NODEJS WITH EXPRESSJS COMPLETE NOTES

MUHAMMAD ZOHAIB IRSHAD

Table of Contents

1. HELLO WORLD IN NODEJS

To run a JavaScript file:

You have to write “node index.js” in the node js terminal to run the file and where index.js is a JavaScirpt file.

To start a project:

To start a project you have to take following steps:

- Open terminal of in visual studio
- Write command “npm init” to start a project.
- Then you to input these values:

Name	Value
Package name :	
Version :	
Description :	
Entry Point :	
Test Command :	
Git Repository :	
Key words :	
Author :	
License :	

- You can also skip these values blank by pressing enter.
- Then your package.json file will be created.
- You can also add more scripts in the package.json file.
- For this purpose, in package.json there will be scripts where you can write :

```
"scripts": {

  "test": "echo \"Error: no test specified\" && exit 1",
  "start":"node index.js"

}
```

- And then you have to just write “npm start” in the terminal and your project will be started.

---

2. MODULES IN NODEJS

In Node.js, modules are essential building blocks that allow you to organize and reuse code efficiently. A module in Node.js is simply a file or a collection of functions and objects that are grouped together. These modules can either be built-in (like fs for file system operations) or custom (written by you or third-party developers).

Types of modules:

1. Core modules:

These are built-in modules provided by Node.js and are available for use without any installation.

For example:

- **http:** To create web servers
- **fs:** To interact with the file system
- **path:** To work with file and directory paths
- **os:** To interact with the operating system
- **url:** To interact with the url’s.

How to install:

- You have to enter command “npm install url” to install the url modules on your machine from npmjs.com site.

2. Custom modules:

You can create your own modules by grouping related functionality in a separate file. These modules are called custom modules. You can export functions, objects, or values from one file and import them into another file.

3. Third party modules:

These are modules that you can install from the npm (Node Package Manager) registry. You can install them using the npm install command.

Example: Express (npm install express), Lodash, etc.

Action	Code	Explanation
To import module in main file	const var= require();	require is a built in function in node js that will impot a custom module or Core Modules.
To export a module	module.exports= export-value;	In place of “export-value” you will give a function, string, etc. to export,

Example of Importing a function through variable:

Let suppose we have the index.js file and this file we have the code:

```
console.log("Math value is:", add(4,5));
```

and in the module.js file you have the code:

```
function add(a,b){
    return a+b;
}
```

To link the module.js file with the index.js file you have to export function from module.js and import function in index.js. For this purpose you have to do following:

1. Write

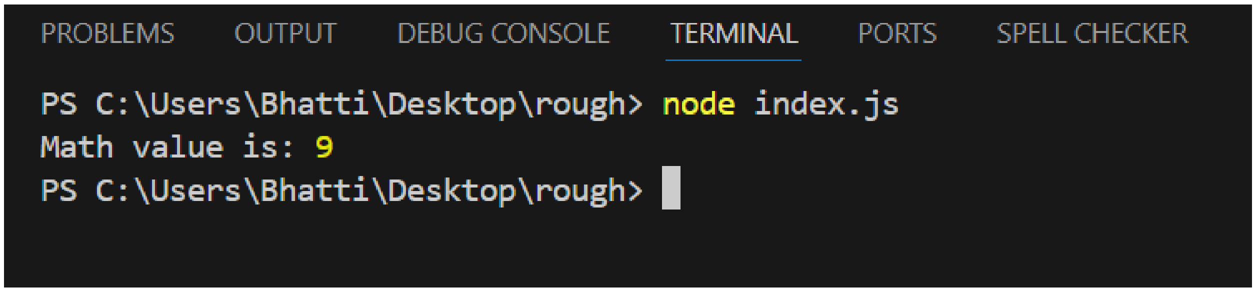
```
module.exports=add;
```

in the module.js file to export add function.

2. And write

```
const add= require('./math.js');
```

in the index.js file to import function from module.js file. When you will write node index.js then you will get following output.



```
PS C:\Users\Bhatti\Desktop\rough> node index.js
Math value is: 9
PS C:\Users\Bhatti\Desktop\rough>
```

When you will export two or more functions then you have to export in form of objects. Upper example can be given as:

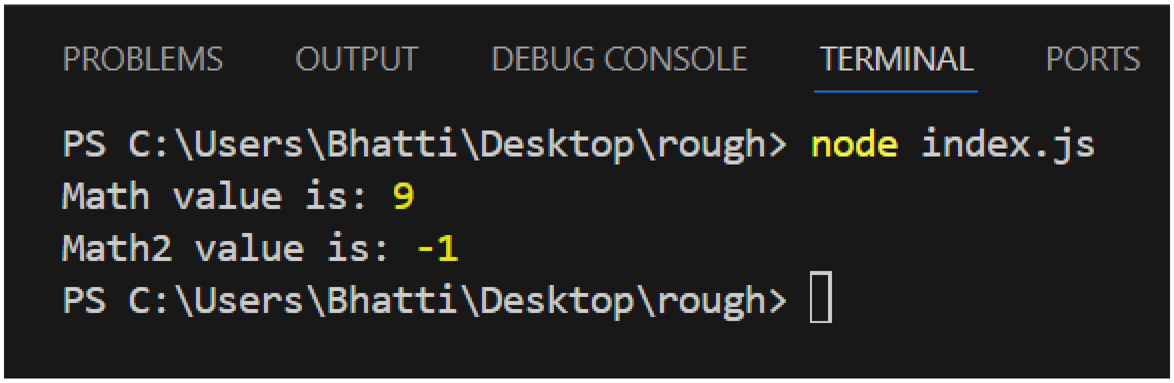
Index.js file code:

```
const {add, sub}= require('./module.js');

console.log("Math value is:", add(4,5));
console.log("Math2 value is:", sub(4,5));
```

module.js file code:

```
function add(a,b){
    return a+b;
}
function sub(a,b){
    return a-b;
}

module.exports={
    add, sub,
};
```

Output:



```
PS C:\Users\Bhatti\Desktop\rough> node index.js
Math value is: 9
Math2 value is: -1
PS C:\Users\Bhatti\Desktop\rough> 
```

Explanation:

In the upper code of module.js file, we have exported functions in form of object. It is mandatory to do that when you will export two or more functions to avoid overlapping.

