Node JS: Beginner Guide

Description:
Node.js is an open-source, cross-platform runtime environment built on Chrome's V8 JavaScript engine. It allows developers to execute JavaScript code outside of a web browser, making it ideal for building scalable and high-performance server-side applications.

Key Topics:

- Understand the fundamentals of Node.js and its advantages.
- Be proficient in setting up a Node.js environment and using npm.
- Build and test basic Node.js applications, including an HTTP server.
- Use core modules effectively for file and server operations.
- Create and handle REST APIs.
- Debug and manage errors in Node.js applications.

Project: Build a mini-project for working with Rest Api and crud operations.

Course Outline: Node.js Basics: A Beginner's Guide to Building Fast and Scalable Applications

Course Title: Node JS Basics Beginner guide
Target Audience: Fresher and Beginners with basic knowledge of Java Script
Duration: 2 weeks (10 sessions, 2 hours each, 1 week for project work)

Module 1: Introduction to Node.js

- **Duration**: 2 hours
- **Topics Covered**:
  - What is Node.js?
  - How Node.js works (overview of the V8 engine and event loop).
  - Advantages of using Node.js.
  - Popular use cases and real-world applications.
- **Hands-On Activity**:
  - Explore existing Node.js applications for inspiration.

Module 2: Setting Up a Node.js Environment

- **Duration**: 1.5 hours
- **Topics Covered**:
  - Installing Node.js and npm on different operating systems.
  - Introduction to npm (Node Package Manager).
  - Managing dependencies with npm.
  - Understanding package.json and its role in a Node.js project.
- **Hands-On Activity**:
  - Create a basic package.json file and install a few simple npm packages (e.g., lodash).

Module 3: Creating Your First Node.js Application

- **Duration**: 2 hours
- **Topics Covered**:
  - Writing a simple "Hello World" HTTP server using Node.js.
  - Understanding the http module and server creation basics.
  - Running the server and testing with a browser or Postman.
- **Hands-On Activity**:
  - Build and run your first HTTP server.

Module 4: Understanding Event-Driven Programming

- **Duration**: 2 hours
- **Topics Covered**:
  - Introduction to event-driven programming and the Node.js event loop.
  - Exploring the events module and creating custom events.
  - Asynchronous behavior: Callbacks, Promises, and async/await.
- **Hands-On Activity**:
- Create and emit custom events in a small application

Module 5: Exploring Core Node.js Modules

- **Duration**: 3 hours
- **Topics Covered**:
  - File system operations using the fs module.
  - Working with file paths using the path module.
  - Creating and managing servers with the http module.
  - Parsing URLs using the URL module.
  - Using streams for operations
  - Using Buffer and Timer module
- **Hands-On Activity**:
  - Create a program that reads, writes, and appends data to files using the fs module
  - Use the path and url modules to parse and handle file paths dynamically.
  - Use the streams for files operations
  - Use the Buffer for handling binary data and timer module

Module 6: Exploring Third party(MySql and MongoDB) Modules for Database Connectivity

- **Duration**: 2 hours
- **Topics Covered**:
  - Database operations using the sql module.
  - CRUD Operations in MySQL:Creating and managing servers with the http module.
  - Setting Up MongoDB Database Connection in Node.js
  - CRUD Operations in MongoDB

- **Hands-On Activity**:
    - Create a program that perform CRUD operation on data using the MySql and MongoDB module.

Module 7: Introduction to REST APIs

- **Duration**: 3 hours
- **Topics Covered**:
    - What is a REST API, and why is it important?
    - Setting up routes to handle GET, POST, PUT, and DELETE requests.
    - Sending and receiving JSON data.
    - Tools for testing APIs (Postman, cURL).
- **Hands-On Activity**:
    - Build a simple REST API for a to-do list application.

Module 8: Basic Error Handling and Debugging

- **Duration**: 2 hours
- **Topics Covered**:
    - Types of errors in Node.js (syntax, runtime, and logical errors).
    - Handling errors with try...catch.
    - Using Node.js debugging tools (e.g., console.log, debugger, and Node Inspector).
    - Best practices for managing errors and logging.
- **Hands-On Activity**:
    - Add error handling to the REST API and debug common issues.

