### **Introduction to Node.js**

#### **1. What is Node.js?**

Node.js is a **JavaScript runtime environment** built on Google Chrome's V8 JavaScript engine. It allows you to run JavaScript code outside a web browser. Traditionally, JavaScript was used only for frontend web development in browsers. Node.js extends its capabilities to the **backend**, enabling server-side development.

#### **2. Why is Node.js used?**

Node.js is primarily used to build **scalable and efficient applications**. Its non-blocking, event-driven architecture makes it suitable for applications that handle multiple requests simultaneously, such as:

* Real-time chat applications
* Streaming services (e.g., Netflix, Spotify)
* APIs and microservices
* Data-intensive applications

#### **3. Advantages of Node.js**

* **Fast Execution**: Built on the V8 engine, Node.js executes JavaScript code quickly.
* **Non-Blocking I/O**: It uses an asynchronous, event-driven model, meaning operations (like reading files or databases) don't block the execution of other code.
* **Single Language for Full Stack**: Developers can use JavaScript for both frontend and backend.
* **Large Ecosystem**: The Node Package Manager (npm) provides thousands of reusable packages/modules.
* **Scalability**: Handles many concurrent connections efficiently.

#### **4. Why use Node.js over other technologies?**

* **Compared to PHP**: Node.js is non-blocking, while PHP is synchronous (blocking) by default. Node.js performs better for real-time applications.
* **Compared to Python**: Python is excellent for data processing but less suitable for handling high-concurrency tasks compared to Node.js.
* **Compared to Java**: Java is powerful but requires more resources and time to set up compared to Node.js.

### **Examples and Execution**

#### **Example 1: A Simple Node.js Server**

**Description**: This example creates a basic web server that responds with "Hello, World!" when accessed.

**Code**:

javascript

Copy code

// Import the built-in HTTP module

const http = require('http');

// Create a server

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

res.statusCode = 200; // HTTP status: OK

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

res.end('Hello, World!\n'); // Send response

});

// Define the port

const PORT = 3000;

// Start the server

server.listen(PORT, () => {

console.log(`Server is running on http://localhost:${PORT}`);

});

**Steps to Execute**:

1. Save the code in a file named server.js.
2. Open a terminal and navigate to the directory containing server.js.

Run the command:  
bash  
Copy code  
node server.js

1. Open your web browser and go to http://localhost:3000 to see "Hello, World!".

#### **Example 2: Reading a File Asynchronously**

**Description**: This example reads a file and displays its contents.

**Code**:

javascript

Copy code

// Import the built-in File System module

const fs = require('fs');

// Define the file to read

const filePath = './sample.txt';

// Read the file asynchronously

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

if (err) {

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

return;

}

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

});

**Steps to Execute**:

Create a text file named sample.txt in the same directory with some content, like:  
vbnet  
Copy code  
This is a sample text file.

1. Save the code in a file named readFile.js.
2. Open a terminal and navigate to the directory containing readFile.js.

Run the command:  
bash  
Copy code  
node readFile.js

1. The terminal will display the contents of sample.txt.

---------------------------------------------------------------------------------------------------------------------------

### **1. What is a JavaScript Runtime Environment?**

A **JavaScript runtime environment** is a platform that provides the necessary tools, libraries, and features to execute JavaScript code. It allows developers to write and run JavaScript outside of the browser.

* **In Browsers**: The runtime includes a JavaScript engine (e.g., V8 in Chrome) and a web API to handle tasks like DOM manipulation, timers, and events.
* **In Node.js**: The runtime includes the V8 engine, along with libraries and APIs (like fs, http, etc.) to interact with files, networks, and other resources.

**Why is Node.js a runtime environment?**Node.js extends JavaScript’s functionality by adding APIs for server-side operations like file reading, network requests, and database interactions. This makes JavaScript suitable for backend development.

### **2. What is Asynchronous?**

**Asynchronous** means tasks are executed in a non-blocking way, allowing the program to continue running other operations while waiting for a task (e.g., file reading or network request) to complete.

In an **asynchronous model**:

* Tasks do not wait for previous tasks to finish.
* When a task is completed, a callback function is triggered to handle the result.

**Example**:  
You order food at a restaurant (task). Instead of waiting idly (blocking), you chat with friends (continue other operations). When the food is ready, the waiter (callback) informs you.

**Asynchronous in Node.js**:  
Node.js uses this model extensively to handle tasks like:

* File system operations
* Database queries
* HTTP requests

### **3. What are Concurrent Connections?**

**Concurrent connections** refer to multiple clients (users or devices) interacting with a server simultaneously.

* **Example**: A web server serving 1,000 users at the same time is handling 1,000 concurrent connections.
* **In Node.js**: The event-driven model efficiently manages thousands of connections using a single thread. It listens for events and processes them without blocking other connections.

**Why is this important?**Traditional servers like PHP handle each connection with a separate thread, which consumes more memory. Node.js, however, handles many connections in the same thread, reducing resource consumption.

### **4. How is the 2nd Example Asynchronous?**

#### **Code Recap:**

javascript

Copy code

const fs = require('fs');

const filePath = './sample.txt';

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

if (err) {

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

return;

}

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

});

#### **Explanation:**

The **fs.readFile** function is **asynchronous** because:

1. When called, it does not block the execution of the rest of the code.
2. Instead of waiting for the file to be read, the program continues running other tasks.
3. Once the file is fully read, the **callback function** is executed to handle the result (either display the file contents or handle an error).

#### **Comparison: Synchronous vs. Asynchronous**

**Synchronous Code**:

javascript

Copy code

const data = fs.readFileSync('./sample.txt', 'utf8'); // Blocks here

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

console.log('This runs after the file is read');

* The program **waits** until readFileSync completes.
* Other tasks cannot run during this time.

**Asynchronous Code**:

javascript

Copy code

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

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

});

console.log('This runs immediately after calling readFile'); // Runs immediately

* The program **does not wait** for fs.readFile to finish.
* The second console.log runs immediately.

**Key Advantage**:  
In real-world applications, asynchronous operations ensure that servers don't get stuck waiting for one slow task, allowing them to handle multiple requests efficiently.

----------------------------------------------------------------------------------------------------------------------------

### **HTTP Module in Node.js**

#### **1. What is the HTTP Module?**

The **HTTP module** in Node.js is a built-in module that allows you to create and manage HTTP servers and clients. It provides tools to handle requests (from clients) and send responses (from the server).

#### **2. Why is the HTTP Module Used?**

The HTTP module is used to:

* **Build web servers** that serve content to users.
* Handle HTTP methods like GET, POST, PUT, DELETE, etc.
* Manage client-server communication by reading requests and sending appropriate responses.

#### **3. Advantages of the HTTP Module**

* **Built-in Support**: No need to install external libraries.
* **Lightweight**: Provides just the tools needed for HTTP communication.
* **Customizability**: Allows full control over how requests and responses are handled.

#### **4. Why Use the HTTP Module Over Others?**

While frameworks like **Express.js** simplify server creation, the HTTP module gives developers complete control, making it ideal for learning and building lightweight applications.