In the same way, we imported function in form of structures and then utilizes the functions.

We use the function name because we received the function in structures as it is. Like in first example we received add function in form of variable so we used this variable to run the add function.

---

3. FILE HANDLING IN NODEJS

The file handling in NodeJS provides various methods to interact with the file system, allowing you to perform operations such as reading, writing, updating, and deleting files. There are no such functions in JavaScript due to security purposes.

First of all you will import module of file system by including following code in your file:

```
const file= require("fs");
```

Different operations you can perform with the files are given below in the table:

Code	Work

`file.writeFile("./test.txt","I am feeling good! Alhamdulilah.",(err)=>{});`	It will create file test.txt in the current directory with the text “I a feeling good! Alhamdulilah.”
`file.writeFile("test.txt", "lorem ipsum dollar imit", (err)=>{});`	It will create a file test.txt in the current directory with text “lorem ipsum dollar imit”.

`let result=`
`file.readFileSync("./test.txt","utf-8");`
`console.log(result);`
	It will print the file content of the test.txt file. While

`file.readFile("./test.txt", "utf-8",`
`(err, result)=>{`
`      if(err){`
`    console.log("Error", err)`
`      }`
`      else{`
`          console.log(result);`
`      }`
`  })`
	It will print test.txt file content.

`file.appendFileSync("./test.txt", "golo molo g aur tolo molo g hi hae\n" )`	It will add content “golo molo g aur tolo molo g hi hae\n” in the test.txt file.
`file.cpSync("./test.txt","./copy.txt");`	It will create the copy.txt file in the current directory as a copy of test.txt file.
`file.unlinkSync("./copy.txt");`	It will delete the copy.txt file from the current directory.
`console.log(file.statSync("./test.txt"));`	It will give the stats of the file such as when and how file is created, etc.
`file.mkdirSync("my-folder");`	It will create a folder with name my-folder in the current directory.

4. HOW NODEJS WORKS

The working of a **Node.js server** can be understood by examining how it handles **blocking** and **non-blocking** requests. These concepts define how Node.js processes operations and determines whether the application waits for a task to complete before moving on to the next one.

Key Concepts:

1. **Single-Threaded Event Loop**:

- Node.js operates on a **single-threaded event loop**, which is optimized for handling **non-blocking I/O operations**.
- This allows Node.js to efficiently handle many requests simultaneously without creating a new thread for each one.

2. **Blocking Requests**:

- A **blocking request** is one where the server waits (or blocks) for a task to complete before processing the next operation.
- This can cause delays and degrade performance because no other requests can be processed until the current operation finishes.

3. **Non-Blocking Requests**:

- A **non-blocking request** allows the server to continue processing other operations while waiting for the completion of the current task.
- Node.js achieves this using callbacks, promises, or async/await.

Working of Node.js Server

1. With Blocking Requests

Blocking requests are typically associated with synchronous operations.

What Happens:

- When a request is made to the server, it reads the file using fs.readFileSync, a synchronous (blocking) operation.
- The server waits for the file to be completely read before sending a response.
- While waiting, the server cannot handle other incoming requests.

Downside:

If the file is large or the operation is time-consuming, it blocks the event loop, causing the server to become unresponsive for other requests.

2. With Non-Blocking Requests

Non-blocking requests use asynchronous operations, allowing the server to continue handling other tasks while waiting for the I/O operation to complete.

What Happens:

- When a request is made, the server starts reading the file asynchronously using fs.readFile.
- While waiting for the file read operation to complete, the server can handle other incoming requests.
- Once the file read operation finishes, the callback function sends the response to the client.

Benefits:

- The server remains responsive and can handle multiple requests simultaneously.
- This is the default and recommended approach for high-performance applications in Node.js.

---

5. SERVER IN NODEJS

A server in Node.js is a program that listens for and responds to requests from clients over a network. Node.js, being a runtime environment built on Chrome's V8 JavaScript engine, is well-suited for building lightweight, efficient, and scalable servers.

Core Concept of a Server in Node.js

- **Listening** for incoming requests (e.g., HTTP, HTTPS, WebSocket).
- **Processing** those requests.
- **Sending back** the appropriate response.

Node.js is often used to create web servers because of its non-blocking, event-driven architecture. This means it can handle multiple requests concurrently without getting "stuck" waiting for one task to complete.

Code:

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

const fs=require("fs");

const myServer=http.createServer((req,res)=>{
    const log=`${Date.now()}: New request received.\n`;
    fs.appendFile("log.txt",log,(err,data)=>{res.end("Hello
from the server once again.")})
});
myServer.listen(8000,()=>{console.log("server started!")});
```

First of all, create new project by using my upper tutorial of hello world. Then run this code and then enter localhost:8000 in browser to check if server is receiving requests.

Here is code explanation:

Explanation:

1) Importing the required modules:

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

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

- **http**: The http module is used to create a server and handle HTTP requests and responses.
- **fs**: The fs module allows interaction with the file system, such as reading, writing, and appending files.

2) Creating HTTP Server:

```
const myServer=http.createServer((req,res)=>{
    const log=`${Date.now()}: New request received.\n`;
    fs.appendFile("log.txt",log,(err,data)=>{res.end("Hello
from the server once again.")})
});
```

http.createServer((req, res) => { ... }):

a) Creates an HTTP server and takes a callback function that is executed whenever the server receives a request.
b) Parameters:

   i. **req**: The request object, containing information about the incoming request (e.g., URL, headers, method).
   ii. **res**: The response object, used to send a response back to the client.

Inside the callback:

- **const log = \${Date.now()}: New request received.\n`;`**:

  a) Generates a log entry with the current timestamp (using Date.now()) and a message indicating that a new request was received.
  b) The log ends with a newline character (\n).

- **fs.appendFile("log.txt", log, (err, data) => { ... })**:

  a) Appends the log message to a file named log.txt. If the file doesn't exist, it will be created.
  b) Parameters:

     i. "log.txt": The file to which the log will be appended.
     ii. log: The log message to append.
     iii. (err, data) => { ... }: A callback function that runs after the append operation finishes.
        1. If there's an error (err), it's not explicitly handled here.
        2. Regardless, the server sends a response back to the client.

- **res.end("Hello from the server once again.");**:

  a) Ends the HTTP response and sends the text "Hello from the server once again." to the client.

3) Starting the Server:

```
• mySerer.listen(8000,()=>{console.log("server
  started!")});
```

myServer.listen(8000, () => { ... }):

a) Starts the server and listens on port 8000 for incoming connections.
b) The second argument is a callback function that runs when the server starts successfully.

console.log("server started!");:

a) Outputs the message "server started!" to the console, indicating that the server is running.

---

6. HANDLING URL's IN NODEJS

A **URL (Uniform Resource Locator)** is the complete web address used to identify resources (web pages, files, APIs, etc.) on the internet. It provides a way to locate resources and access them through protocols like HTTP or HTTPS.

Components of URL:

Let's explain with an example;

Example:

```
protocol://hostname:port/pathname?query#hash
```

Let's explain the components of URL using upper example.

Component	Defination	Example's Part
Protocol	Specifies the protocol used to access the resource (e.g., http, https, ftp)	`https://`

Hostname	The domain or IP address of the server where the resource is located.	`www.example.com`
Port	The port number on the server (optional). Defaults are 80 for HTTP and 443 for HTTPS	`:8080`
Pathname	The specific path to the resource on the server.	`/products/item123`
Query String	A set of key-value pairs providing additional data to the server, starting with ?.	`?id=42&color=red`
Hash	A fragment identifier pointing to a section of the resource.	`#section2`

Example of Code to explain terms of URL handling in Nodejs:

```
const url = require('url');
const querystring = require('querystring');

// Example URL
const myURL =
'https://www.example.com:8080/products/item123?id=42&color=red#section
2';

// Parse the URL using url.parse()
const parsedURL = url.parse(myURL);

// Access various components of the URL
console.log('Full URL:', parsedURL.href);              // Full URL
console.log('Protocol:', parsedURL.protocol);          // Protocol:
https:
console.log('Host:', parsedURL.host);                  // Host:
www.example.com:8080
console.log('Hostname:', parsedURL.hostname);          // Hostname:
www.example.com
console.log('Port:', parsedURL.port);                  // Port: 8080
console.log('Pathname:', parsedURL.pathname);          // Pathname:
/products/item123
console.log('Query string:', parsedURL.query);         // Query
string: id=42&color=red
```

```
console.log('Hash:', parsedURL.hash);                    // Hash:
#section2



// Parse the query string into an object
const queryParams = querystring.parse(parsedURL.query);
console.log('Query Parameters:', queryParams);           // Query
Parameters: { id: '42', color: 'red' }

// Access specific query parameters
console.log('ID:', queryParams.id);                      // ID: 42
console.log('Color:', queryParams.color);                // Color: red
```

Output of this code will be:

```
Full URL:
https://www.example.com:8080/products/item123?id=42&color=red#section2
Protocol: https:
Host: www.example.com:8080
Hostname: www.example.com
Port: 8080
Pathname: /products/item123
Query string: id=42&color=red
Hash: #section2
Query Parameters: { id: '42', color: 'red' }
ID: 42
Color: red
```

Functions of terms used in upper code :

Term	Function
`url.parse()`	Parses a URL string into an object containing its components (Deprecated in favor of the URL class).
`url.href`	Returns the full URL string.
`url.protocol`	Returns the protocol used in the URL (e.g., http: or https:).
`url.host`	Returns the hostname and port (if present) together (e.g., www.example.com:8080).
`url.hostname`	Returns the hostname without the port (e.g., www.example.com).

`url.port`	Returns the port number specified in the URL (e.g., 8080).
`url.pathname`	Returns the path of the URL after the host (e.g., /products/item123).
`url.query`	Returns the query string part of the URL as a raw string (e.g., id=42&color=red).
`querystring.parse()`	Converts the query string into an object of key-value pairs (e.g., { id: '42', color: 'red' }).
`url.hash`	Returns the fragment identifier (hash) starting with # (e.g., #section2).

7. HTTP METHODS

HTTP methods (also known as HTTP verbs) define the type of action to be performed on a resource when communicating between a client and a server. These methods are a core part of the HTTP protocol.

Common HTTP Methods:

HTTP Method	Description	Use-Case
GET	Used to request data from a server without modifying it.	Fetching a webpage or API data. i.e, when we put a url in browser and browser uses GET request to server to bring the webpage.
POST	Used to send data to a server to create or update the server data.	Filling the signup page to login into the Facebook.
PUT	Used to put data on the server.	Uploading files on the Facebook.
PATCH	Used to update or replace an existing data with the data provided.	Updating the Facebook account information.
DELETE	Used to delete a part of data on the server.	Deleting the blog post or Facebook account.

8. INTRODUCTION TO EXPRESSJS

Express.js is a fast, minimalist, and flexible web application framework for Node.js. It simplifies the process of creating web servers and APIs by providing powerful tools and utilities.

Installation:

After starting the project, you have to enter following command in the terminal while NodeJS presence is important:

```
npm install express
```

Hello World in ExpressJS:

```
const express = require('express');
const app = express();

// Define a route
app.get('/', (req, res) => {
    res.send('Hello World!');
});