Capstone Project: Building a Basic Web Application

- **Duration**: 4 hours
- **Description**:
  Participants will build a basic web application using the skills learned throughout the course.
- **Project Requirements**:
    - Set up an HTTP server with routes.
    - Use core modules to handle file operations.
    - Create RESTful endpoints for CRUD operations.
    - Implement error handling and logging.

Module 1: Introduction to Node.js

What is Node.js?

Node.js is a **JavaScript runtime environment** that enables developers to run JavaScript code outside the web browser. It is built on the **V8 JavaScript engine**, the same engine used by Google Chrome, which makes it fast and efficient.

In simpler terms, Node.js allows you to use JavaScript for **server-side programming**, making it possible to build back-end services, APIs, and applications.

Key Features of Node.js:

- **Open-Source**: Free to use and supported by a vast community.
- **Cross-Platform**: Runs on Windows, macOS, and Linux.
- **Single-Threaded**: Uses a single thread to handle multiple requests asynchronously.

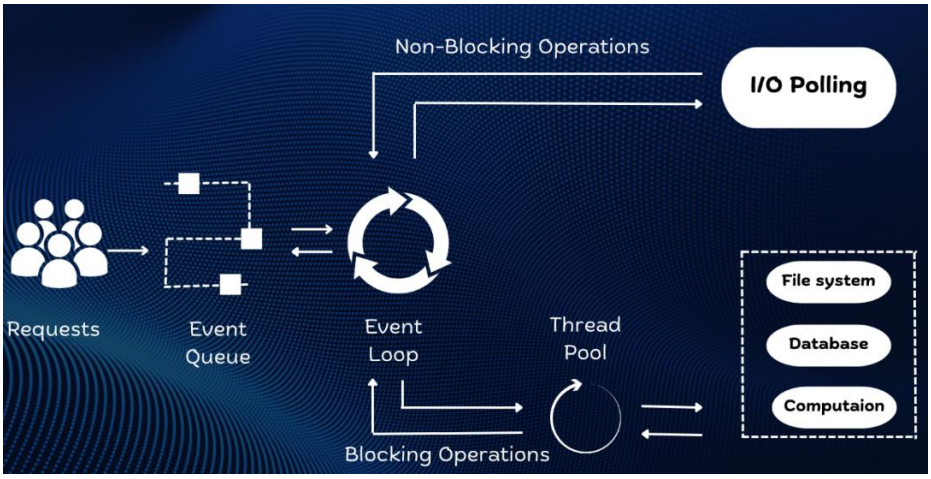
How Node.js Works

Node.js uses a unique architecture compared to traditional server-side environments like PHP or Java.

1. **V8 JavaScript Engine**:
    - Converts JavaScript code into machine code, making execution extremely fast.
2. **Event-Driven Architecture**:
    - Node.js operates on an **event loop**, which listens for events like incoming requests or completed file operations.
    - When an event occurs, Node.js processes it without waiting for other tasks to finish.
3. **Asynchronous and Non-Blocking I/O**:
    - Tasks like reading files or querying a database are done in the background.
    - Node.js does not block other tasks while waiting for these operations to complete.

Workflow Example:

- A client sends a request to fetch data.
- Node.js sends the query to the database and continues handling other requests.
- When the database responds, Node.js processes the result and sends it back to the client.



Advantages of Using Node.js

1. **High Performance**:
    - Powered by the V8 engine, Node.js processes tasks quickly.
    - Its non-blocking architecture ensures efficient handling of multiple requests.
2. **Scalability**:
    - Suitable for building scalable applications with minimal resources.
    - Supports horizontal scaling (cloning processes to handle more users).
3. **Unified Language**:
    - Developers can use JavaScript for both the front-end and back-end, simplifying development.
4. **Rich Ecosystem**:
    - Includes **npm (Node Package Manager)**, which offers thousands of reusable libraries and modules.
5. **Real-Time Applications**:
    - Perfect for applications that require instant updates, like chat apps and gaming servers.