### **Examples and Execution**

#### **Example 1: Basic HTTP Server**

**Description**: This code creates a simple HTTP server that responds with "Hello, World!" when accessed.

**Code**:

javascript

Copy code

// Import the HTTP module

const http = require('http');

// Create the server

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

// Set response headers

res.statusCode = 200; // HTTP Status: OK

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

// Send the response

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

});

// Define the port to listen on

const PORT = 3000;

// Start the server

server.listen(PORT, () => {

console.log(`Server running at http://localhost:${PORT}/`);

});

**Steps to Execute**:

1. Save the code in a file named http\_server.js.
2. Open a terminal and navigate to the file's directory.

Run the command:  
bash  
Copy code  
node http\_server.js

Open a web browser or use a tool like curl to access http://localhost:3000.  
bash  
Copy code  
curl http://localhost:3000

#### **Example 2: Responding Based on URL Path**

**Description**: This server responds differently based on the URL path in the request.

**Code**:

javascript

Copy code

// Import the HTTP module

const http = require('http');

// Create the server

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

// Set response header

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

// Check the request URL

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

res.statusCode = 200;

res.end('Welcome to the homepage!\n');

} else if (req.url === '/about') {

res.statusCode = 200;

res.end('This is the About page.\n');

} else {

res.statusCode = 404;

res.end('Page not found.\n');

}

});

// Define the port to listen on

const PORT = 3000;

// Start the server

server.listen(PORT, () => {

console.log(`Server running at http://localhost:${PORT}/`);

});

**Steps to Execute**:

1. Save the code in a file named http\_routing.js.
2. Open a terminal and navigate to the file's directory.

Run the command:  
bash  
Copy code  
node http\_routing.js

1. Test the server:
   * Open a browser or use curl:

For the homepage:  
bash  
Copy code  
curl http://localhost:3000/

For the About page:  
bash  
Copy code  
curl http://localhost:3000/about

For a non-existing page:  
bash  
Copy code  
curl http://localhost:3000/unknown

#### **5. Explanation of How HTTP Module Works**

* **Request (req)**: Contains details about the incoming HTTP request (e.g., URL, method, headers).
* **Response (res)**: Used to send data (e.g., status, headers, body) back to the client.
* **Methods**:
  + req.url: The requested URL path.
  + res.statusCode: Sets the HTTP status code (e.g., 200 for success, 404 for not found).
  + res.setHeader: Sets HTTP headers like Content-Type.
  + res.end: Ends the response and sends data to the client.

#### **Real-World Usage of HTTP Module**

* **APIs**: Create REST APIs to interact with databases and return data.
* **Custom Servers**: Build servers for specific tasks, such as streaming or chat.

—-------------------------------------------------------------------------------------------------------------------------

### **npm (Node Package Manager)**

#### **1. What is npm?**

**npm** (Node Package Manager) is a tool that helps manage **JavaScript packages (libraries or modules)**. It is installed automatically with Node.js and is used to:

* Install and manage external libraries or frameworks.
* Share your own JavaScript code as reusable packages.
* Manage project dependencies efficiently.

#### **2. Why is npm used?**

npm simplifies the process of adding external code to your project. Instead of writing everything from scratch, developers can use pre-built libraries for tasks like handling databases, creating APIs, or working with files.

**Key Use Cases:**

* **Install Packages**: Add functionality to your project using thousands of ready-made packages.
* **Version Control**: Manage package versions to ensure compatibility.
* **Custom Scripts**: Automate tasks like testing, building, and deployment.

#### **3. Advantages of npm**

* **Huge Ecosystem**: Provides access to over 1.5 million packages.
* **Easy Installation**: Quickly install, update, or remove dependencies.
* **Versioning**: Ensures compatibility between different library versions.
* **Open Source**: Free to use and widely supported by the developer community.

#### **4. Why Use npm Over Alternatives?**

* **Compared to Yarn**: npm is simpler and comes pre-installed with Node.js. Yarn is faster in some cases, but npm has improved significantly in recent versions.
* **Compared to Manual Downloads**: npm automatically resolves dependencies and avoids conflicts.

### **Examples and Execution**

#### **Example 1: Installing and Using a Package**

**Description**: This example shows how to install and use the popular lodash package for JavaScript utilities.

**Steps**:

Create a new directory for your project:  
bash  
Copy code  
mkdir my\_project && cd my\_project

Initialize a new Node.js project:  
bash  
Copy code  
npm init -y

1. This creates a package.json file to manage project dependencies.

Install the lodash package:  
bash  
Copy code  
npm install lodash

Create a file named app.js and use lodash:  
javascript  
Copy code  
// Import lodash

const \_ = require('lodash');

// Use lodash to manipulate an array

const numbers = [10, 5, 3, 8];

const sortedNumbers = \_.sortBy(numbers);

console.log('Sorted Numbers:', sortedNumbers);

Run the code:  
bash  
Copy code  
node app.js

**Output**:  
less  
Copy code  
Sorted Numbers: [ 3, 5, 8, 10 ]

#### **Example 2: Running Custom npm Scripts**

**Description**: Use npm to automate a task with a custom script.

**Steps**:

In your my\_project directory, open the package.json file and add a script:  
json  
Copy code  
"scripts": {

"start": "node app.js",

"greet": "echo 'Hello from npm!'"

}

Run the custom script:  
bash  
Copy code  
npm run greet

**Output**:  
csharp  
Copy code  
Hello from npm!

Use the start script to run your app:  
bash  
Copy code  
npm start

#### **5. How npm Works**

* **npm install**: Installs packages into a node\_modules directory.
* **package.json**: Keeps track of installed packages and their versions.
* **package-lock.json**: Ensures exact versions of dependencies for consistent results.
* **npm uninstall**: Removes installed packages.
* **Global vs Local Installation**:
  + Local: Packages are available only to the current project.
  + Global: Packages are available system-wide.

---------------------------------------------------------------------------------------------------------------

### **Initializing a Node.js Project Using npm**

When you start a Node.js project, it is important to set up a **package.json** file. This file tracks dependencies, scripts, and other configuration information for your project. Here's a step-by-step guide to initializing a Node.js project.

### **Step-by-Step Guide**

#### **1. Create a Project Directory**

First, create a folder for your project. This is where all your files and dependencies will be stored.

bash

Copy code

mkdir my\_project

cd my\_project

#### **2. Initialize the Node.js Project**

Use the following npm command to generate a package.json file in your project directory:

bash

Copy code

npm init

This command will prompt you with several questions:

* **name**: The name of your project (default is the current folder name).
* **version**: The version of your project (default is 1.0.0).
* **description**: A brief description of your project.
* **entry point**: The main file for your app (default is index.js).
* **test command**: The command to run tests (you can leave it empty if you're not using testing yet).
* **repository**: The version control repository (e.g., GitHub URL, optional).
* **keywords**: Tags to help people find your project (optional).
* **author**: Your name.
* **license**: The license for your project (default is ISC).

You can skip most of these questions by pressing Enter, or just use npm init -y to automatically accept default values for all options.

#### **3. package.json File**

After running the npm init command, you'll see a package.json file created in your project directory. It will look something like this:

json

Copy code

{

"name": "my\_project",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "",

"license": "ISC"

}

* **name**: The name of your project.
* **version**: The version of your project.
* **main**: The main entry point file (this file will be executed when you run node <filename>).
* **scripts**: Contains commands that you can run using npm run <script-name>.
* **dependencies**: This section will store the external libraries you install.

#### **4. Install Packages/Dependencies**

Once your project is initialized, you can start installing packages (e.g., libraries like express, lodash, etc.). For example, to install the express package:

bash

Copy code

npm install express

This command will:

* Download the express library.
* Add it to the node\_modules/ folder.
* Automatically add express to the dependencies section of your package.json file:

json

Copy code

{

"dependencies": {

"express": "^4.17.1"

}

}

#### **5. Running npm Scripts**

You can define custom scripts to make development easier. For example, in your package.json, you can add a custom script to start your server. Update the scripts section:

json

Copy code

"scripts": {

"start": "node app.js"

}

To run the script:

bash

Copy code

npm start

This will execute node app.js and run your project.

#### **6. Managing Dependencies**

**Install a specific version of a package**:  
bash  
Copy code  
npm install express@4.16.0

**Install a package globally (useful for CLI tools)**:  
bash  
Copy code  
npm install -g nodemon

**Uninstall a package**:  
bash  
Copy code  
npm uninstall express

#### **7. package-lock.json File**

When you install packages using npm install, a package-lock.json file is created. This file locks the versions of installed packages to ensure that everyone working on the project uses the same versions.