// Start the server
app.listen(8000, () => {
    console.log("Server running at http://localhost:8000");
});
```

Check result:

Run "node file-name" to start the server then enter `http://localhost:8000` in the browser to send request to the server.

9. VERSIONING IN NPM

When working with Node.js projects, dependencies in the package.json file often include version numbers with special characters like ^ and ~. These characters define how npm should handle version updates when installing or updating packages.

Version Format:

4.18.01
MAJOR.MINOR.PATCH

MAJOR: Introduces breaking changes.
Example: Moving from 4.x.x to 5.x.x.
MINOR: Adds new features but keeps backward compatibility.
Example: Moving from 4.18.x to 4.19.x.
PATCH: Fixes bugs and makes backward-compatible improvements.
Example: Moving from 4.18.1 to 4.18.2.

i. Caret (^):

The ^ symbol allows updates **only for the same MAJOR version**, i.e., it permits updates to MINOR and PATCH versions.

- **Example 1**:
  ^4.18.1
  - Permitted versions: 4.18.1, 4.19.0, 4.20.3, etc.
  - Not permitted: 5.0.0 (because it's a new MAJOR version).
- **Behavior**: It's safe for automatic updates because MAJOR versions often introduce breaking changes.

ii. Tilde (~):

The ~ symbol allows updates **only for the same MINOR version**, i.e., it permits updates to PATCH versions.

- **Example 1**:
  ~4.18.4
  - Permitted versions: 4.18.4, 4.18.5, 4.18.6, etc.
  - Not permitted: 4.19.0 or 5.0.0 (new MINOR or MAJOR versions).

- **Behavior**: Tighter control compared to ^, allowing updates only for PATCH releases.

---

10. INTRODUCTION TO RESTFULL API

RESTful API stands for Representational State Transfer Application Programming Interface. It is a set of rules and conventions for building and interacting with web services. RESTful APIs are based on the principles of REST architecture, which is a lightweight and scalable approach to designing APIs.

Difference Between RESTful API and Generic API:

Rest Api	Generic Api
A specific type of API that adheres to REST principles and uses standard HTTP methods for communication.	The term "API" broadly refers to an interface or contract between software components that allows them to communicate. It can be implemented in various ways (SOAP, RPC, GraphQL, etc.).

Key Principles of a RESTful API

i. **Server-Client Architecture**:
   a. The architecture separates the **client** (frontend or consumer of the API) from the **server** (backend providing resources).
   b. This separation promotes scalability, flexibility, and reusability.
   c. The server only provides data and does not handle how the client uses or presents the data.

ii. **Stateless Communication**:
   a. Each request from the client to the server must contain all the information needed to process it.
   b. The server does not store client state between requests, making it highly scalable and reliable.

iii. **Uniform Interface**:

All requests are made to defined resources (e.g., /users, /orders) with standard HTTP methods:

i. **GET**: Retrieve data.
ii. **POST**: Create new resources.
iii. **PUT**: Update existing resources (entire resource).
iv. **PATCH**: Update part of a resource.
v. **DELETE**: Remove a resource.

iv. **Resource-Based Design**:

Resources (like users, products, orders) are identified using URIs (e.g., https://api.example.com/users).

v. **HTTP Methods**:

RESTful APIs respect and fully utilize HTTP methods for different operations. Each method has a specific purpose:

i. GET should not modify server state (idempotent).
ii. POST creates a new resource (not idempotent).
iii. PUT and DELETE should be idempotent (repeated calls yield the same result).

vi. **Representation of Resources**:

Resources can be represented in various formats, such as JSON, XML, or plain text.

JSON is the most commonly used format due to its lightweight nature.

vii. **Caching**:

RESTful APIs allow responses to be cacheable, reducing the need for repeated requests to the server.

viii. **Layered System**:

The architecture allows intermediate layers (e.g., load balancers, proxies) to exist between the client and server for better scalability.

Server-Side Rendering (SSR) vs. Client-Side Rendering (CSR):

1) Server-Side Rendering (SSR):

In SSR, the server generates the complete HTML for a page and sends it to the client. The browser displays the fully rendered HTML as soon as it is received.

Advantages:

1. Faster initial page load (HTML is ready on arrival).
2. Better for SEO (search engines can easily crawl server-rendered pages).
3. Suitable for content-heavy websites like blogs or e-commerce sites.

Disadvantages:

1. Slower subsequent interactions (requires reloading or fetching a new page).
2. Increased load on the server for rendering pages.

2) **Client-Side Rendering (CSR):**

While in CSR, the server sends a barebones HTML page along with JavaScript. The browser executes the JavaScript to render the page dynamically on the client-side.

Advantages:

1. Faster subsequent interactions (no full-page reloads; only data updates).
2. More interactive and fluid user experiences.
3. Reduces server-side rendering workload.

Disadvantages:

1. Slower initial page load (JavaScript must execute first).
2. SEO challenges (search engines may struggle to index JavaScript-rendered content).
3. Heavier load on the client's device.

---

11. INTRODUCTION TO EXPRESS MIDDLEWARE:

In Express.js, middleware refers to functions that execute during the request-response cycle. They have access to the request object (req), the response object (res), and the next middleware function (next).

Middleware is used to:

1. Modify requests or responses.
2. Log information.
3. Run authentication checks.
4. Handle errors.

Example:

```
function (req, res, next) {
    // Middleware logic
    next(); // Pass control to the next middleware or route
handler
}
```

Working of Middleware:

1. **Start**
   ↓
2. **Request Received**
   ↓
3. **Is Middleware Applied?**
   - Yes → Go to Step 4
   - No → Skip to Step 7
     ↓
4. **Execute Middleware Logic**
   ↓
5. **Call next()?**
   - Yes → Go to Step 6
   - No → Go to Step 8
     ↓

6. **Is There Another Middleware?**
    - Yes ⍈ Repeat Step 4 for the next middleware
    - No ⍈ Go to Step 7

      ⍗
7. **Route Handler Executed**

   ⍗
8. **Send Response**

   ⍗
9. **Response Sent**

   ⍗

10. End

---

12. BUILD YOUR OWN RESTFULL API:

Github Repository

---

13. HTTP HEADERS:

In the context of Node.js and Express.js, HTTP headers are key-value pairs sent between the client and the server in HTTP requests and responses. These headers contain metadata about the request or response, and in APIs, they are used for things like authentication, content negotiation, and session handling.

Understand with Mail Example:

Data
Client
From: Person A
To: Person B
Weight: 200Gms
Request
Headers

The data that is given outside the packet is called as header data that is used to send letter and where inside data is safe. You can take same example in the client request that is send to server. The request contains ip address and data type that is used to send request to server.

How to add a custom header:

You can add a custom header by using following syntax:

- *res.setHeader("HeaderName", "HeaderValue");*

Example:

```
app.get("/api/users", (req, res) => {
  res.setHeader("MyHeader","Zohaib Irshad");
  return res.json(users);
});
```

It will add new header with name MyHeader and value Zohaib Irshad.

Difference between Custom Header and Builtin Header:

1. Built-in Headers

These are standard headers defined by the HTTP protocol. They are commonly used to communicate essential information between the client and server.

Some built-in headers:

- **Content-Type**:

Specifies the media type of the resource (e.g., application/json, text/html).

- **Authorization**:

Used for authentication (e.g., bearer tokens).

- **Cache-Control**:

Directs caching mechanisms for responses.

- **Accept**:

Informs the server about the media types the client can process.

- **User-Agent**:

Contains information about the client's software.

2. Custom Headers

Custom headers are user-defined headers that allow developers to send additional information between the client and server. **Custom headers usually start with X- (e.g., X-Custom-Header), though this convention is not strictly required.**

---

14. HTTP RESPONSE STATUS CODES:

The status codes that are given by the server in a response to the client's request are called HTTP response status codes. These status codes has a specific meaning with an information.

1. Informational Responses (100–199):

Informational status codes indicate that the server has received the request and is processing it, but no final response is yet available.

- **100 Continue**: The server has received the request headers and the client should proceed to send the request body.
- **101 Switching Protocols**: The server agrees to switch to a different protocol, as requested by the client.
- **103 Early Hints**: Used to preload resources while the server prepares a final response.

2. **Successful Responses (200–299):**

These codes indicate that the request was successfully received, understood, and processed.

- **200 OK**: The request was successful, and the server returned the requested resource.
- **201 Created**: The request was successful, and a new resource was created (e.g., after a POST request).
- **202 Accepted**: The request has been accepted for processing but is not yet complete.
- **204 No Content**: The server successfully processed the request but returned no content.
- **206 Partial Content**: The server is delivering part of the resource as requested by the client (e.g., for range requests).

3. **Redirection Responses (300–399):**

These codes indicate that further action is needed to complete the request, often involving redirection to a new resource.

- **301 Moved Permanently**: The resource has been permanently moved to a new URL.
- **302 Found**: The resource is temporarily located at a different URL.
- **303 See Other**: The client should make a GET request to a different URL.
- **304 Not Modified**: Indicates that the cached version of the resource is still valid.
- **307 Temporary Redirect**: A temporary redirection where the method (GET/POST) should not change.
- **308 Permanent Redirect**: A permanent redirection where the method should not change.

4. **Client Error Responses (400–499):**

These codes indicate that the client's request contained an error.

- **400 Bad Request**: The server could not understand the request due to invalid syntax.
- **401 Unauthorized**: Authentication is required and has failed or has not been provided.
- **403 Forbidden**: The client does not have permission to access the resource.
- **404 Not Found**: The requested resource could not be found on the server.
- **405 Method Not Allowed**: The HTTP method used is not supported by the resource.
- **408 Request Timeout**: The server timed out waiting for the request.
- **429 Too Many Requests**: The client has sent too many requests in a given amount of time.

5. **Server Error Responses (500–599):**

These codes indicate that the server failed to fulfill a valid request.

- **500 Internal Server Error**: A generic error message indicating an unexpected condition.
- **501 Not Implemented**: The server does not support the functionality required to fulfill the request.
- **502 Bad Gateway**: The server, while acting as a gateway, received an invalid response from an upstream server.
- **503 Service Unavailable**: The server is not ready to handle the request, often due to maintenance or overloading.
- **504 Gateway Timeout**: The server, while acting as a gateway, did not receive a timely response from an upstream server.
- **505 HTTP Version Not Supported**: The server does not support the HTTP version used in the request.

Common Use Cases:

1. **200 OK**: When the request succeeds, such as retrieving data via a GET request.
2. **201 Created**: After creating a new resource using a POST request.

3. **404 Not Found**: When a user tries to access a nonexistent page or resource.
4. **500 Internal Server Error**: When something unexpected goes wrong on the server.
5. **301 Moved Permanently**: Redirect users to a new URL for a resource.

---

15. MODEL VIEW CONTROLLER (MVC) PATTERN IN NODEJS:

The Model-View-Controller (MVC) design pattern is a common architectural pattern used for building organized, maintainable, and scalable applications. In the context of Node.js, this pattern helps to separate concerns by structuring your code into three interconnected components:

1. Model:

The Model represents the data layer and the business logic of the application. It interacts with the database and performs operations like querying, creating, updating, or deleting data.

- **Responsibilities**:
    - Defines the structure of your data (e.g., schemas, models in MongoDB using Mongoose).
    - Handles interactions with the database.
    - Implements validation and business logic.
- **Example:**

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

const UserSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
});

const User = mongoose.model('User', UserSchema);
```

```
module.exports = User;
```

2. View

The **View** is the presentation layer. It defines how data is displayed to the user, typically in the form of HTML templates or API responses.

- **Responsibilities**:
  - Displays data to the user in a structured format (e.g., dynamic web pages or JSON responses).
  - Handles rendering logic.
- In **Node.js**, the view can be:
  - HTML templates (e.g., EJS, Pug, Handlebars).
  - API responses in JSON format (used in RESTful APIs).
- **Example:**

```
<h1>Welcome, <%= user.name %>!</h1>
<p>Your email: <%= user.email %></p>
```

3. Controller

The Controller acts as the intermediary between the Model and the View. It processes incoming requests, interacts with the model to retrieve or modify data, and sends the appropriate response to the view.

- **Responsibilities**:
  - Handles HTTP requests and routes.
  - Invokes the model for data manipulation.
  - Chooses the view to render or sends a JSON response.
- **Example:**

```
const User = require('../models/User');

const getUserProfile = async (req, res) => {
  try {
    const user = await User.findById(req.params.id);
    if (!user) {
      return res.status(404).json({ error: 'User not
found' });
    }
    res.render('profile', { user }); // Sends data to the
view
```

```
  } catch (err) {
    res.status(500).json({ error: 'Server error' });
  }
};

module.exports = { getUserProfile };
```

How MVC Works in Node.js:

1. **Request**: A user sends an HTTP request (e.g., GET /users/1).
2. **Controller**: The request is routed to a controller method that handles the logic.

   Example: getUserProfile() fetches the user data by ID.

3. **Model**: The controller interacts with the model to fetch or update data from the database.

   Example: User.findById() retrieves the user's data.

4. **View**: The controller sends the retrieved data to the view, which renders the response.

