MERN-1 : Intro to nodejs and backend development

Topics to cover in Node:

1. Exploring different node inbuilt modules
2. Understanding server side development with node (http)
3. Understanding http module with req and res
4. Serving static html sites & json files via the server

Nodejs

Node.js is an open-source, server-side JavaScript runtime environment that allows you to run JavaScript code on the server. It was created by Ryan Dahl and first released in 2009.

Node.js is built on the V8 JavaScript engine, which is also used in Google Chrome, and it provides an event-driven, non-blocking I/O model that makes it well-suited for building scalable and high-performance network applications.

Why use nodejs for web server

1. Node.js is a popular choice for web servers, especially in scenarios involving heavy I/O operations and small server

requirements. Here's why Node.js is a suitable option for such use cases:

2. Non-Blocking I/O Model:
    a. Node.js is designed around a non-blocking, event-driven architecture. This means it can efficiently handle multiple I/O operations concurrently without blocking the execution of other tasks.
    b. When performing heavy I/O operations, such as reading and writing files, making network requests, or interacting with databases, Node.js can initiate these operations and continue executing other code while waiting for the I/O operations to complete. This asynchronous approach is highly advantageous for scenarios with many concurrent I/O tasks.
3. Scalability:
    a. In situations involving heavy I/O, it's common for multiple clients to make simultaneous requests to the server. Node.js's non-blocking model allows it to handle a large number of concurrent connections efficiently, making it a suitable choice for scalable applications. It can process incoming requests as soon as they arrive, rather than waiting for each request to complete before moving on to the next one.
4. Low Overhead:
    a. Node.js has a relatively small memory footprint compared to some other web server technologies. This makes it

well-suited for small server applications where resource utilization needs to be efficient. You can run multiple Node.js instances on a single server without consuming excessive system resources.

5. Rich Ecosystem:
    a. Node.js has a vast ecosystem of libraries and modules available through npm, which can simplify the development of web servers for various purposes. Developers can find packages to handle specific I/O tasks, such as file uploads, database connections, and HTTP requests, making it easier to build web servers tailored to their needs.
    b. https://github.com/sindresorhus/awesome-nodejs

title: Fs module

The fs module in Node.js stands for "File System," and it provides a way to work with the file system on your computer or server. It allows you to read from and write to files, manipulate directories, perform file operations, and more. Let's explore some of the key functionalities of the fs module in-depth:

1. Reading Files:

The fs module provides methods for reading the contents of files. The most commonly used method for this purpose is fs.readFile():

```
const fs = require("fs"); // file system module ; Common Js module
```

```
// import fs from "fs"; // ES6 module



fs.readFile("file.txt", "utf8", (err, data) => {
 // error first callback
 if (err) {
   console.log(err);
   return;
 } else {
   console.log("data from file", data);
 }
});
```

2. Writing Files:

You can also use the fs module to write data to files using methods like fs.writeFile():

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

const content = "This is a new file created by Node.js";
fs.writeFile("example.txt", content, "utf8", (err) => {
 if (err) {
   console.log(err);
   return;
 }
 console.log("File created successfully");
});
```

3.Synchronous vs. Asynchronous Operations:

Most fs module functions come in both synchronous and asynchronous versions. The asynchronous versions (e.g., fs.readFile()) allow non-blocking file operations, while synchronous

versions (e.g., fs.readFileSync()) block the Node.js event loop until the operation is complete.

Asynchronous methods are typically preferred in Node.js to maintain the application's responsiveness.

```
fs.rename("example.txt", "new-example.txt", (err) => {
 if (err) {
   console.log(err);
   return;
 }
 console.log("File renamed successfully");
});

fs.unlink("new-example.txt", (err) => {
 if (err) {
   console.log(err);
   return;
 }
 console.log("File deleted successfully");
});
```

6. File Statistics:

The fs.stat() method provides information about a file's status, including its size, permissions, and modification timestamp.

```
fs.stat('file.txt', (err, stats) => {
   if (err) {
     console.error(err);
     return;
   }
```

```
  console.log('File size: ' + stats.size);
  console.log('Is directory? ' + stats.isDirectory());
});
```

Now lets deal with directories

Working with Directories:

You can perform operations on directories using methods like fs.mkdir(), fs.rmdir(), fs.readdir(), and fs.stat(). These methods allow you to create, remove, list, and get information about directories, respectively.