### **Recap of Key Commands**

* **npm init**: Initializes a new Node.js project and creates a package.json file.
* **npm install <package-name>**: Installs a package locally for your project.
* **npm install -g <package-name>**: Installs a package globally (useful for CLI tools).
* **npm install**: Installs all dependencies listed in package.json.
* **npm start**: Runs the "start" script from package.json.
* **npm uninstall <package-name>**: Uninstalls a package and removes it from package.json.

### **Example of a Fully Initialized Node.js Project**

**Initialize your project**:  
bash  
Copy code  
mkdir my\_project && cd my\_project

npm init -y

**Install some packages** (e.g., express):  
bash  
Copy code  
npm install express

**Create app.js**:  
javascript  
Copy code  
const express = require('express');

const app = express();

const PORT = 3000;

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

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

});

app.listen(PORT, () => {

console.log(`Server running at http://localhost:${PORT}`);

});

**Add a custom start script** in package.json:  
json  
Copy code  
"scripts": {

"start": "node app.js"

}

**Run the project**:  
bash  
Copy code  
npm start

This should start an Express server, and you can visit http://localhost:3000 in your browser to see "Hello, World!"

------------------------------------------------------------------------------------------------------------------------

### **Example: Simple HTTP Server**

#### **Steps to Follow:**

**Create the Project Folder**Create a new folder for the project and navigate into it:  
bash  
Copy code  
mkdir my\_http\_project && cd my\_http\_project

**Initialize the Node.js Project**Run the following command to generate a package.json file:  
bash  
Copy code  
npm init -y

1. This will create a package.json file with default values.

**Create the server.js File**Now, create the server.js file that will contain the logic to set up the HTTP server. You can create this file in your project directory:  
bash  
Copy code  
touch server.js

**Write the Code for the HTTP Server**Open the server.js file and write the following code:  
javascript  
Copy code  
// Import the built-in HTTP module

const http = require('http');

// Set the hostname and port

const hostname = '127.0.0.1';

const port = 3000;

// Create the HTTP server

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

// Set the response HTTP headers

res.statusCode = 200; // OK status

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

// Serve a different response based on the URL

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

res.end('<h1>Welcome to the Home Page!</h1>');

} else if (req.url === '/about') {

res.end('<h1>This is the About Page</h1>');

} else {

res.statusCode = 404;

res.end('<h1>Page Not Found</h1>');

}

});

// Make the server listen on the specified port

server.listen(port, hostname, () => {

console.log(`Server running at http://${hostname}:${port}/`);

});

**Run the Server**In the terminal, run the following command to start the server:  
bash  
Copy code  
node server.js

1. **Access the Server in a Browser or Using curl**Open your browser and go to http://127.0.0.1:3000 or use curl to test the endpoints:

For the home page:  
bash  
Copy code  
curl http://127.0.0.1:3000/

**Output**:  
css  
Copy code  
<h1>Welcome to the Home Page!</h1>

For the about page:  
bash  
Copy code  
curl http://127.0.0.1:3000/about

**Output**:  
css  
Copy code  
<h1>This is the About Page</h1>

For a non-existent page:  
bash  
Copy code  
curl http://127.0.0.1:3000/unknown

**Output**:  
css  
Copy code  
<h1>Page Not Found</h1>

### **Breakdown of the Code:**

1. **http.createServer()**: This method creates an HTTP server. The callback function takes two arguments:
   * req: The incoming request (containing details like the URL and HTTP headers).
   * res: The outgoing response (used to set status codes, headers, and send the response body).
2. **res.statusCode = 200**: Sets the status code of the response (200 means the request was successful).
3. **res.setHeader()**: Sets the content type of the response, which tells the client (browser or tool) what type of data to expect.
4. **req.url**: Checks the URL of the request. Based on the URL, the server sends a different HTML response. If the URL is not recognized, the server responds with a 404 status code (Page Not Found).
5. **server.listen()**: Starts the server, making it listen on the specified hostname (localhost) and port (3000).

### **Running the Server with npm (Optional)**

To run the server using an npm script, you can add a start script to your package.json:

Open your package.json file and add this section:  
json  
Copy code  
"scripts": {

"start": "node server.js"

}

Now, you can run the server with the following command:  
bash  
Copy code  
npm start

This will execute node server.js and start the HTTP server, just like using the node command directly.

### **Conclusion**

This example shows how to build a simple Node.js HTTP server without using external libraries like Express. You created a basic server that can respond to different URLs with custom content. This approach gives you direct control over how the server handles requests and responses.

====================================================================

### **MySQL - DDL and DML Commands**

In MySQL (and most relational databases), commands are divided into different categories based on their function. The two most important categories are **DDL (Data Definition Language)** and **DML (Data Manipulation Language)** commands.

### **1. DDL (Data Definition Language) Commands**

**What are DDL commands?**DDL commands are used to **define, modify, or delete database structures**, such as tables, indexes, and schemas. These commands do not manipulate the actual data but control the structure of the database objects.

#### **Why is DDL Used?**

* To **create** new database objects (tables, views, etc.).
* To **modify** the structure of existing database objects (e.g., adding or removing columns from a table).
* To **delete** database objects when they are no longer needed.

#### **Common DDL Commands:**

1. **CREATE**: Used to create a new database or table.
2. **ALTER**: Used to modify the structure of an existing table (e.g., adding or deleting columns).
3. **DROP**: Used to delete an existing database or table.
4. **TRUNCATE**: Used to delete all rows from a table, but the structure of the table remains intact.

#### **Examples of DDL Commands:**

**Example 1: Creating a Table**

sql

Copy code

CREATE TABLE users (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(50),

email VARCHAR(100),

age INT

);

* This command creates a new table called users with four columns: id, name, email, and age.
* The id column is an auto-incrementing primary key, which means it will automatically generate a unique value for each new record.

**Example 2: Modifying a Table (ALTER)**

sql

Copy code

ALTER TABLE users ADD COLUMN phone VARCHAR(20);

* This command adds a new column phone to the users table.
* The data type of the new column is VARCHAR(20).

**Example 3: Deleting a Table (DROP)**

sql

Copy code

DROP TABLE users;

* This command completely deletes the users table and all the data it contains.

**Example 4: Deleting All Rows from a Table (TRUNCATE)**

sql

Copy code

TRUNCATE TABLE users;

* This command deletes all rows in the users table but keeps the table structure intact for future use.

### **2. DML (Data Manipulation Language) Commands**

**What are DML commands?**DML commands are used to **manipulate the data** within the tables. These commands allow you to **insert, update, delete, and retrieve** data from the tables.

#### **Why is DML Used?**

* To **insert** new data into a table.
* To **update** existing data in a table.
* To **delete** specific data from a table.
* To **retrieve** data from one or more tables.

#### **Common DML Commands:**

1. **INSERT**: Adds new records into a table.
2. **UPDATE**: Modifies existing records in a table.
3. **DELETE**: Removes records from a table.
4. **SELECT**: Retrieves data from a table.

#### **Examples of DML Commands:**

**Example 1: Inserting Data into a Table**

sql

Copy code

INSERT INTO users (name, email, age)

VALUES ('John Doe', 'john.doe@example.com', 30);

* This command inserts a new row into the users table with the values 'John Doe', 'john.doe@example.com', and 30 for the name, email, and age columns, respectively.
* Notice that we do not need to insert a value for the id column because it is an auto-increment field.

**Example 2: Updating Data in a Table**

sql

Copy code

UPDATE users

SET age = 31

WHERE name = 'John Doe';

* This command updates the age of the user named John Doe to 31.
* The WHERE clause is used to specify which rows should be updated. Without it, all rows would be updated.