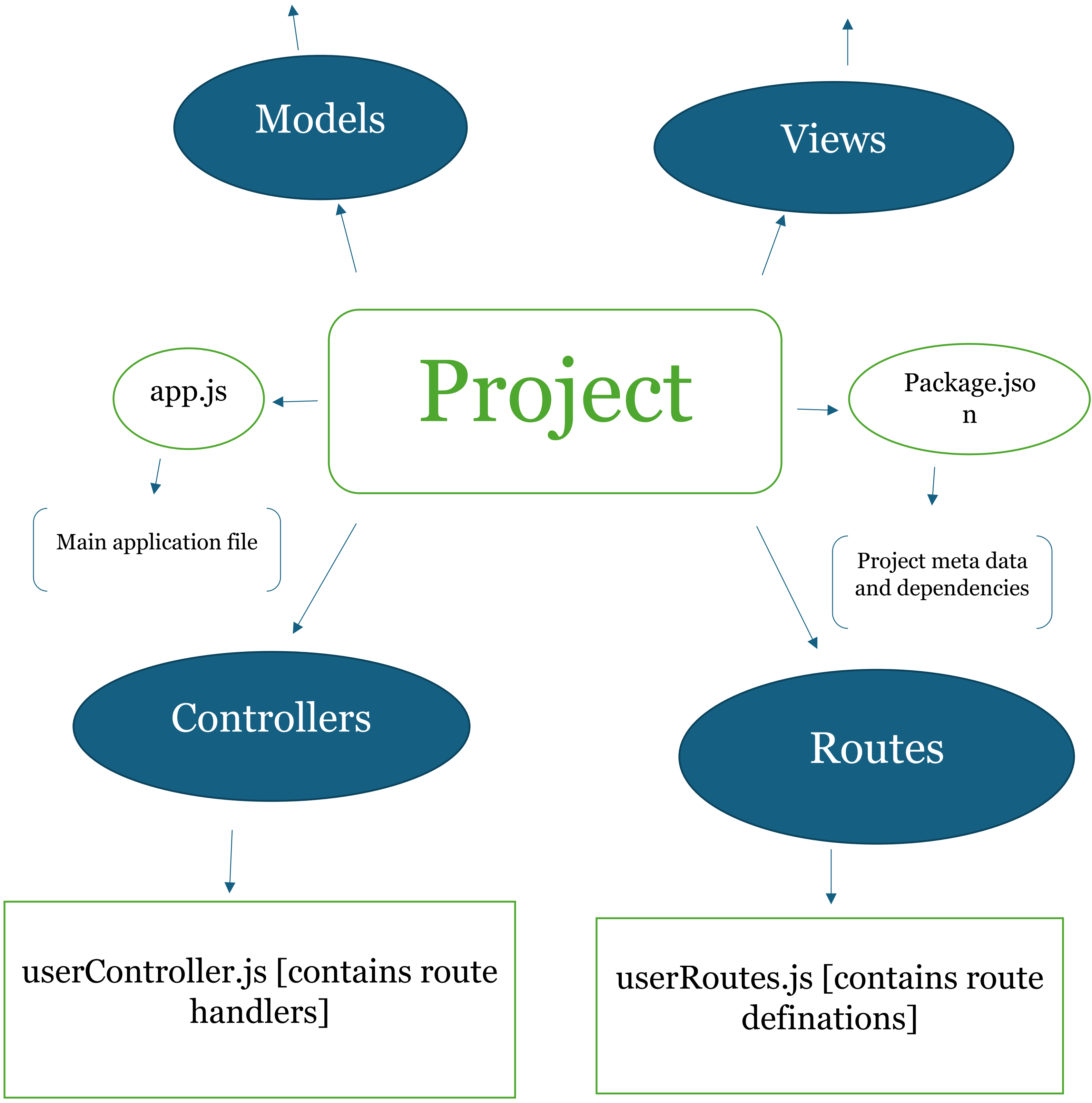
   Example: res.render('profile', { user }) generates an HTML page for the user.

Benefits of MVC in Node.js

1. **Separation of Concerns**: Keeps the logic for data (Model), business rules (Controller), and UI (View) separate.
2. **Scalability**: Simplifies scaling because components are modular and independent.
3. **Maintainability**: Code is easier to read, debug, and extend.
4. **Reusability**: Components can be reused in different parts of the application.

User.js [contains database models]

profile.ejs [contains view templates]


Models
Views
Project
app.js
Package.jso n
Main application file
Project meta data and dependencies
Controllers
Routes
userController.js [contains route handlers]
userRoutes.js [contains route definations]

Practice project:

Arrange the following project using Model View Controller pattern.

Github Repository

---

16. INTRODUCTION TO EJS:

EJS (Embedded JavaScript) is a simple templating language that lets you generate HTML markup with plain JavaScript. It's commonly used with **Node.js** applications to create dynamic web pages.

Key Features of EJS:

1. **Embed JavaScript Code in HTML**: You can use JavaScript logic directly within your HTML using <% %> syntax.

```
<ul>
  <% for(let i = 0; i < items.length; i++) { %>
    <li><%= items[i] %></li>
  <% } %>
</ul>
```

2. **Tags for Different Purposes**:

- <% %>: Executes JavaScript code but does not output anything.
- <%= %>: Outputs escaped HTML (to prevent XSS attacks).
- <%- %>: Outputs raw HTML without escaping.

3. **Reusable Templates**: You can include partials or components (like headers, footers) to avoid code repetition. Example:

```
<%- include('header') %>
<h1>Welcome</h1>
<%- include('footer') %>
```

4. **Fast and Lightweight**: EJS is designed to be lightweight and runs efficiently with Node.js.

5. **Supports Logic and Data Binding**: Pass data from your server-side code to EJS templates to render dynamic content.

Installing EJS:

You have to run following command in the terminal.

```
npm install ejs
```

Here is the practice project for you to create to understand the basics of NodeJS.

URL Shortener Project in NodeJS

---

17. AUTHENTICATION VS AUTHORIZATION:

Authentication: Verifies who you are (Login system, JWT, Sessions)
Authorization: Checks what you can do (Roles, Permissions)

Example:

- **Authentication** = Logging into a website.
- **Authorization** = Checking if the logged-in user can edit a post or delete a comment**.**

Types of Authentication in NodeJS:

a) **Stateful Authentication** (Session-based)

- Stores user login session on the server (inside memory, database, or Redis).
- Uses cookies to track logged-in users.
- Best for web applications (not ideal for APIs).