Popular Use Cases and Real-World Applications

1. **Web Servers**:
    - Used to build lightweight, high-performance web servers for dynamic websites and APIs.
2. **Real-Time Applications**:
    - Chat apps (e.g., WhatsApp Web).
    - Online gaming (e.g., multiplayer games).
3. **Streaming Services**:
    - Netflix uses Node.js for fast and scalable data streaming.
4. **Microservices**:
    - Node.js is commonly used for building APIs and breaking applications into smaller, manageable services.
5. **Command-Line Tools**:
    - Tools like npm and webpack are built with Node.js.

Why Freshers Should Learn Node.js

- **Demand**: Node.js is widely used in industries, making it a valuable skill.

- **Simplicity**: Its JavaScript foundation makes it easy to learn if you're familiar with front-end development.
- **Versatility**: Enables you to build diverse applications, from simple APIs to complex web platforms.

Features of Node Js

Feature	Description
Non-Blocking, Asynchronous I/O	Efficient handling of I/O operations without blocking execution.
Single-Threaded Event Loop	Uses a single thread to manage concurrent operations, making it lightweight.
Fast Execution (V8 Engine)	Compiles JavaScript to machine code, improving performance.
Cross-Platform	Node.js works seamlessly on Windows, Mac, and Linux.
NPM (Node Package Manager)	Manage dependencies and install external packages from the NPM registry.
Built-In Modules	Provides core modules for tasks like HTTP requests, file system operations, etc.
Scalability (Cluster Module)	Scales across multiple CPU cores for high traffic and performance.
Real-Time Capabilities (WebSocket's)	Ideal for building real-time, interactive applications like chat and gaming.

Module 2: Setting Up a Node.js Environment

For freshers, setting up a Node.js environment is the first step toward building powerful applications. Below is a step-by-step guide to ensure you can start developing with Node.js.

---

1. Installing Node.js and npm on Different Operating Systems

What is Node.js?

Node.js is the JavaScript runtime, while **npm (Node Package Manager)** is a tool that comes with Node.js, allowing you to manage libraries, frameworks, and dependencies in your projects.

Installation Steps:

For Windows:

1. Go to the official Node.js website.
2. Download the **LTS (Long-Term Support)** version for stability.
3. Run the installer and follow the setup wizard:
    - Accept the license agreement.
    - Choose a destination folder.
    - Ensure you check the box to install **npm** along with Node.js.
4. Verify the installation:
    - Open the command prompt and type:

```
node -v
npm -v
```

    - This will display the versions of Node.js and npm installed.

2. Introduction to npm (Node Package Manager)

npm is a tool that comes with Node.js, allowing you to:

- Install libraries and frameworks for your project.
- Manage dependencies.
- Share and publish your own Node.js modules.

---

3. Managing Dependencies with npm

Dependencies are external packages your project relies on. npm helps you manage these efficiently.

NPM (Node Package Manager) is an essential tool for managing Node.js dependencies. It allows you to install, manage, and share packages (libraries or modules) that you can use in your Node.js projects. Below are some of the most commonly used NPM commands with practical examples that will help you understand how they work.

---

1. npm install (or npm i)

Purpose: Install packages or dependencies.

- **Install a package locally (in your project)**:
  - When you run npm install in a project, it installs the dependencies listed in the package.json file into the node_modules folder.

  Example:

  ```
  npm install express
  ```

  This command installs the express package into your project.

- **Install all dependencies from package.json**: If you already have a package.json file with dependencies listed, you can run:

  ```
  npm install
  ```

  This installs all the dependencies mentioned in the package.json file.

- **Install a package globally**: You can install a package globally to use it in any project:

```
npm install -g nodemon
```

This installs nodemon globally, which is a tool for auto-restarting the server during development.

- **Install dependencies to the development :**

  --save (Deprecated since npm v5)

  - Purpose: Adds a package as a dependency in the dependencies section of package.json.

  - Usage:

    ```
    npm install <package-name> --save
    ```

  - Effect:The package is recorded in the dependencies field of your package.json file. Dependencies are required for your application to run in production.