**Example 3: Deleting Data from a Table**

sql

Copy code

DELETE FROM users

WHERE name = 'John Doe';

* This command deletes the row(s) where the name is 'John Doe' from the users table.
* Again, the WHERE clause is important to specify which rows should be deleted.

**Example 4: Selecting Data from a Table**

sql

Copy code

SELECT \* FROM users;

* This command retrieves all rows and columns from the users table.

The \* means "all columns". You can replace it with specific column names if you only need certain columns, e.g.:  
sql  
Copy code  
SELECT name, age FROM users;

* This would retrieve only the name and age columns from the users table.

### **How to Execute DDL and DML Commands**

#### **Executing DDL Commands**

1. Open MySQL terminal or MySQL Workbench (or any other MySQL client).

Log in to MySQL:  
bash  
Copy code  
mysql -u root -p

Create a database (if necessary):  
sql  
Copy code  
CREATE DATABASE my\_database;

Select the database:  
sql  
Copy code  
USE my\_database;

1. Execute the desired DDL command, e.g., CREATE TABLE or DROP TABLE.

#### **Executing DML Commands**

1. Once you've created a database and table, you can insert, update, or delete data:

Insert:  
sql  
Copy code  
INSERT INTO users (name, email, age) VALUES ('Jane Doe', 'jane.doe@example.com', 25);

Update:  
sql  
Copy code  
UPDATE users SET age = 26 WHERE name = 'Jane Doe';

Delete:  
sql  
Copy code  
DELETE FROM users WHERE name = 'Jane Doe';

Select:  
sql  
Copy code  
SELECT \* FROM users;

### **Key Differences Between DDL and DML:**

* **DDL (Data Definition Language)**:
  + Defines the structure of the database and its objects.
  + Examples: CREATE, ALTER, DROP, TRUNCATE.
  + Affects database schema and objects but not the data inside them.
* **DML (Data Manipulation Language)**:
  + Manages the data within the database.
  + Examples: INSERT, UPDATE, DELETE, SELECT.
  + Affects the records (rows) in the tables, not the structure.

### **Conclusion**

* **DDL commands** are used to manage the structure of databases and tables, such as creating or deleting tables, altering their structure, or truncating data.
* **DML commands** focus on managing the data itself, allowing you to insert new records, update existing ones, delete data, and retrieve data from the database.

========================================================================

To use **DDL** and **DML** commands with **Node.js** and **MySQL** (in XAMPP with phpMyAdmin), you'll need to use a Node.js MySQL client, like the popular mysql2 library. This allows Node.js to communicate with your MySQL database to execute SQL commands.

I'll guide you through the process in detail, following your request.

### **Setting Up the Environment:**

1. **Install MySQL**: Since you're using **XAMPP**, MySQL should already be installed and running. Ensure that the XAMPP control panel has MySQL started.

**Install the mysql2 Library**: We’ll use the mysql2 package to interact with MySQL in Node.js. Run the following command in your project directory to install it:  
bash  
Copy code  
npm install mysql2

1. **Database Configuration**: You'll need to configure the database connection in your Node.js project. You can find the connection details in phpMyAdmin under "User accounts". Usually, for XAMPP, the database connection details will look like this:
   * **Host**: localhost
   * **Username**: root (default for XAMPP)
   * **Password**: "" (empty by default in XAMPP)
   * **Database Name**: Your chosen database name (e.g., my\_database).

### **Creating a Database Connection:**

Create a database.js file to set up the connection to MySQL:

javascript

Copy code

const mysql = require('mysql2');

// Create a connection to the database

const connection = mysql.createConnection({

host: 'localhost', // MySQL server host

user: 'root', // MySQL username (default is 'root' for XAMPP)

password: '', // MySQL password (default is empty for XAMPP)

database: 'my\_database' // Name of the database you're using

});

// Connect to the database

connection.connect((err) => {

if (err) {

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

return;

}

console.log('Connected to the database.');

});

module.exports = connection;

### **Using DDL Commands in Node.js**

#### **1. Create a Table (DDL Command: CREATE)**

javascript

Copy code

const connection = require('./database.js');

// Create a table

const createTableQuery = `

CREATE TABLE users (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

email VARCHAR(100),

age INT

);

`;

connection.query(createTableQuery, (err, results) => {

if (err) {

console.error('Error creating table:', err);

return;

}

console.log('Table "users" created successfully');

});

#### **2. Alter a Table (DDL Command: ALTER)**

javascript

Copy code

const alterTableQuery = `

ALTER TABLE users

ADD COLUMN phone VARCHAR(20);

`;

connection.query(alterTableQuery, (err, results) => {

if (err) {

console.error('Error altering table:', err);

return;

}

console.log('Column "phone" added to the "users" table');

});

#### **3. Drop a Table (DDL Command: DROP)**

javascript

Copy code

const dropTableQuery = 'DROP TABLE users';

connection.query(dropTableQuery, (err, results) => {

if (err) {

console.error('Error dropping table:', err);

return;

}

console.log('Table "users" dropped successfully');

});

#### **4. Truncate a Table (DDL Command: TRUNCATE)**

javascript

Copy code

const truncateTableQuery = 'TRUNCATE TABLE users';

connection.query(truncateTableQuery, (err, results) => {

if (err) {

console.error('Error truncating table:', err);

return;

}

console.log('All data removed from the "users" table');

});

### **Using DML Commands in Node.js**

#### **1. Insert Data (DML Command: INSERT)**

javascript

Copy code

const insertDataQuery = `

INSERT INTO users (name, email, age)

VALUES ('John Doe', 'john.doe@example.com', 30)

`;

connection.query(insertDataQuery, (err, results) => {

if (err) {

console.error('Error inserting data:', err);

return;

}

console.log('Data inserted into the "users" table');

});

#### **2. Update Data (DML Command: UPDATE)**

javascript

Copy code

const updateDataQuery = `

UPDATE users

SET age = 31

WHERE name = 'John Doe'

`;

connection.query(updateDataQuery, (err, results) => {

if (err) {

console.error('Error updating data:', err);

return;

}

console.log('Data updated in the "users" table');

});

#### **3. Delete Data (DML Command: DELETE)**

javascript

Copy code

const deleteDataQuery = `

DELETE FROM users

WHERE name = 'John Doe'

`;

connection.query(deleteDataQuery, (err, results) => {

if (err) {

console.error('Error deleting data:', err);

return;

}

console.log('Data deleted from the "users" table');

});

#### **4. Select Data (DML Command: SELECT)**

javascript

Copy code

const selectDataQuery = 'SELECT \* FROM users';

connection.query(selectDataQuery, (err, results) => {

if (err) {

console.error('Error selecting data:', err);

return;

}

console.log('Data retrieved from "users" table:');

console.log(results);

});

* The result of the SELECT query will return an array of objects, where each object corresponds to a row in the users table. You can access individual columns by their names (e.g., results[0].name).

### **Executing the Commands**

You can create a main.js file and call the above code inside it to execute the SQL commands. Here's an example of how to structure your main.js:

javascript

Copy code

const connection = require('./database.js');

// Sample DML and DDL Commands execution

const createTableQuery = `

CREATE TABLE users (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

email VARCHAR(100),

age INT

);

`;

connection.query(createTableQuery, (err, results) => {

if (err) {

console.error('Error creating table:', err);

return;

}

console.log('Table "users" created successfully');

// Insert data after creating the table

const insertDataQuery = `

INSERT INTO users (name, email, age)

VALUES ('John Doe', 'john.doe@example.com', 30)

`;

connection.query(insertDataQuery, (err, results) => {

if (err) {

console.error('Error inserting data:', err);

return;

}

console.log('Data inserted into the "users" table');

// Now, select the data

const selectDataQuery = 'SELECT \* FROM users';

connection.query(selectDataQuery, (err, results) => {

if (err) {

console.error('Error selecting data:', err);

return;

}

console.log('Data retrieved from "users" table:');

console.log(results);

connection.end(); // Close the connection

});

});

});

### **Conclusion**

By using the mysql2 library, you can easily execute both **DDL** and **DML** commands in your Node.js application. This allows you to create tables, insert data, update records, delete data, and query your MySQL database, all from within your Node.js application.