How It Works?

1. User logs in → Server creates a session (stores user info).
2. A session ID is stored in a cookie on the client.
3. On every request, the cookie is sent to the server.
4. Server checks the session and verifies the user.

Code Example:

Necessary Dependencies to be installed:

```
npm init -y
npm install express express-session bcryptjs body-parser
```

Code:

```
const express = require("express");
const session = require("express-session");
const bcrypt = require("bcryptjs");
const bodyParser = require("body-parser");

const app = express();
app.use(bodyParser.json());

// Session Configuration
app.use(session({
    secret: "superSecretKey",
    resave: false,
    saveUninitialized: true,
    cookie: { secure: false }  // Secure should be true in
production with HTTPS
}));

const users = {};  // In-memory user store (use a database
in production)

// User Registration
app.post("/register", async (req, res) => {
    const { username, password } = req.body;
    if (users[username]) return res.status(400).json({
message: "User already exists" });

    const hashedPassword = await bcrypt.hash(password, 10);
    users[username] = { password: hashedPassword };
    res.json({ message: "User registered successfully" });
});

// User Login
app.post("/login", async (req, res) => {
    const { username, password } = req.body;
    const user = users[username];

    if (!user || !(await bcrypt.compare(password,
user.password))) {
        return res.status(400).json({ message: "Invalid
credentials" });
    }

    req.session.user = username;  // Session stored
    res.json({ message: "Login successful" });
});

// Protected Route
app.get("/dashboard", (req, res) => {
    if (!req.session.user) return res.status(401).json({
message: "Unauthorized" });
    res.json({ message: `Welcome ${req.session.user}!` });
});

// Logout
app.post("/logout", (req, res) => {
    req.session.destroy();
    res.json({ message: "Logged out successfully" });
});

app.listen(3000, () => console.log("Server running on port
3000"));
```

b) Stateless Authentication (Token-based):

- **No session stored on the server** 🚀
- Uses **JSON Web Tokens (JWTs)** for authentication.
- Best for **REST APIs and mobile apps**.

How It Works?

1. User logs in → Server **generates a JWT** (token).
2. The token is **sent to the client**.
3. On every request, the client sends the token in the **Authorization header**.
4. Server **verifies the token** and allows/denies access.

Code Example:

Necessary dependencies to be installed:

```
npm init -y
npm install express jsonwebtoken bcryptjs body-parser dotenv
```

Code:

```
const express = require("express");
const jwt = require("jsonwebtoken");
const bcrypt = require("bcryptjs");
const bodyParser = require("body-parser");
require("dotenv").config();

const app = express();
app.use(bodyParser.json());

const users = {};  // Store users (use a database in
production)

// Secret Key for JWT
const SECRET_KEY = process.env.JWT_SECRET ||
"superSecretKey";

// User Registration
```

```
app.post("/register", async (req, res) => {
    const { username, password } = req.body;
    if (users[username]) return res.status(400).json({
message: "User already exists" });

    const hashedPassword = await bcrypt.hash(password, 10);
    users[username] = { password: hashedPassword };
    res.json({ message: "User registered successfully" });
});

// User Login (Generate JWT)
app.post("/login", async (req, res) => {
    const { username, password } = req.body;
    const user = users[username];

    if (!user || !(await bcrypt.compare(password,
user.password))) {
        return res.status(400).json({ message: "Invalid
credentials" });
    }

    const token = jwt.sign({ username }, SECRET_KEY, {
expiresIn: "1h" });
    res.json({ token });
});

//  Middleware to Protect Routes
function authenticateJWT(req, res, next) {
    const token = req.header("Authorization");
    if (!token) return res.status(401).json({ message:
"Access Denied" });

    jwt.verify(token, SECRET_KEY, (err, user) => {
        if (err) return res.status(403).json({ message:
"Invalid Token" });
        req.user = user;
        next();
    });
}
```

```
//  Protected Route
app.get("/dashboard", authenticateJWT, (req, res) => {
    res.json({ message: `Welcome ${req.user.username}!` });
});

app.listen(3000, () => console.log("Server running on port 3000"));
```

Quick Quizz:

Now Break this code into files using models view controller (mvc) patten.

Session vs JWT Authentication:

Feature	Statefull (session)	Stateless (JWT)
Storage	Stored in Server Memory or DB.	Stored in client side (token)
Ideal For	Web Apps	APIs and Mobile apps
Security	Secure data (stored on server)	Token can be stolen. Needs extra security.
Scalability	Less scalable	High Scalable
Logout	Server can destroy session.	Needs token blacklist.

PRACTICE PROJECT: BLOG APPLICATION

18. INTRODUCTION TO COOKIES:

Cookies:

Cookies are small pieces of data stored on the client's (browser's) side that are sent to the server with every request. They help in maintaining user sessions, storing preferences, and implementing authentication.

How cookies work in Node.js:

- Client makes a request to the server.
- Server responds with a Set-Cookie header.
- Browser stores the cookie.

- On the next request, the browser sends the cookie automatically.

Example:

```
res.cookie("userToken", "abc123", {
    maxAge: 1000 * 60 * 60,   // 1 hour expiry
    httpOnly: true,           // Prevents access via
JavaScript
    secure: true,             // Only sends over HTTPS
    sameSite: "Strict"        // Prevents CSRF attacks
});
```

Cookie Options:

Option	Description
maxAge	Expiry time in mili seconds.
httpOnly	Restricts access to the HTTP requests.
secure	Send cookies only on HTTPS
sameSite	Controlls cross-site cookie sharing (strict, lax, none)

---

19. SOCKET.IO IN NODEJS:

Socket.io is a JavaScript library that enables real-time, bidirectional, and event-based communication between web clients and servers. It is built on WebSockets but provides additional fallbacks for older browsers.

Why Use Socket.io?