```
const directoryName = 'my-directory';

fs.mkdir(directoryName, (err) => {
 if (err) {
   console.error(`Error creating directory: ${err}`);
 } else {
   console.log(`Directory "${directoryName}" created successfully.`);
 }
});
```

Deleting the directory

```
fs.rmdir(directoryName, { recursive: true }, (err) => {
   if (err) {
     console.error(`Error deleting directory: ${err}`);
   } else {
     console.log(`Directory "${directoryName}" deleted successfully.`);
   }
 });
```

In this code, fs.rmdir() is used to delete the "my-directory" directory. The { recursive: true } option is provided to ensure that the directory

and its contents are deleted recursively. The callback function is called when the deletion is complete, and any errors are logged.

Make sure to handle errors appropriately when creating or deleting directories in your Node.js applications to ensure that your code is robust and reliable.

You can check whether a directory or file exists in Node.js using the fs module. Here's how you can do it:

Checking if a Directory Exists:

To check if a directory exists, you can use the fs.existsSync() method.

Create a directory dir

```
const directoryPath = '/my-directory';

if (fs.existsSync(directoryPath)) {
 console.log(`The directory "${directoryPath}" exists.`);
} else {
 console.log(`The directory "${directoryPath}" does not exist.`);
}
```

Checking if a File Exists:

```
const filePath = './dir/file1.txt';

if (fs.existsSync(filePath)) {
 console.log(`The file "${filePath}" exists.`);
} else {
 console.log(`The file "${filePath}" does not exist.`);
```

```
}
```

Extra notes

It's worth noting that fs.existsSync() is a synchronous method, so it can block the Node.js event loop. If you prefer an asynchronous approach, you can use the fs.access() method. Here's an example:

```
fs.access(filePath, fs.constants.F_OK, (err) => {
 if (err) {
   console.log(`The path "${filePath}" does not exist.`);
 } else {
   console.log(`The path "${filePath}" exists.`);
 }
});
```

fs.constants.F_OK is used as a flag in the fs.access method to check the existence of a file or directory at the specified path (filePath)

Path Module

The path module in Node.js provides utilities for working with file and directory paths. It's an essential module when dealing with file system operations and path manipulation in your Node.js applications. Here are some important functions and concepts from the path module:

File path formats differ between operating systems. For instance, Windows uses backslashes (\) while POSIX (Linux, macOS) uses forward slashes (/). The path module abstracts these differences,

allowing you to write code that works across all platforms without worrying about the underlying file system differences.

path.join([...paths]):

This method joins multiple path segments into a single path string, taking care of platform-specific path separators (e.g., backslashes on Windows and forward slashes on Unix-like systems).

```
const path = require('path');
const fullPath = path.join('folder', 'subfolder', 'file.txt');
```

path.resolve([...paths]):

Resolves an absolute path from multiple path segments, starting from the root directory. It can be helpful for creating absolute paths based on relative ones.

```
const absolutePath = path.resolve('folder', 'subfolder', 'file.txt');
```

path.basename(path[, ext]):

Return the last portion of a path. Similar to the Unix basename command. Often used to extract the file name from a fully qualified path

```
const fileName = path.basename('./dir/file.txt');
```

path.dirname(path):

Returns the directory name of a path.

```
const dirName = path.dirname('/path/to/file.txt');
console.log(dirName)
```

path.extname(path):

Returns the file extension of a path, including the dot.

```
const extension = path.extname('/path/to/file.txt');
console.log(extension)
```

path.parse(pathString):

Parses a path string into an object with properties like root, dir, base, name, and ext.

```
const pathInfo = path.parse('/path/to/file.txt');
console.log(pathInfo)
```

path.normalize(path):

Normalizes a path by resolving '..' and '.' segments and converting slashes to the appropriate platform format.

```
const normalizedPath = path.normalize('/path/to/../file.txt');
console.log(normalizedPath)
```

path.isAbsolute(path):

Checks if a path is an absolute path.

```
const isAbsolute = path.isAbsolute('/path/to/file.txt');
console.log(isAbsolute)
```

POSIX Systems: On POSIX systems (MacOS, Linux based systems) , path.isAbsolute returns true if the path starts with a /.
Windows Systems: On Windows systems, path.isAbsolute returns true if the path starts with a drive letter followed by a colon and a backslash (e.g., C:\)

path.relative(from, to):

Returns the relative path from one path to another.