====================================================================

Here’s an example of how you can organize all the **DDL** and **DML** commands inside a single main.js file:

### **Example: main.js file with DDL and DML commands**

javascript

Copy code

const mysql = require('mysql2');

// Create a connection to the database

const connection = mysql.createConnection({

host: 'localhost', // MySQL server host

user: 'root', // MySQL username (default is 'root' for XAMPP)

password: '', // MySQL password (default is empty for XAMPP)

database: 'my\_database' // Name of the database you're using

});

// Connect to the database

connection.connect((err) => {

if (err) {

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

return;

}

console.log('Connected to the database.');

});

// Create a table (DDL Command: CREATE)

const createTableQuery = `

CREATE TABLE IF NOT EXISTS users (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

email VARCHAR(100),

age INT

);

`;

connection.query(createTableQuery, (err, results) => {

if (err) {

console.error('Error creating table:', err);

return;

}

console.log('Table "users" created successfully');

// Insert data (DML Command: INSERT)

const insertDataQuery = `

INSERT INTO users (name, email, age)

VALUES ('John Doe', 'john.doe@example.com', 30),

('Jane Doe', 'jane.doe@example.com', 25),

('Sam Smith', 'sam.smith@example.com', 22);

`;

connection.query(insertDataQuery, (err, results) => {

if (err) {

console.error('Error inserting data:', err);

return;

}

console.log('Data inserted into the "users" table');

// Select data (DML Command: SELECT)

const selectDataQuery = 'SELECT \* FROM users';

connection.query(selectDataQuery, (err, results) => {

if (err) {

console.error('Error selecting data:', err);

return;

}

console.log('Data retrieved from "users" table:');

console.log(results);

// Update data (DML Command: UPDATE)

const updateDataQuery = `

UPDATE users

SET age = 31

WHERE name = 'John Doe'

`;

connection.query(updateDataQuery, (err, results) => {

if (err) {

console.error('Error updating data:', err);

return;

}

console.log('Data updated in the "users" table');

// Delete data (DML Command: DELETE)

const deleteDataQuery = `

DELETE FROM users

WHERE name = 'Sam Smith'

`;

connection.query(deleteDataQuery, (err, results) => {

if (err) {

console.error('Error deleting data:', err);

return;

}

console.log('Data deleted from the "users" table');

// Alter table (DDL Command: ALTER)

const alterTableQuery = `

ALTER TABLE users

ADD COLUMN phone VARCHAR(20)

`;

connection.query(alterTableQuery, (err, results) => {

if (err) {

console.error('Error altering table:', err);

return;

}

console.log('Column "phone" added to the "users" table');

// Drop table (DDL Command: DROP)

const dropTableQuery = 'DROP TABLE users';

connection.query(dropTableQuery, (err, results) => {

if (err) {

console.error('Error dropping table:', err);

return;

}

console.log('Table "users" dropped successfully');

connection.end(); // Close the connection after everything is done

});

});

});

});

});

});

});

### **What’s Happening Here?**

1. **Creating the Table** (CREATE):
   * The first query attempts to create the users table if it doesn't already exist. It defines columns id, name, email, and age.
2. **Inserting Data** (INSERT):
   * After the table is created successfully, we insert three rows of sample data into the users table.
3. **Selecting Data** (SELECT):
   * Once data is inserted, we retrieve all rows from the users table and log the results to the console.
4. **Updating Data** (UPDATE):
   * We then update the age of John Doe to 31.
5. **Deleting Data** (DELETE):
   * Next, we delete the record where the name is Sam Smith.
6. **Altering the Table** (ALTER):
   * After updating and deleting, we modify the users table by adding a new column phone.
7. **Dropping the Table** (DROP):
   * Finally, we drop the entire users table, removing both the table and all its data.

### **Important Notes:**

* **Sequential Execution**: The commands are executed sequentially, ensuring that each command runs after the previous one has finished. This is crucial when you're working with DDL commands, as dropping or altering a table affects the structure of the database.
* **Error Handling**: If any error occurs in one of the queries, the error will be logged, and subsequent commands won’t run. This helps in debugging.
* **Database Connection**: We call connection.end() at the end to close the connection after all queries are executed.

### **Executing the File:**

1. Make sure MySQL is running in your XAMPP control panel.
2. Ensure your my\_database exists in phpMyAdmin.

Run your main.js file:  
bash  
Copy code  
node main.js

1. Check the console for logs and check phpMyAdmin for any changes in your database.

=====================================================================

To call individual rows from the result of a SELECT query in MySQL using Node.js, you can access the data returned in the results array. Each row in the result is an object where the keys are the column names, and the values are the column values for that row.

Here’s an example showing how you can access individual rows and columns from the result of a SELECT query:

### **Example: Accessing Individual Rows**

javascript

Copy code

const mysql = require('mysql2');

// Create a connection to the database

const connection = mysql.createConnection({

host: 'localhost',

user: 'root',

password: '',

database: 'my\_database' // Ensure this is the correct database name

});

// Select data from the "users" table

const selectDataQuery = 'SELECT \* FROM users';

connection.query(selectDataQuery, (err, results) => {

if (err) {

console.error('Error selecting data:', err);

return;

}

// Iterate through each row in the result

results.forEach((row, index) => {

// Access individual columns by their names (e.g., 'name', 'email', 'age')

console.log(`Row ${index + 1}:`);

console.log(`Name: ${row.name}`);

console.log(`Email: ${row.email}`);

console.log(`Age: ${row.age}`);

console.log('-------------------------');

});

// Close the connection

connection.end();

});

### **Explanation:**

1. **results**:
   * The results array contains the rows retrieved from the users table. Each element of the results array is an object representing a row, with properties corresponding to the column names.
2. **Accessing Columns**:
   * To access individual columns in a row, you use the column name as a property of the row object. For example, row.name will give you the value of the name column, and row.email will give you the value of the email column.
3. **Iterating Through Rows**:
   * The forEach method iterates through each row in the results array, allowing you to log or process each row individually.

### **Output Example:**

Assuming the users table has the following rows:

| **id** | **name** | **email** | **age** |
| --- | --- | --- | --- |
| 1 | John Doe | john.doe@example.com | 30 |
| 2 | Jane Doe | jane.doe@example.com | 25 |
| 3 | Sam Smith | sam.smith@example.com | 22 |

The output from the above script would be:

yaml

Copy code

Row 1:

Name: John Doe

Email: john.doe@example.com

Age: 30

-------------------------

Row 2:

Name: Jane Doe

Email: jane.doe@example.com

Age: 25

-------------------------

Row 3:

Name: Sam Smith

Email: sam.smith@example.com

Age: 22

-------------------------

This method allows you to process each row individually and access specific columns as needed.

==================================================================

Let's dive into **Server-Side Applications** in **Node.js**. This is an essential topic, especially since Node.js is commonly used for building server-side applications.

### **What is a Server-Side Application?**

A **server-side application** is an application that runs on a server rather than on the client's device (like a browser). It handles requests from clients (e.g., browsers or mobile devices), processes data, and sends responses back to the client.

In Node.js, a **server-side application** is typically built to listen for incoming HTTP requests, process those requests, interact with databases, and respond with data (HTML, JSON, etc.).