  Example:

  ```
  npm install express –save
  ```

  Result in package.json:

  ```
  "dependencies": {
  "express": "^4.18.2"
  }
  ```

  - Current Behavior: From npm v5 onward, this behavior is default. Running npm install <package-name> automatically saves the package under dependencies.

---

2. --save-dev

- Purpose: Adds a package as a development dependency in the devDependencies section of package.json.
- Usage:

```
npm install <package-name> --save-dev
```

- Effect:
  The package is recorded in the devDependencies field of package.json.
  Development dependencies are tools or packages that are only needed during development (e.g., testing frameworks, linters, or build tools).

Example:

```
npm install jest --save-dev
```

Result in package.json:

```
"devDependencies": {
 "jest": "^29.0.3"}
```

Differences Between dependencies and devDependencies:

Aspect	dependencies	devDependencies
Purpose	Needed for production.	Needed only during development.
Install Command	npm install --production includes these.	Excluded with --production.
Examples	express, mongoose, etc.	eslint, webpack, jest, etc.

Modern Usage Without Flags

- To save as a dependency (default):

```
npm install <package-name>
```

- To save as a development dependency:

```
npm install <package-name> --save-dev
```

Both flags help organize dependencies for better project management. Use devDependencies for development-specific tools and libraries.

---

2. npm uninstall (or npm remove)

Purpose: Remove a package from the project.

- **Uninstall a package locally**: If you no longer need a package in your project, you can uninstall it by running:

```
npm uninstall express
```

This will remove the express package from node_modules and update the package.json file.

- **Uninstall a package globally**: To uninstall a globally installed package:

```
npm uninstall -g nodemon
```

This will remove the global package nodemon.

---

3. npm search

Purpose: Search for a package in the NPM registry.

- **Search for a package**: If you want to find a package in the NPM registry, use the npm search command followed by the name or a keyword related to the package.

```
npm search express
```

This will return a list of packages related to "express" from the NPM registry.

---

4. npm update

Purpose: Update installed packages to the latest version.

- **Update a specific package**: You can update a specific package to its latest version by running:

```
npm update express
```

This updates the express package to its latest compatible version, according to the version specified in package.json.

- **Update all packages**: To update all packages in your project to the latest version (within the version range specified in package.json):

```
npm update
```

5. npm ls

Purpose: List installed packages and their dependencies.

- **List all installed packages**: This command shows all the packages installed in your project (including their versions):

```
npm ls
```

- **List a specific package**: To check if a specific package is installed:

```
npm ls express
```

6. npm init

Purpose: Create a package.json file for a new project.

- **Initialize a new Node.js project**: If you are starting a new Node.js project and need to create a package.json file, you can use the npm init command. This command will prompt you to answer questions about your project (name, version, description, etc.) and generate a package.json file.

```
npm init
```

- **Quick initialization**: If you want to skip the prompts and accept the default values, you can use the -y flag:

```
npm init -y
```

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

7. npm outdated

Purpose: Check for outdated packages.

- **Check for outdated packages**: This command shows which packages are outdated and their current and latest versions. It's helpful when you want to update your packages to the newest versions.

```
npm outdated
```

8.npm audit

Purpose: Check for security vulnerabilities in your dependencies.

- **Audit your project's dependencies**: Running this command helps you identify any security vulnerabilities in the packages used by your project.

```
npm audit
```

- **Fix vulnerabilities**: If there are any vulnerabilities, you can run the following command to attempt an automatic fix:

```
npm audit fix
```

9. npm version

Purpose: Manage your project's versioning.