```
const relativePath = path.relative('/path/from', '/path/to');
console.log(relativePath)
```

The path module is particularly useful when working on cross-platform applications or when dealing with file and directory paths dynamically in your Node.js code. It ensures that your path manipulation is consistent and compatible with various operating systems.

Copy a file from one folder to another in Node.js

To copy a file from one folder to another in Node.js, you can use the fs module and a concept of streams in NOde

a. Streams are collections of data that might not be available all at once and don't have to fit in memory. They allow handling of reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way
b. Like in video streams, A portion of the video is buffered (pre-loaded) and played while the next portion is being downloaded. This approach ensures a smooth viewing experience without requiring the entire file to be downloaded first.

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

// Define the source and destination file paths
const sourceFilePath = './dir/file.txt';
const destinationFilePath = './destination-file.txt';

// Create a readable stream from the source file
const readStream = fs.createReadStream(sourceFilePath);

// Create a writable stream to the destination file
const writeStream = fs.createWriteStream(destinationFilePath);

// Pipe the data from the source file to the destination file
readStream.pipe(writeStream);

// Handle any errors that may occur during the copy process
readStream.on('error', (err) => {
 console.error(`Error reading the source file: ${err}`);
});

writeStream.on('error', (err) => {
 console.error(`Error writing to the destination file: ${err}`);
});

// When the copy is complete, log a success message
writeStream.on('finish', () => {
```

```
 console.log('File copied successfully.');
});
```

It uses the fs.createReadStream() method to create a readable stream from the source file.

It uses the fs.createWriteStream() method to create a writable stream to the destination file.

It uses the .pipe() method to pipe the data from the source stream to the destination stream, effectively copying the file.
It sets up error event listeners on both the source and destination streams to handle any errors that may occur during the copy process.
It sets up a finish event listener on the destination stream to log a success message when the copy is complete.

This code will copy the contents of the source file to the destination file. If the destination file already exists, it will be overwritten. Make sure to handle errors and adjust file paths as needed for your specific use case.

title: Server side development

Server-side development, client-side development, and working with database clients are essential components of modern web application development. Let's explore these concepts:

1. Server-Side Development:

Server-side development refers to the part of web application development that occurs on the server, typically using server-side technologies and programming languages. Here are key aspects:

a. Server: A server is a computer or software application that responds to client requests over a network. In web development, a server typically hosts the backend of a web application.
b. Server-Side Technologies: Common server-side technologies include Node.js, Python (with frameworks like Django or Flask), Ruby (with Ruby on Rails), Java (with Spring or Java EE), PHP, and more. These technologies enable you to create the server logic, handle requests from clients, interact with databases, and generate dynamic content.
c. Server Logic: Server-side code manages user authentication, business logic, data processing, and database interactions. It often generates HTML, JSON, or other data to send back to the client.
d. Security: Security measures like input validation, authentication, authorization, and protecting against common vulnerabilities (e.g., SQL injection, XSS) are typically implemented on the server side.

2. Client-Side Development:

Client-side development focuses on the part of web application development that occurs in the user's web browser. Here are key aspects:

a. Client: The client is the user's device (e.g., web browser) that sends requests to a server to access web content or services.
b. Client-Side Technologies: Common client-side technologies include HTML, CSS, and JavaScript. HTML is used for structuring web content, CSS for styling, and JavaScript for adding interactivity and functionality to web pages.
c. Front-End Frameworks: Developers often use front-end frameworks and libraries like React, Angular, or Vue.js to build complex and responsive user interfaces.
d. User Experience: Client-side development is responsible for creating an engaging and user-friendly experience. This includes handling user interactions, form validations, and rendering dynamic content without requiring full page reloads.
e. Performance: Optimizing client-side performance is crucial, as the client device has limited resources. Techniques like lazy loading, minification, and caching are employed to enhance the user experience.

3. Database Client:

A database client is a software component or library that allows your server-side code to communicate with a database management system (DBMS). Here are key aspects:

a. Database Management System (DBMS): A DBMS is software that manages databases, including storing, retrieving, updating, and organizing data. Examples of DBMSs include MySQL, PostgreSQL, MongoDB, and SQLite.
b. Database Client Libraries: To interact with a DBMS, developers use specific client libraries or drivers for their chosen programming language. These libraries provide functions and methods to connect to the database, execute queries, and retrieve results.
c. ORM (Object-Relational Mapping): Some server-side frameworks and languages offer ORMs (e.g., Sequelize for Node.js, Hibernate for Java) that provide a higher-level abstraction for working with databases. ORMs map database tables to objects in code, simplifying database interactions.
d. ODM: Models data using schemas and documents.Example of ODM (Mongoose for MongoDB)
e. ORM: Models data using classes and tables.
f. Data Access: Server-side code uses database clients to perform CRUD operations (Create, Read, Update, Delete) on data stored in databases. This includes querying data, inserting new records, updating existing records, and deleting records.