### **Why Use Node.js for Server-Side Applications?**

1. **Non-blocking I/O**: Node.js uses an event-driven, non-blocking I/O model, which is ideal for handling a large number of concurrent connections.
2. **Fast**: Since Node.js uses **Google's V8 JavaScript engine**, it is very fast, making it an excellent choice for server-side applications.
3. **Single Language for Frontend & Backend**: Node.js uses JavaScript for both the frontend (browser) and backend (server), making development simpler and more efficient.
4. **Scalability**: Node.js is designed to be highly scalable, supporting applications that need to handle many concurrent connections (like real-time applications).

### **Building a Simple Server-Side Application in Node.js**

To understand how server-side applications work in Node.js, we’ll start by building a **basic HTTP server** that can handle client requests and return responses.

### **Step-by-Step Example of a Server-Side Application**

**Create a New Project Directory**: Create a new folder for your project:  
bash  
Copy code  
mkdir nodejs-server

cd nodejs-server

1. **Create a server.js File**: In the nodejs-server folder, create a file named server.js.

**Write Basic HTTP Server Code**:  
javascript  
Copy code  
const http = require('http'); // Import the http module

// Create a server that listens to requests

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

// Set the response header to indicate the content type

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

// Write the response body

res.write('Hello, this is a server-side response!');

// End the response

res.end();

});

// Make the server listen on port 3000

server.listen(3000, () => {

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

});

1. **Explanation of the Code**:
   * **http.createServer()**: This method creates an HTTP server. The function inside it takes two arguments: the request (req) and the response (res).
     + **req**: Contains information about the client's request (e.g., URL, headers).
     + **res**: Used to send a response back to the client.
   * **res.writeHead(200)**: Sets the HTTP status code to 200, which means "OK" (success).
   * **res.write('...')**: Sends data back to the client. In this case, it's a simple text message.
   * **server.listen(3000)**: The server listens on port 3000. Once the server is running, you can access it by navigating to http://localhost:3000 in your browser.

**Run the Server**: To run your server, execute the following command in the terminal:  
bash  
Copy code  
node server.js

1. **Test the Server**: Open a browser and go to http://localhost:3000. You should see the message: **"Hello, this is a server-side response!"**

### **Handling Different HTTP Methods**

You can make the server respond differently based on the HTTP method (GET, POST, PUT, DELETE, etc.) that the client sends.

Here’s how you can modify your server to handle a **GET request** and a **POST request**:

javascript

Copy code

const http = require('http');

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

// Handle GET request

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

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

res.write('This is a response to a GET request!');

res.end();

}

// Handle POST request

if (req.method === 'POST') {

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

res.write('This is a response to a POST request!');

res.end();

}

});

// Listen on port 3000

server.listen(3000, () => {

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

});

* **req.method**: This property allows you to check which HTTP method the client is using. In this example, we check if the method is GET or POST and respond accordingly.

### **Building a More Complex Server-Side Application with Routing**

In a real-world application, you’d want to handle different routes (URLs) and provide different responses based on the requested route. For example, you might want the server to return different pages or data based on the path.

Here’s a more complex example where the server handles multiple routes:

javascript

Copy code

const http = require('http');

const url = require('url'); // To parse the URL of the request

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

const parsedUrl = url.parse(req.url, true); // Parse the request URL

if (parsedUrl.pathname === '/') {

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

res.write('<h1>Welcome to the Home Page</h1>');

res.end();

} else if (parsedUrl.pathname === '/about') {

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

res.write('<h1>About Us</h1><p>This is the About page.</p>');

res.end();

} else {

res.writeHead(404, { 'Content-Type': 'text/plain' });

res.write('Page Not Found');

res.end();

}

});

// Listen on port 3000

server.listen(3000, () => {

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

});

### **Explanation:**

* **url.parse()**: This method is used to parse the URL of the incoming request, which gives us the path (/, /about, etc.).
* **Routing**: We use if statements to check the requested path (parsedUrl.pathname) and respond accordingly.
* **404 Handling**: If the requested route does not exist, we return a 404 Not Found error.

### **Running the Server:**

Open your terminal and run the server:  
bash  
Copy code  
node server.js

1. Go to the following URLs in your browser:
   * http://localhost:3000/ → Home page
   * http://localhost:3000/about → About page
   * http://localhost:3000/any-other-path → 404 Page Not Found

### **Conclusion**

In this topic, we’ve covered the basics of creating **server-side applications** in Node.js. We:

1. Learned how to set up a basic HTTP server.
2. Handled different HTTP methods (GET, POST).
3. Implemented basic **routing** to respond to different URLs.

Node.js is powerful for building fast, scalable server-side applications, and as you continue to explore it, you’ll be able to add more complex functionality like handling form submissions, connecting to databases, managing sessions, and much more.

==================================================================

### **Authentication vs Authorization**

Before we proceed with Node.js implementation, it’s important to understand the distinction between **authentication** and **authorization**:

* **Authentication**: Verifying the identity of a user or system. It is the process of ensuring that someone is who they claim to be. For example, logging in with a username and password.
* **Authorization**: Determining whether the authenticated user has permission to access a specific resource or perform an action. For example, determining whether a user can view a particular page or access certain data.

In simpler terms:

* **Authentication**: "Who are you?"
* **Authorization**: "What are you allowed to do?"

### **How to Implement Authentication and Authorization in Node.js**

In Node.js, we often use libraries like **Passport.js** for authentication, and **JWT (JSON Web Tokens)** for both authentication and authorization.

### **Steps to Implement Authentication and Authorization**

1. **Set up the Project**:
   * Create a new Node.js project or use an existing one.
   * Initialize the project with npm init -y.
2. **Install Required Packages**: You'll need a few packages to help with authentication and authorization:
   * **express**: For creating the server and handling routes.
   * **bcryptjs**: For hashing passwords.
   * **jsonwebtoken**: For generating and verifying JWT tokens.
   * **dotenv**: For managing environment variables.
   * **body-parser**: To parse incoming request bodies.

Install these dependencies using npm:  
bash  
Copy code  
npm install express bcryptjs jsonwebtoken dotenv body-parser

### **Example 1: Simple Authentication with JWT**

This example will demonstrate basic **authentication** using **JWT**. We’ll create a simple login system where the server authenticates the user and returns a JWT token.

#### **Step-by-Step Example**

**Create a .env file** for storing your secret key:  
bash  
Copy code  
touch .env

Add your secret key in .env:  
makefile  
Copy code  
JWT\_SECRET=your\_jwt\_secret\_key

1. **Create the server.js File**:

javascript

Copy code

const express = require('express');

const bcrypt = require('bcryptjs');

const jwt = require('jsonwebtoken');

const dotenv = require('dotenv');

const bodyParser = require('body-parser');

// Load environment variables from .env file

dotenv.config();

// Initialize Express

const app = express();

const port = 3000;

// Middleware to parse JSON bodies

app.use(bodyParser.json());

// Dummy in-memory user database

const users = [

{

id: 1,

username: 'user1',

password: '$2a$10$u5lRp2Fq0XO4T5GjFjXI4udZQ0w8H1BftZXpBO6dGjOwlRzX7waXa' // 'password123' hashed

}

];

// Route for user login (Authentication)

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

const { username, password } = req.body;

// Find the user

const user = users.find(u => u.username === username);

if (!user) {

return res.status(400).json({ message: 'Invalid credentials' });

}

// Compare the password with the hashed one stored

bcrypt.compare(password, user.password, (err, isMatch) => {

if (err) {

return res.status(500).json({ message: 'Server error' });

}

if (!isMatch) {

return res.status(400).json({ message: 'Invalid credentials' });

}

// Create a JWT token after successful authentication

const token = jwt.sign({ userId: user.id, username: user.username }, process.env.JWT\_SECRET, { expiresIn: '1h' });