- **Check the current version of your project**: You can use this command to check your project's current version:

```
npm version
```

- **Update the version**: To bump the version of your project, you can use the following commands:
  - To increment the **patch** version (e.g., 1.0.0 → 1.0.1):

```
npm version patch
```

  - To increment the **minor** version (e.g., 1.0.0 → 1.1.0):

```
npm version minor
```

  - To increment the **major** version (e.g., 1.0.0 → 2.0.0):

```
npm version major
```

Summary of Commands

Command	Description
npm install <package>	Installs a package locally in the project.
npm uninstall <package>	Removes a package from the project.
npm search <keyword>	Searches the NPM registry for a package.
npm update <package>	Updates a specific package to the latest version.
npm ls	Lists all installed packages in the project.

npm init	Initializes a new Node.js project and creates a package.json file.
npm outdated	Lists outdated packages and their current and latest versions.
npm audit	Checks for security vulnerabilities in the project's dependencies.
npm version <bump>	Updates the version of your project (patch, minor, major).

4. Understanding package.json and Its Role in a Node.js Project

What is package.json?

- package.json is a configuration file that stores important metadata about your project.
- It keeps track of the dependencies, project scripts, and version information.

Creating a package.json File:

1. Navigate to your project directory in the terminal.
2. Run the following command to initialize package.json:

```
npm init
```

3. Answer the prompts (name, version, description, etc.).
4. A package.json file will be created in your project folder.

Key Sections of package.json:

- **Name**
- **Version**:
  Information about your project.
- **Dependencies**:
  Lists the packages your project relies on. Example:

```
"dependencies": {
  "express": "^4.17.1"
}
```

- **Scripts**: Allows you to define commands for your project

  Example:

```
"scripts": {
"start": "node index.js",
"test": "echo 'No test specified'"
}
```

Understanding package-lock.json

- **Purpose**: It is automatically generated and ensures that the exact same versions of dependencies (and their dependencies) are installed, providing a consistent environment.
- **Key Features**:
    - Records the **exact versions** of all dependencies (both direct and nested).
    - Ensures reproducible builds by "locking" dependency versions.
    - Generated automatically by `npm` when you run `npm install` or modify dependencies.

- **Example**:

```
{
  "name": "my-app",
  "version": "1.0.0",
  "lockfileVersion": 2,
  "dependencies": {
    "express": {
      "version": "4.17.1",
      "resolved": "https://registry.npmjs.org/express/-
/express-4.17.1.tgz",
      "integrity": "sha512-abc123",
      "requires": {
        "body-parser": "^1.19.0"
      }
    },
    "body-parser": {
      "version": "1.19.0",
      "resolved": "https://registry.npmjs.org/body-parser/-
/body-parser-1.19.0.tgz",
      "integrity": "sha512-def456"
    }
  }
}
```

- **Usage**:
  - Not meant to be edited manually.
  - Ensures all developers in a team install the same dependency versions when running `npm install`.
  - Critical for projects where consistent environments are required (e.g., CI/CD pipelines).

Key Differences

Feature	package.json	package-lock.json
Purpose	Describes the project and lists its dependencies.	Locks the exact versions of dependencies.
Dependency Versions	Defines version ranges (e.g., `^1.0.0`).	Pins exact versions (e.g., `1.0.0`).
Editing	Edited manually by developers.	Auto-generated; not meant for manual editing.

Generation	Created when initializing a project with `npm init`.	Created/updated automatically by `npm install`.
Role in Teams	Defines dependencies for sharing the project.	Ensures consistent dependency versions.

Assignment 1: Installing and Managing Dependencies

Question:
You are working on a Node.js project that requires two dependencies: express for building a web server and nodemon for automatic server restarts during development.

1. Create a new directory for your Node.js project.
2. Run the appropriate npm commands to:

- Initialize a new package.json file.
- Install express and nodemon as dependencies.
- Verify that the node_modules folder and package.json reflect the installation of these packages.

Bonus:

- Add a script to package.json that uses nodemon to run your server in development mode.

Assignment 2: Understanding npm init and package.json

Question:
You need to set up a simple Node.js application for a project. Follow these steps:

1. Use the npm init command to initialize a new Node.js project.
2. Provide the following values during the initialization process:
   - Name: task-manager
   - Version: 1.0.0
   - Description: A simple task management app
   - Entry point: app.js
   - Test command: echo "Error: no test specified"
   - Git repository: Leave blank
   - Keywords: Node.js, task manager
   - Author: Your Name
   - License: MIT
3. After initializing, open the package.json file and ensure all the values you entered are correctly captured.

