**Batch T7**

**Practical No. 8**

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

**Student Name: Vaishnavi Jadhav**

**Student PRN: 22510119**

**Problem Statement 1: Introduction to Node.js**

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

Ans. Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. It allows developers to run JavaScript on the server-side. The main differences from traditional platforms are:

* Node.js uses an event-driven, non-blocking I/O model, making it lightweight and efficient.
* It uses JavaScript for both front-end and back-end development, allowing for full-stack JavaScript applications.
* Node.js is designed for building scalable network applications, while Apache is primarily a web server and PHP is a scripting language.

**2.** What is the purpose of the V8 engine in Node.js?

The V8 engine, developed by Google for Chrome, is used in Node.js to:

* Parse and execute JavaScript code
* Handle memory allocation
* Garbage collect unused memory
* Optimize code execution

It compiles JavaScript directly to native machine code before executing it, which significantly improves performance.

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

Node.js uses a single-threaded event loop model:

* It runs in a single thread, but uses non-blocking I/O calls to support thousands of concurrent connections.
* The event loop handles all asynchronous callbacks.
* When an operation completes, a callback is pushed to the event queue.
* The event loop checks the queue and executes callbacks when the call stack is empty.

This architecture allows Node.js to handle many concurrent operations without the overhead of thread context switching.

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

Node.js is non-blocking because:

* It uses asynchronous I/O operations by default.
* Instead of waiting for an operation to complete, it moves to the next task.
* When the previous operation finishes, a callback function is called to handle the result.
* This approach allows the server to handle multiple requests simultaneously without getting blocked on any single operation.

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

npm (Node Package Manager) is:

* The default package manager for Node.js
* A large repository of JavaScript packages
* A command-line tool to install, share, and manage dependencies in Node.js projects

It's used to:

* Install third-party packages
* Manage project dependencies
* Run scripts defined in package.json
* Publish and distribute Node.js packages

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

A module in Node.js is a reusable block of code that encapsulates related functionality. To export a module:

**// myModule.js**

**module.exports = {**

**someFunction: function() {**

**// function code**

**},**

**someVariable: 'value'**

**};**

To import:

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

**myModule.someFunction();**

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

require() is the CommonJS way of importing modules, while import is part of the ECMAScript (ES6) module system.

* require() is synchronous and can be called anywhere in the file.
* import is asynchronous and must be used at the top of the file.
* import offers more flexibility with named imports and default exports.

Node.js supports both, but import requires either using .mjs file extension or setting "type": "module" in package.json.

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

To create a custom module:

1. Create a new file (e.g., myModule.js)
2. Write your code and export what you want to make available:

**// myModule.js**

**function myFunction() {**

**console.log('Hello from myModule');**

**}**

**module.exports = { myFunction };**

To Import

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

**myModule.myFunction();**

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

The package.json file:

* Describes the project and its dependencies
* Contains metadata about the project (name, version, author, etc.)
* Lists project dependencies and their versions
* Defines scripts for running, testing, and managing the project
* Can specify the project's entry point, license, and other configurations

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

**To install a package locally (in the current project):**

**npm install package-name**

**To install a package globally (available system-wide):**

**npm install -g package-name**

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

Synchronous programming:

* Executes code sequentially, blocking until each operation completes
* Can lead to performance issues with time-consuming tasks

Asynchronous programming:

* Allows code to continue executing while waiting for I/O operations
* Uses callbacks, promises, or async/await to handle results
* Improves performance by not blocking the event loop

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

Here is a basic server:

const http = require('http');

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

res.statusCode = 200;

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

res.end('Hello World');

});

server.listen(3000, 'localhost', () => {

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

});

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

http.createServer():

* Part of Node.js core, no additional installation needed
* Provides basic HTTP server functionality
* Requires more manual handling of routes, requests, and responses

Express.js:

* A web application framework built on top of Node.js
* Provides additional features like middleware, easier routing, and template engines
* Simplifies common web development tasks and improves code organization

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

Using the built-in http module:

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

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

**if (req.method === 'GET') {**

**// Handle GET request**

**if (req.url === '/') {**

**res.writeHead(200, { 'Content-Type': 'text/plain' });**

**res.end('Hello World!');**

**}**

**} else if (req.method === 'POST') {**

**// Handle POST request**

**let body = '';**

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

**body += chunk.toString();**

**});**

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

**console.log(body);**

**res.writeHead(200, { 'Content-Type': 'text/plain' });**

**res.end('Data received');**

**});**

**}**

**});**

**server.listen(3000);**

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

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

**Middleware in Express.js 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, commonly denoted by a variable named next.**

Key points about middleware:

* It can execute any code
* It can make changes to the request and response objects
* It can end the request-response cycle
* It can call the next middleware function in the stack

Middleware functions can perform the following tasks:

* Execute any code
* Make changes to the request and response objects
* End the request-response cycle
* Call the next middleware function in the stack

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

**Creating custom middleware in Express.js is straightforward. Here's a basic structure:**

**function myMiddleware(req, res, next) {**

**// Middleware logic here**

**console.log('This middleware was called!');**

**// Call next() to pass control to the next middleware function**

**next();**

**}**

**You can then use this middleware in your Express application like this:**

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

**const app = express();**

**// Use the middleware for all routes**

**app.use(myMiddleware);**

**// Or use it for a specific route**

**app.get('/some-route', myMiddleware, (req, res) => {**

**res.send('Hello from some-route');**

**});**

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

**Middleware functions are executed sequentially in the order they are defined. This order is crucial to understand:**

1. Middleware defined using app.use() or app.METHOD() are called in the order they are defined in the code.
2. When a request is received, Express starts at the first middleware function and works its way down the stack.
3. Each middleware function can either end the request-response cycle by sending a response to the client, or it can pass control to the next middleware function using next().
4. If next() is called with an argument (typically an error object), Express will skip any remaining "regular" middleware and pass control to error-handling middleware.

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

**const app = express();**

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

**console.log('This always runs first');**

**next();**

**});**

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

**console.log('This runs for GET requests to /example');**

**next();**

**}, (req, res, next) => {**

**console.log('This also runs for GET requests to /example');**

**next();**

**});**

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

**console.log('This runs after all other middleware if no response has been sent');**

**res.send('Hello from Express!');**

**});**

**app.listen(3000);**

**In this example:**

1. The first middleware always runs for every request.
2. For GET requests to '/example', the second and third middleware functions run in order.
3. The last middleware runs only if no response has been sent by previous middleware.

It's important to note that once res.send(), res.json(), or any other method that sends a response is called, subsequent middleware in the stack won't be executed for that request-response cycle.

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

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

**The fs module provides both synchronous and asynchronous methods for file operations. Here are examples of reading and writing files:**

Reading a file:

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

**// Asynchronous read**

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

**if (err) {**

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

**return;**

**}**

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

**});**

**// Synchronous read**

**try {**

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

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

**} catch (err) {**

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

**}**

Writing to a file:

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

**const content = 'Hello, Node.js!';**

**// Asynchronous write**

**fs.writeFile('output.txt', content, (err) => {**

**if (err) {**

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

**return;**

**}**

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

**});**

**// Synchronous write**

**try {**

**fs.writeFileSync('output.txt', content);**

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

**} catch (err) {**

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

**}**

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

**The main differences are:**

* fs.readFile() is asynchronous:
  + It doesn't block the execution of the program.
  + It takes a callback function that is called when the read operation is complete.
  + It's generally preferred for better performance in I/O-heavy applications.
* fs.readFileSync() is synchronous:
  + It blocks the execution of the program until the file is read.
  + It returns the file contents directly.
  + It's simpler to use but can cause performance issues if reading large files or in high-concurrency scenarios.