// Send the token to the client

res.json({ message: 'Login successful', token });

});

});

// Route for a protected resource (Authorization)

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

// Get the token from the Authorization header

const token = req.header('Authorization')?.split(' ')[1];

if (!token) {

return res.status(401).json({ message: 'Access denied. No token provided.' });

}

// Verify the token

jwt.verify(token, process.env.JWT\_SECRET, (err, decoded) => {

if (err) {

return res.status(401).json({ message: 'Invalid token' });

}

// Access granted

res.json({ message: 'Access granted', user: decoded });

});

});

// Start the server

app.listen(port, () => {

console.log(`Server is running on http://localhost:${port}`);

});

#### **Explanation of the Code:**

1. **Login Route (/login)**:
   * Accepts the username and password in the request body.
   * Finds the user from the dummy users database.
   * Uses **bcryptjs** to compare the entered password with the stored hashed password.
   * If authentication is successful, a **JWT token** is created using **jsonwebtoken** and sent to the client.
2. **Protected Route (/protected)**:
   * The client needs to send the **JWT token** in the Authorization header as Bearer <token>.
   * The server verifies the token using the jwt.verify() method.
   * If the token is valid, the server grants access to the protected resource.

### **Running the Example**

**Start the server**:  
bash  
Copy code  
node server.js

1. **Login**: You can log in using **Postman** or **curl**:

**POST** request to http://localhost:3000/login with the following JSON body:  
json  
Copy code  
{

"username": "user1",

"password": "password123"

}

If the login is successful, you will receive a response containing the **JWT token**:  
json  
Copy code  
{

"message": "Login successful",

"token": "<JWT\_Token>"

}

1. **Access Protected Route**:
   * To access the protected route, make a **GET** request to http://localhost:3000/protected and include the JWT token in the Authorization header as Bearer <JWT\_Token>.

If the token is valid, you will get a response:  
json  
Copy code  
{

"message": "Access granted",

"user": {

"userId": 1,

"username": "user1"

}

}

If the token is invalid or not provided, you will get an error response:  
json  
Copy code  
{

"message": "Access denied. No token provided."

}

### **Example 2: Authorization with Roles**

You can extend this example by adding **roles** to the JWT payload and checking user roles to authorize access to specific resources.

Here’s how you can modify the /protected route to restrict access based on roles:

1. **Modify the login route to include roles**:
   * Modify the token creation to include a role (e.g., "admin" or "user").
2. **Check the role in the protected route**:

javascript

Copy code

// Role-based Authorization

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

const token = req.header('Authorization')?.split(' ')[1];

if (!token) {

return res.status(401).json({ message: 'Access denied. No token provided.' });

}

jwt.verify(token, process.env.JWT\_SECRET, (err, decoded) => {

if (err) {

return res.status(401).json({ message: 'Invalid token' });

}

// Check if the user is an admin

if (decoded.role !== 'admin') {

return res.status(403).json({ message: 'Access denied. Insufficient privileges.' });

}

res.json({ message: 'Welcome, admin!' });

});

});

Now, only users with the **admin** role can access the /admin route. You can set the role when creating the token in the login route:

javascript

Copy code

const token = jwt.sign(

{ userId: user.id, username: user.username, role: 'admin' },

process.env.JWT\_SECRET,

{ expiresIn: '1h' }

);

### **Conclusion**

In this section, we covered:

1. **Authentication**: Verifying the identity of the user using **JWT**.
2. **Authorization**: Granting or denying access to resources based on user roles.
3. **JWT Tokens**: Used for securely transmitting information between the client and the server.

**JWT** provides a robust way to handle authentication and authorization in server-side applications, and it’s commonly used with Node.js for building secure APIs.

================================================================

### **1. Using the Built-In fs Module**

You can serve static HTML files directly using Node.js's built-in **fs module** (file system). Here's how:

#### **Example: Serving an HTML File**

1. **Create an HTML File**:

Create a file named index.html in your project directory:  
html  
Copy code  
<!DOCTYPE html>

<html>

<head>

<title>Node.js with HTML</title>

</head>

<body>

<h1>Welcome to Node.js with HTML!</h1>

<p>This is served using Node.js and the fs module.</p>

</body>

</html>

**Serve the HTML File with Node.js**: Create a server.js file:  
javascript  
Copy code  
const http = require('http');

const fs = require('fs');

// Create a server

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

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

// Read the HTML file

fs.readFile('index.html', (err, data) => {

if (err) {

res.writeHead(500, { 'Content-Type': 'text/plain' });

res.end('Server Error');

} else {

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

res.end(data); // Send the HTML file content

}

});

} else {

res.writeHead(404, { 'Content-Type': 'text/plain' });

res.end('Page Not Found');

}

});

// Start the server

server.listen(3000, () => {

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

});

**Run the Server**:  
bash  
Copy code  
node server.js

1. Visit http://localhost:3000 in your browser. You'll see your HTML file rendered.

### **2. Using the express Framework**

While the fs module works for serving static files, using a framework like **Express** makes handling HTML much easier.

#### **Steps for Serving HTML with Express**

**Install Express**:  
bash  
Copy code  
npm install express

1. **Create an HTML File**: Use the same index.html file from the previous example.

**Serve the HTML File**: Create a new server.js file:  
javascript  
Copy code  
const express = require('express');

const path = require('path');

const app = express();

// Serve the HTML file

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

res.sendFile(path.join(\_\_dirname, 'index.html'));

});

// Start the server

const port = 3000;

app.listen(port, () => {

console.log(`Server is running on http://localhost:${port}`);

});

**Run the Server**:  
bash  
Copy code  
node server.js

1. Visit http://localhost:3000 in your browser to see the HTML page.

### **3. Using HTML Templates for Dynamic Content**

If you want to dynamically render content on your HTML pages, you can use template engines like **EJS**, **Pug**, or **Handlebars**.

#### **Example: Dynamic HTML with EJS**

**Install EJS**:  
bash  
Copy code  
npm install ejs

**Set Up the Project**: Create the following structure:  
bash  
Copy code  
/views

└ index.ejs

server.js

1. **Create an index.ejs File**:

Inside the views folder, create index.ejs:  
html  
Copy code  
<!DOCTYPE html>

<html>

<head>

<title><%= title %></title>

</head>

<body>

<h1>Welcome, <%= user %>!</h1>

<p>This is dynamically generated using EJS.</p>

</body>

</html>

**Create the server.js File**:  
javascript  
Copy code  
const express = require('express');

const app = express();

// Set EJS as the template engine

app.set('view engine', 'ejs');

// Route for rendering the HTML

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

res.render('index', { title: 'Dynamic HTML with EJS', user: 'John Doe' });

});

// Start the server

const port = 3000;

app.listen(port, () => {

console.log(`Server is running on http://localhost:${port}`);

});

**Run the Server**:  
bash  
Copy code  
node server.js

1. Visit http://localhost:3000. You’ll see the dynamic content (e.g., the user’s name) rendered in the HTML.

### **4. Serving Static Files (CSS, Images, JS)**

To serve static files (like CSS, images, or JavaScript), you can use Express's built-in middleware.