Assignment 3: Uninstalling npm Packages

Question:
Imagine you are cleaning up unused packages from your Node.js project.

1. Create a simple Node.js project using npm init.
2. Install at least 3 npm packages (e.g., lodash, chalk, moment).

3. Identify one package that you no longer need.
4. Uninstall that package using the appropriate npm command and ensure it is removed from both the node_modules folder and the package.json file.

Bonus:

- Use the npm ls command to check the list of installed packages after uninstalling.

Assignment 4: Updating npm Packages

Question:
You are working on a Node.js project that uses several npm dependencies. Your manager has informed you that the packages need to be updated to the latest versions.

1. Run the appropriate npm command to check for outdated packages.
2. Update all the outdated packages in your project to their latest versions using npm update.
3. After updating, use the npm ls command to verify that all packages are now up-to-date.

Bonus:

- After updating the packages, test your application to ensure everything still works correctly.

Assignment 5: Searching for npm Packages

Question:
You need to find a package that helps with file manipulation in Node.js, such as reading, writing, or appending to files.

1. Use the npm search command to search for "file system" related packages.
2. Browse the search results and choose a package that seems useful for your project.
3. Install the chosen package using npm install <package-name>.
4. Create a small script that uses the installed package to perform a basic file operation (e.g., reading from a file or writing to a new file).

Module 3: Creating Your First Node.js Application

Creating Your First Node.js Application: "Hello World" HTTP Server

This guide will help you build a simple "Hello World" HTTP server in Node.js. It's an excellent starting point for understanding server creation and testing.

Step 1: Writing a Simple "Hello World" HTTP Server

1. Create a New Project Folder:

Create a folder for your project (e.g., my-first-node-app) and navigate into it using your terminal.

```
mkdir my-first-node-app
cd my-first-node-app
```

2. **Initialize the Project:**

Run the following command to create a package.json file (optional for this basic example):

```
npm init -y
```

3. Create the Server File:

Create a new file named server.js in your project folder. This file will contain the server code.

```
// Import the HTTP module
const http = require('http');

// Create the HTTP server
const server = http.createServer((req, res) => {
  // Set the response header
  res.writeHead(200, { 'Content-Type': 'text/plain' });

  // Send the "Hello World" response
  res.end('Hello World\n');
});

// Specify the port number and start the server
const PORT = 3000;
server.listen(PORT, () => {
  console.log(`Server is running at http://localhost:${PORT}`);
});
```

Step 2: Understanding the http Module and Server Creation Basics

- **http.createServer(callback)**:
  Creates an HTTP server. The callback function is executed for every request received by the server.
  - req: Represents the incoming request.
  - res: Represents the response sent back to the client.
- **res.writeHead(statusCode, headers)**:
  Sends HTTP headers. Here, 200 is the status code for success, and Content-Type: text/plain specifies plain text content.
- **res.end(content)**:
  Ends the response and sends the specified content to the client.
- **server.listen(port, callback)**:
  Starts the server and listens on the specified port.

Step 3: Running the Server

1. **Run the Server**:

   Start the server using Node.js:

```
node server.js
```

2. **Verify the Output:**

   If everything is correct, you will see the following message in the terminal:

```
Server is running at http://localhost:3000
```

3. **Test the Server:**

   Open a browser and navigate to http://localhost:3000. You should see the text **"Hello World"** displayed.

Step 4: Testing with Postman

Postman is a tool that allows you to test APIs and servers.

1. **Install Postman**: Download and install Postman if you haven't already.
2. **Create a New Request**:
    - Select **GET** as the HTTP method.
    - Enter the URL: http://localhost:3000.
    - Click **Send** to see the response.

Hands-On Activity: Build and Run Your First HTTP Server

Objective:

- Build a basic HTTP server that responds with "Hello World".
- Test it in a browser and Postman.