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

**You can use fs.existsSync()(synchronous) or the asynchronous fs.access() (asynchronous) method:**

const fs = require('fs');

// Synchronous check

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

console.log('The file exists');

} else {

console.log('The file does not exist');

}

// Asynchronous check

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

if (err) {

console.log('The file does not exist');

} else {

console.log('The file exists');

}

});

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

**There are three main ways to handle asynchronous file operations in Node.js:**

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

**if (err) {**

**console.error('Error:', err);**

**return;**

**}**

**console.log('Data:', data);**

**});**

**Promises (using fs.promises):**

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

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

**.then(data => console.log('Data:', data))**

**.catch(err => console.error('Error:', err));**

**Async/Await (using fs.promises):**

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

**async function readFileAsync() {**

**try {**

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

**console.log('Data:', data);**

**} catch (err) {**

**console.error('Error:', err);**

**}**

**}**

**readFileAsync();**

**Problem Statement 4: Database Connectivity**

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

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

You can connect to MongoDB using either the native MongoDB driver or an ODM (Object Data Modeling) library like Mongoose. Here's how to do it with both:

a. Using the native MongoDB driver:

First, install the MongoDB driver:

**npm install mongodb**

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

const uri = "mongodb://localhost:27017/mydatabase";

async function connectToMongo() {

try {

const client = new MongoClient(uri);

await client.connect();

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

return client.db("myDatabase");

} catch (error) {

console.error("Could not connect to MongoDB", error);

process.exit(1);

}

}

// Usage

connectToMongo().then(db => {

// Use the database object for operations

});

To connect with moongosh:

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

**const uri = "mongodb://localhost:27017/myDatabase";**

**mongoose.connect(uri, { useNewUrlParser: true, useUnifiedTopology: true })**

**.then(() => console.log("Connected successfully to MongoDB"))**

**.catch(error => {**

**console.error("Could not connect to MongoDB", error);**

**process.exit(1);**

**});**

**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. Its main purposes are:**

* Provide a straight-forward, schema-based solution to model your application data
* Include built-in type casting, validation, query building, business logic hooks and more
* Offer a higher level of abstraction over the MongoDB driver
* Manage relationships between data
* Provide schema validation
* Translate between objects in code and the representation of those objects in MongoDB

Mongoose allows you to define schemas for your data, which can include validation rules, default values, and other metadata. It also provides a more intuitive API for interacting with your MongoDB database.

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

Let's use Mongoose for these examples as it provides a more intuitive API. First, we'll define a simple schema and model:

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

**const userSchema = new mongoose.Schema({**

**name: String,**

**email: { type: String, unique: true },**

**age: Number**

**});**

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

**Now, let's perform CRUD operations:**

a. Create (C):

**async function createUser(userData) {**

**try {**

**const newUser = new User(userData);**

**const savedUser = await newUser.save();**

**console.log('User created:', savedUser);**

**return savedUser;**

**} catch (error) {**

**console.error('Error creating user:', error);**

**}**

**}**

**// Usage**

**createUser({ name: 'John Doe', email: 'john@example.com', age: 30 });**

**b. Read (R):**

**async function findUsers(query) {**

**try {**

**const users = await User.find(query);**

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

**return users;**

**} catch (error) {**

**console.error('Error finding users:', error);**

**}**

**}**

**// Usage**

**findUsers({ age: { $gte: 18 } }); // Find all users 18 or older**

**c. Update (U):**

**async function updateUser(id, updateData) {**

**try {**

**const updatedUser = await User.findByIdAndUpdate(id, updateData, { new: true });**

**console.log('User updated:', updatedUser);**

**return updatedUser;**

**} catch (error) {**

**console.error('Error updating user:', error);**

**}**

**}**

**// Usage**

**updateUser('user\_id\_here', { age: 31 });**

**d. Delete (D):**

**async function deleteUser(id) {**

**try {**

**const deletedUser = await User.findByIdAndDelete(id);**

**console.log('User deleted:', deletedUser);**

**return deletedUser;**

**} catch (error) {**

**console.error('Error deleting user:', error);**

**}**

**}**

**// Usage**

**deleteUser('user\_id\_here');**

1. Error Handling
2. Validation: Mongoose schemas can include validation rules. For example:

**const userSchema = new mongoose.Schema({**

**name: { type: String, required: true },**

**email: { type: String, required: true, unique: true },**

**age: { type: Number, min: 18 }**

**});**