Mongoose

await mongoose.connect(url, {

useNewUrlParser: true,

useUnifiedTopology: true,

});

console.log('Connected to MongoDB using Mongoose');

} catch (err) {

console.error('Failed to connect to MongoDB', err);

}

}

connectToDatabase();

**2. What is the purpose of the Mongoose library in Node.js?**

**Mongoose** is an **Object Data Modeling (ODM)** library for MongoDB and Node.js. It provides a higher-level abstraction for working with MongoDB databases by defining schemas, models, and validation rules for your data. Mongoose simplifies interacting with MongoDB by allowing you to define data models using schemas, which provide structure and validation to your documents.

Some key purposes of Mongoose include:

* Defining schemas and models for documents.
* Simplifying CRUD (Create, Read, Update, Delete) operations.
* Automatic data validation.
* Query building and relationships between documents.
* Middleware for pre/post hooks in document operations.

Example of defining a schema and model using Mongoose:

js

Copy code

const mongoose = require('mongoose');

// Define a schema for the User model

const userSchema = new mongoose.Schema({

name: String,

email: String,

age: Number,

});

// Create a model for the User collection

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

// Now you can perform CRUD operations on the User model

**3. Explain how you would perform basic CRUD operations (Create, Read, Update, Delete) using MongoDB and Node.js.**

**Using MongoDB Node.js Driver:**

1. **Create (Insert a document):**

js

Copy code

const { MongoClient } = require('mongodb');

const client = new MongoClient('mongodb://localhost:27017');

async function createDocument() {

try {

await client.connect();

const db = client.db('myDatabase');

const collection = db.collection('users');

const newUser = { name: 'John', email: 'john@example.com', age: 25 };

const result = await collection.insertOne(newUser);

console.log('Document inserted:', result.insertedId);

} finally {

await client.close();

}

}

createDocument();

1. **Read (Find documents):**

js

Copy code

async function readDocuments() {

try {

await client.connect();

const db = client.db('myDatabase');

const collection = db.collection('users');

const users = await collection.find({}).toArray();

console.log('Documents found:', users);

} finally {

await client.close();

}

}

readDocuments();

1. **Update (Update a document):**

js

Copy code

async function updateDocument() {

try {

await client.connect();

const db = client.db('myDatabase');

const collection = db.collection('users');

const result = await collection.updateOne(

{ name: 'John' },

{ $set: { age: 26 } }

);

console.log('Document updated:', result.modifiedCount);

} finally {

await client.close();

}

}

updateDocument();

1. **Delete (Remove a document):**

js

Copy code

async function deleteDocument() {

try {

await client.connect();

const db = client.db('myDatabase');

const collection = db.collection('users');

const result = await collection.deleteOne({ name: 'John' });

console.log('Document deleted:', result.deletedCount);

} finally {

await client.close();

}

}

deleteDocument();

**Using Mongoose for CRUD operations:**

1. **Create (Insert a document):**

js

Copy code

const mongoose = require('mongoose');

// Define schema and model

const userSchema = new mongoose.Schema({ name: String, email: String, age: Number });

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

async function createUser() {

const user = new User({ name: 'John', email: 'john@example.com', age: 25 });

const result = await user.save();

console.log('User created:', result);

}

mongoose.connect('mongodb://localhost:27017/myDatabase').then(createUser);

1. **Read (Find documents):**

js

Copy code

async function getUsers() {

const users = await User.find();

console.log('Users:', users);

}

mongoose.connect('mongodb://localhost:27017/myDatabase').then(getUsers);

1. **Update (Update a document):**

js

Copy code

async function updateUser() {

const result = await User.updateOne({ name: 'John' }, { age: 26 });

console.log('User updated:', result);

}

mongoose.connect('mongodb://localhost:27017/myDatabase').then(updateUser);

1. **Delete (Remove a document):**

js

Copy code

async function deleteUser() {

const result = await User.deleteOne({ name: 'John' });

console.log('User deleted:', result);

}

mongoose.connect('mongodb://localhost:27017/myDatabase').then(deleteUser);

**Summary of basic CRUD operations:**

* **Create**: Use insertOne() in MongoDB driver or save() in Mongoose.
* **Read**: Use find() or findOne() to query documents.
* **Update**: Use updateOne() or findByIdAndUpdate() to update existing documents.
* **Delete**: Use deleteOne() or findByIdAndRemove() to remove documents.

By using Mongoose, you gain the benefit of a structured schema and built-in validation, whereas the native MongoDB driver gives you more direct control over the database but requires more manual setup.

**Problem Statement 5: Building a RESTful API**