#### **Example:**

**Folder Structure**:  
arduino  
Copy code  
/public

└ style.css

index.html

server.js

**CSS File (/public/style.css)**:  
css  
Copy code  
body {

font-family: Arial, sans-serif;

background-color: #f0f0f0;

color: #333;

}

h1 {

color: #007BFF;

}

**HTML File (index.html)**:  
html  
Copy code  
<!DOCTYPE html>

<html>

<head>

<link rel="stylesheet" href="/style.css">

<title>Static Files Example</title>

</head>

<body>

<h1>Hello, Node.js with Static Files!</h1>

</body>

</html>

**Server (server.js)**:  
javascript  
Copy code  
const express = require('express');

const path = require('path');

const app = express();

// Serve static files from the "public" directory

app.use(express.static(path.join(\_\_dirname, 'public')));

// Serve the HTML file

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

res.sendFile(path.join(\_\_dirname, 'index.html'));

});

// Start the server

const port = 3000;

app.listen(port, () => {

console.log(`Server is running on http://localhost:${port}`);

});

**Run the Server**:  
bash  
Copy code  
node server.js

1. Visit http://localhost:3000. You’ll see the HTML page styled with the CSS file.

### **Summary**

1. **Integrating HTML with Node.js** can be done in different ways:
   * Using the fs module for simple static HTML.
   * Using **Express** for more flexible routing and serving files.
   * Using template engines like **EJS** for dynamic HTML generation.
2. You can also serve static files (CSS, images, JavaScript) using Express's static middleware.

Each approach depends on your application needs. If you’re building a dynamic web application, Express with a template engine or even front-end frameworks like React or Angular (served via Node.js) might be the way to go.

======================================================================

### **Project Overview**

1. **Authentication**: Users must log in to access the form.
2. **HTML Form**: The form collects user details (name, address, age, gender, phone, and email).
3. **Database Integration**: Form data is saved in a MySQL database.
4. **Node.js Backend**: Handles user authentication, form submission, and database interaction.

### **1. Setup**

#### **Folder Structure**

scss

Copy code

/project

├── /public

│ ├── style.css (Optional: CSS file for styling)

├── /views

│ ├── login.html (HTML login page)

│ ├── form.html (HTML form for user data entry)

├── .env (Environment variables)

├── server.js (Main Node.js server file)

├── package.json (Node.js project file)

#### **Install Required Packages**

bash

Copy code

npm init -y

npm install express mysql2 bcryptjs jsonwebtoken body-parser dotenv

### **2. Database Setup**

#### **Create a Database and Tables**

1. Open **phpMyAdmin** (http://localhost/phpmyadmin).
2. Create a new database: html\_node\_project.
3. Create two tables:
   * **users**: For authentication.
   * **user\_details**: To store form data.

sql

Copy code

CREATE TABLE users (

id INT AUTO\_INCREMENT PRIMARY KEY,

username VARCHAR(50) NOT NULL,

password VARCHAR(255) NOT NULL

);

CREATE TABLE user\_details (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

address TEXT,

age INT,

gender ENUM('Male', 'Female'),

phone VARCHAR(15),

email VARCHAR(100)

);

1. Insert a default user for login:

sql

Copy code

INSERT INTO users (username, password)

VALUES ('admin', '$2a$10$rCknXwiMi9Q3MvH4NcGcSeC9ZdBv/ULdmufw2CbkDl8XZaHdV7rr6'); -- Password: admin123

### **3. Code Implementation**

#### **3.1. Environment Variables (.env)**

Create a .env file:

env

Copy code

PORT=3000

JWT\_SECRET=your\_jwt\_secret

DB\_HOST=localhost

DB\_USER=root

DB\_PASSWORD=

DB\_NAME=html\_node\_project

#### **3.2. Backend (server.js)**

This file handles:

1. User login (authentication).
2. Authorization via JWT.
3. Saving form data to MySQL.

javascript

Copy code

const express = require('express');

const bodyParser = require('body-parser');

const mysql = require('mysql2');

const bcrypt = require('bcryptjs');

const jwt = require('jsonwebtoken');

const dotenv = require('dotenv');

const path = require('path');

dotenv.config();

const app = express();

const port = process.env.PORT;

// Middleware

app.use(bodyParser.urlencoded({ extended: true }));

app.use(bodyParser.json());

app.use(express.static(path.join(\_\_dirname, 'public')));

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

res.sendFile(path.join(\_\_dirname, "views", "login.html"));

});

// Database connection

const db = mysql.createConnection({

host: process.env.DB\_HOST,

user: process.env.DB\_USER,

password: process.env.DB\_PASSWORD,

database: process.env.DB\_NAME,

});

// Connect to the database

db.connect(err => {

if (err) {

console.error('Database connection error:', err);

} else {

console.log('Connected to the database.');

}

});

// User login route

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

const { username, password } = req.body;

db.query('SELECT \* FROM users WHERE username = ?', [username], (err, results) => {

if (err) return res.status(500).json({ error: 'Database error' });

if (results.length === 0) return res.status(401).json({ error: 'Invalid credentials' });

const user = results[0];

bcrypt.compare(password, user.password, (err, isMatch) => {

if (err || !isMatch) {

return res.status(401).json({ error: 'Invalid credentials' });

}

// Generate JWT

const token = jwt.sign({ userId: user.id, username: user.username }, process.env.JWT\_SECRET, { expiresIn: '1h' });

res.json({ message: 'Login successful', token });

});

});

});

// Save form data (protected route)

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

const token = req.header('Authorization')?.split(' ')[1];

if (!token) {

return res.status(401).json({ error: 'Access denied. No token provided.' });

}

jwt.verify(token, process.env.JWT\_SECRET, (err, decoded) => {

if (err) {

return res.status(401).json({ error: 'Invalid token' });

}

const { name, address, age, gender, phone, email } = req.body;

const query = 'INSERT INTO user\_details (name, address, age, gender, phone, email) VALUES (?, ?, ?, ?, ?, ?)';

db.query(query, [name, address, age, gender, phone, email], (err, result) => {

if (err) {

return res.status(500).json({ error: 'Database error' });

}

res.json({ message: 'Data saved successfully' });

});

});

});

// Start the server

app.listen(port, () => {

console.log(`Server running at http://localhost:${port}`);

});

#### **3.3. Login Page (/views/login.html)**

html

Copy code

<!DOCTYPE html>

<html>

<head>

<title>Login</title>

</head>

<body>

<h2>Login</h2>

<form action="/login" method="POST">

<label>Username:</label>

<input type="text" name="username" required><br><br>

<label>Password:</label>

<input type="password" name="password" required><br><br>

<button type="submit">Login</button>

</form>

</body>

</html>

#### **3.4. Data Entry Form (/views/form.html)**

html

Copy code

<!DOCTYPE html>

<html>

<head>

<title>User Details Form</title>

</head>

<body>

<h2>User Details Form</h2>

<form action="/submit" method="POST">

<label>Name:</label>

<input type="text" name="name" required><br><br>

<label>Address:</label>

<textarea name="address" required></textarea><br><br>

<label>Age:</label>

<input type="number" name="age" required><br><br>

<label>Gender:</label>

<input type="radio" name="gender" value="Male" required> Male

<input type="radio" name="gender" value="Female" required> Female<br><br>

<label>Phone:</label>

<input type="text" name="phone" required><br><br>

<label>Email:</label>

<input type="email" name="email" required><br><br>

<button type="submit">Submit</button>

</form>

</body>

</html>

### **4. Step-by-Step Guide**

1. **Start MySQL (XAMPP)**:
   * Open XAMPP and ensure MySQL is running.
   * Verify the html\_node\_project database and users, user\_details tables exist.

**Start the Node.js Server**:  
bash  
Copy code  
node server.js

1. **Test Login**:
   * Visit http://localhost:3000/login in your browser.
   * Use the credentials: **Username**: admin, **Password**: admin123.
2. **Submit Data**:
   * After logging in, go to http://localhost:3000/form.html.
   * Fill in the form and click "Submit".
3. **Verify Data in MySQL**:
   * Check the user\_details table in phpMyAdmin to confirm the data was saved.

### **Conclusion**

This example covers:

1. **Authentication and Authorization** using **JWT**.
2. **HTML Form Integration** with Node.js.
3. **MySQL Database** to store data.
4. Secure **Password Hashing** using **bcryptjs**.