Steps:

1. Follow the steps above to write and run the server.
2. Modify the response to include a custom message like "Welcome to My First Node.js App!".

```
node server.js
```

3. Restart the server after making changes:

Challenge:

- Add an additional route (e.g., /about) that displays "About Page".
- Hint: Use req.url to identify the incoming route.

Classification of Status Codes

HTTP status codes are grouped into five categories:

1. **1xx (Informational)**
    - These codes indicate that the request was received and understood, and further action is needed.
    - Examples:
        - **100 Continue**: The server has received the request headers, and the client should proceed with sending the request body.
        - **101 Switching Protocols**: The server is switching protocols as requested by the client.
2. **2xx (Success)**
    - These codes indicate that the request was successfully received, understood, and accepted.
    - Examples:
        - **200 OK**: The request was successful.
        - **201 Created**: A new resource has been created as a result of the request.
        - **204 No Content**: The server successfully processed the request but does not return any content.
3. **3xx (Redirection)**
    - These codes indicate that further action is needed to complete the request.
    - Examples:
        - **301 Moved Permanently**: The resource has been permanently moved to a new URL.
        - **302 Found**: The resource is temporarily located at a different URL.
        - **304 Not Modified**: The requested resource has not been modified since the last request.
4. **4xx (Client Errors)**
    - These codes indicate that the request contains bad syntax or cannot be fulfilled by the server.
    - Examples:
        - **400 Bad Request**: The server cannot process the request due to client error.
        - **401 Unauthorized**: Authentication is required.
        - **403 Forbidden**: The client is authenticated but does not have permission to access the resource.
        - **404 Not Found**: The requested resource could not be found.
5. **5xx (Server Errors)**
    - These codes indicate that the server failed to fulfill a valid request.
    - Examples:
        - **500 Internal Server Error**: A generic error occurred on the server.
        - **502 Bad Gateway**: The server received an invalid response from an upstream server.
        - **503 Service Unavailable**: The server is temporarily unable to handle the request.

Common Status Codes in Node.js Development

Code	Name	Description
200	OK	The request was successful.

201	Created	A new resource has been created.
204	No Content	The server processed the request, but no content is returned.
400	Bad Request	The server cannot process the request due to client error.
401	Unauthorized	Authentication is required.
403	Forbidden	The client does not have permission to access the resource.
404	Not Found	The requested resource was not found.
500	Internal Server Error	A generic server-side error occurred.
502	Bad Gateway	The server received an invalid response from an upstream server.
503	Service Unavailable	The server is temporarily unavailable.

Assignment 1: Creating and Testing a Simple HTTP GET Endpoint

Question:

1. Create a Node.js HTTP server using the `http` module that listens on port 3000.
2. Add an endpoint `/welcome` that responds with a plain text message: `"Welcome to Node.js HTTP Server!"`.
3. Use Postman to:
    - Send a GET request to `http://localhost:3000/welcome`.
    - Verify the response message and status code (should be 200).

Bonus:

- Add a header to the response: `Content-Type: text/plain`.

Assignment 2: Handling Query Parameters in HTTP Requests

Question:

1. Create an HTTP server using the `http` module that listens on port 4000.
2. Add an endpoint `/greet` that accepts a query parameter `name`. Example: A request to `http://localhost:4000/greet?name=John` should respond with `"Hello, John!"`.
3. Use Postman to test the endpoint with different values for the `name` parameter.

Bonus:

- If the `name` parameter is missing, respond with `"Hello, Guest!"`.

Assignment 3: Creating a POST Endpoint and Testing with Postman

Question:

1. Create an HTTP server using the `http` module that listens on port 5000.
2. Add a POST endpoint `/submit` that accepts JSON data. The server should:
    - Parse the JSON body sent in the request.
    - Respond with `"Data received: <data>"` where `<data>` is the JSON payload received.
3. Use Postman to:
    - Send a POST request to `http://localhost:5000/submit` with the following JSON body:

```
"name": "Alice",
"email": "alice@example.com"
}
```

    - Verify that the response contains the correct data.

Bonus:

- Add error handling for invalid or empty JSON payloads.