Server-side development using the http module

Server-side development using the http module in Node.js allows you to create a basic HTTP server to handle incoming requests and send responses.

1. Start by requiring the http module in your Node.js script:

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

2. Create the HTTP Server:

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

const server = http.createServer((req, res) => {
  // Handle incoming requests here
});
```

3. Handle Incoming Requests:
Inside the callback function, you can handle incoming HTTP requests. The req object represents the request, and the res object is used to send the response back to the client.

```
const server = http.createServer((req, res) => {
  // Set response header
  res.setHeader('Content-Type', 'text/plain');

  // Write response content
  res.write('Hello, World!');

  // End the response
  res.end();
});
```

In this example, we set the Content-Type header to 'text/plain', write "Hello, World!" as the response content, and then end the response.

4. Specify the Listening Port and Host:
You need to specify the port and host (usually 'localhost' for development) on which your server will listen for incoming requests:

```
const port = 3000;
 const host = 'localhost';
  server.listen(port, host, () => {
   console.log(`Server is listening on http://${host}:${port}`);
 });
```

you can start the server by calling the server.listen() method. This will start listening for incoming HTTP requests on the specified port and host.

5. Test Your Server:

Run your Node.js script, and your server will be accessible at the specified URL (e.g., http://localhost:3000). You can use a web browser or tools like cURL or Postman to send HTTP requests to your server.

This simple HTTP server will respond with "Hello, World!" to any incoming request. You can expand upon this foundation to handle different types of requests and serve dynamic content based on the requested URL or route.

You can modify the code to send an HTML response containing an <h1> tag. Here's an example of a Node.js HTTP server that responds with an HTML <h1> tag:

```
const server = http.createServer((req, res) => {
 // Set response header with Content-Type as text/html
 res.setHeader("Content-Type", "text/html");

 // Write HTML response
 res.write("<html><head><title>Node.js HTTP Server</title></head><body>");
 res.write("<h1>Hello, World!</h1>");
 res.write("</body></html>");

 // End the response
 res.end();
});
```

Ports

a. What is a Port Number?
b. In the context of computer networking, a port number is a way to identify a specific process or service within a device that uses the Internet Protocol (IP).
c. **Why the concept of ports was invented if it is not a physical thing?**
d. Think of ports as a mechanism for multitasking. While an IP address identifies a machine, many different applications and services run on that machine at once. Each of these services needs a way to receive the appropriate data packets without confusion. Ports allow a single machine with one IP address to efficiently manage

multiple services simultaneously. By assigning different services to different ports, a computer can easily determine which application should handle incoming data.

2. Valid Range: Port numbers can range from 0 to 65535.
3. Well-Known Ports: 0-1023, reserved for system and standard services.
4. Registered Ports: 1024-49151, typically used for user applications.
5. Dynamic/Private Ports: 49152-65535, used for dynamic purposes.

Nodemon

Nodemon is a tool that helps you develop Node.js applications by automatically restarting the Node.js process whenever changes are detected in your project's files. It's particularly useful during development because it eliminates the need to manually stop and restart your Node.js application every time you make code changes.

Installation
You can install Nodemon globally using npm (Node Package Manager) with the following command:

npm install -g nodemon

Alternatively, you can install it as a development dependency in your project by running the following command inside your project directory:

npm install --save-dev nodemon

Basic Usage:
After installing Nodemon, you can use it to run your Node.js application instead of the standard node command. For example:

nodemon your-app.js

Json in the response and setting the header

```
const server = http.createServer((req, res) => {
 // Set response header with Content-Type as application/json
 res.setHeader("Content-Type", "application/json");

 // Define JSON data
 const jsonData = {
   message: "Hello, World!",
   date: new Date(),
 };

 // Convert JSON object to a JSON string
 const jsonResponse = JSON.stringify(jsonData);

 // Write JSON response
 res.write(jsonResponse);

 // End the response
 res.end();
});
```

The body of an HTTP response should be a string or Buffer, basically in a format that can be transmitted over the network. JSON, as an object, needs to be converted into a string format to meet this requirement.