- Real-time communication (Chat apps, notifications, live updates)
- Event-driven architecture
- Reliable connections (fallbacks to polling when WebSockets aren't supported)
- Easy integration with Node.js and Express

Installation:

Run following command in the terminal after initializing the NodeJS project:

```
   npm install express socket.io
```

Practice Assignment:

Index.js file

```
const http=require("http");
const path=require("path");
const { Server } = require("socket.io");

const express=require("express");
const { resolve } = require("path");
const app=express();
const server=http.createServer(app);

const io = new Server(server);

// handling Socket.io
io.on("connection", (socket) => {
    socket.on("user-message", (message) => {
      io.emit("message",message);
    });
  });

app.use(express.static(path.resolve("./public")));

app.get("/",(req,res)=>{
    return res.sendFile("/public/index.html")
})

server.listen(9000,()=>{console.log(`Server has started
listening on the port: 9000`)});
```

index.html file

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/
bootstrap.min.css" rel="stylesheet" integrity="sha384-
QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+AL
EwIH" crossorigin="anonymous">
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Chat App</title>
</head>
<body>
    <h1>Chatting:</h1>

    <div class="input-group mb-0">
        <input id="message" type="text"
placeholder="YourName: your message." class="form-control"
aria-label="Recipient's username" aria-describedby="button-
addon2">
        <button id="sendBtn" class="btn btn-outline-
secondary" type="button" id="button-addon2">Button</button>
      </div>
<div id="messages"></div>
    <script src="/socket.io/socket.io.js"></script>
<script>
  const socket = io();
  const sendBtn=document.getElementById("sendBtn");
  const messageVar=document.getElementById("message");
  const allMessages=document.getElementById("messages");
  socket.on("message",(message)=>{
    const p= document.createElement("p");
    p.innerText=message;
    allMessages.appendChild(p);

  })

  sendBtn.addEventListener("click",(e)=>{
    const message=messageVar.value;
    console.log(message);
    socket.emit("user-message",message);
  })
</script>
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bo
otstrap.bundle.min.js" integrity="sha384-
YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN7N6j
IeHz" crossorigin="anonymous"></script>
</body>
</html>
```

The project structure should be:

```
Chat-app-NodeJS/
├── public/index.html    # file that will be shown on the frontend
├── index.js        # Main server file (that will work in the backend)
├── package.json    # Project metadata and dependencies
└──node_modules       # node modules that will be installed through
terminal
```

---

20. CONCEPT OF STREAMS IN NODEJS:

Streams in Node.js are a powerful way to handle data efficiently. They allow us to process data in chunks instead of loading entire data into memory, making them ideal for handling large files or real-time data.

Types of Streams in Node.js:

There are four types of streams in Node.js:

1. **Readable Streams** → Used for reading data (e.g., fs.createReadStream()).
2. **Writable Streams** → Used for writing data (e.g., fs.createWriteStream()).
3. **Duplex Streams** → Can be both readable and writable (e.g., sockets).

4. **Transform Streams** → A special type of duplex stream that can modify data while reading and writing (e.g., zlib.createGzip() for compression).

a. Creating a readable and writable stream:

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

// Create a readable stream
const readableStream = fs.createReadStream("input.txt",
"utf8");

// Create a writable stream
const writableStream = fs.createWriteStream('output.txt');

// Pipe the readable stream into the writable stream
readableStream.pipe(writableStream);

console.log('File copied successfully using streams! ');
```

You must have the input.txt file in your current directory.

Explanation:

- fs.createReadStream('input.txt'): Reads data from input.txt in chunks.
- fs.createWriteStream('output.txt'): Writes data to output.txt.
- pipe(): Directly transfers data from the readable stream to the writable stream without buffering the entire file.

b. Using Streams for HTTP Server:

```
const http = require("http");
const fs = require("fs");

const server = http.createServer((req, res) => {
    // Read file using stream and send response
    const readableStream =
fs.createReadStream("largefile.txt");
```

```
    res.writeHead(200, { "Content-Type": "plain" });
    readableStream.pipe(res); // Pipe the stream directly to
response

});

server.listen(3000, () => {
    console.log('Server running on http://localhost:3000');
});
```

You must have “largefile.txt” with a large amount of text in your directory.

Explanation:

- Instead of loading the full file into memory, the readableStream.pipe(res) sends data chunk by chunk.
- This improves performance when handling large files.

c. Transform Stream (Gzip Compression):

```
const fs = require("fs");
const zlib = require("zlib");

// Create a read stream and a compressed write stream
const readableStream = fs.createReadStream("largefile.txt");
const gzip = zlib.createGzip();
const compressedStream =
fs.createWriteStream("largefile.txt.gz");

// Pipe data through the transform stream
readableStream.pipe(gzip).pipe(compressedStream);

console.log('File compressed successfully!');
```

You must have “largefile.txt” with a large amount of text in your directory.

Explanation:

- fs.createReadStream('input.txt') → Reads the file.
- zlib.createGzip() → Compresses the data.
- fs.createWriteStream('input.txt.gz') → Writes compressed data to a new file.

---

21. NODEJS INTERVIEW QUESTIONS:

Here's a **comprehensive list of industry-level Node.js interview questions** to help you prepare for **top companies in Pakistan** 🚀.

🔥 Basic Node.js Questions

1. What is Node.js, and how does it work?
2. How is Node.js different from JavaScript in the browser?
3. What is the difference between synchronous and asynchronous programming in Node.js?
4. Explain the Event Loop in Node.js.
5. What are callbacks, and why are they used in Node.js?
6. What is the difference between `setImmediate()`, `process.nextTick()`, and `setTimeout()`?
7. What is `package.json`, and why is it important?
8. What is npm, and how does it work?
9. How do you handle errors in Node.js?
10. What is the difference between `require` and `import` in Node.js?

🚀 Advanced Node.js Questions

11. What are streams in Node.js, and how do they work?
12. Explain the concept of middleware in Express.js.
13. What are the differences between `process.env`, `config` files, and `.env` files in Node.js?
14. How do you implement authentication in Node.js (JWT vs. Sessions)?
15. What is the difference between stateless and stateful authentication?
16. How do you prevent security vulnerabilities like SQL Injection, XSS, and CSRF in Node.js?
17. How does clustering work in Node.js?
18. What is the difference between child processes and worker threads?

19. Explain how you would optimize the performance of a Node.js application.
20. What is rate limiting, and how do you implement it in an Express.js API?