Assignment 4: Building an HTTP Server with Multiple Routes

Question:

1. Create an HTTP server using the `http` module that listens on port 6000.
2. Add the following routes:
    - `GET /products`: Responds with a JSON array of product names.
    - `POST /products`: Accepts a JSON payload with a new product name and adds it to the list of products.
    - `DELETE /products`: Clears all products from the list.
3. Use Postman to:
    - Test all three routes by sending appropriate requests.
    - Verify the server's behavior when data is added, retrieved, and deleted.

Bonus:

- Respond with appropriate status codes (200 for success, 400 for bad requests).

Assignment 5: Simulating a REST API with HTTP Module

Question:

1. Create an HTTP server using the `http` module that listens on port 7000.
2. Add the following RESTful endpoints for managing a to-do list:
    - `GET /todos`: Responds with the current list of to-do items in JSON format.
    - `POST /todos`: Accepts a new to-do item (as JSON) and adds it to the list.
    - `PUT /todos/:id`: Updates a to-do item by its ID.
    - `DELETE /todos/:id`: Deletes a to-do item by its ID.
3. Use Postman to:
    - Add several to-do items.
    - Retrieve all to-do items.
    - Update and delete specific items using the appropriate endpoints.

Bonus:

- Implement error handling for scenarios like missing or invalid to-do IDs.

Module 4: Understanding Event-Driven Programming

What is Event-Driven Programming?

Event-driven programming is a programming paradigm where the flow of execution is determined by events such as user actions (clicks, keypresses), messages, or changes in state.

- In Node.js, events are emitted (triggered) and listeners respond to them by executing specific code.
- This design enables **non-blocking**, asynchronous operations, making Node.js ideal for scalable applications.

The Node.js Event Loop

The **event loop** is the mechanism Node.js uses to handle asynchronous operations efficiently.

- **How it works**:
    1. Node.js processes incoming events or requests in a single-threaded environment.
    2. Tasks like I/O operations are delegated to the system (e.g., reading files or querying a database).
    3. While waiting for these tasks to complete, Node.js continues processing other requests.
    4. Once a task is complete, its callback is pushed onto the event loop for execution.
- **Key Benefits**:
    - Non-blocking I/O.
    - Handles thousands of concurrent connections efficiently.

---

2. Exploring the events Module and Creating Custom Events

The events module in Node.js allows you to work with event-driven programming by emitting and listening to custom events.

Creating and Using Custom Events

1. **Import the events module**:

```
const EventEmitter = require('events');
```

2. **Create an EventEmitter instance**:

```
const eventEmitter = new EventEmitter();
```

3. **Define an event listener**:

```
eventEmitter.on('greet', (name) => {
console.log(`Hello, ${name}!`);
});
```

4. **Emit the event**:

```
eventEmitter.emit('greet', 'Alice');
```

Output:

```
Hello, Alice!
```

Explanation:

- **on(eventName, listener)**: Registers a listener for a specific event.
- **emit(eventName, ...args)**: Triggers the event and passes arguments to the listener.

---

3. Asynchronous Behavior: Callbacks, Promises, and async/await

Node.js is built on asynchronous programming, which helps it handle tasks efficiently without blocking operations.

Callbacks

A callback is a function passed as an argument to another function, executed once the task is completed.

Example:

```
const fs = require('fs');
fs.readFile('example.txt', 'utf8', (err, data) => {
   if (err) {
console.error(err);
return;
}
console.log(data);
});
```

Promises

Promises simplify handling asynchronous operations by chaining .then() and .catch() methods.

Example:

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

fs.readFile('example.txt', 'utf8')
.then(data => console.log(data))
.catch(err => console.error(err));
```

async/await

async/await provides a cleaner way to handle asynchronous code, making it look synchronous.

Example:

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

async function readFile() {
try {
const data = await fs.readFile('example.txt', 'utf8');
console.log(data);
} catch (err) {
console.error(err);
}
}

readFile();
```

4. Hands-On Activity: Create and Emit Custom Events in a Small Application

Objective:

Create a small application that uses custom events to notify when a task is complete.

Steps:

1. **Set up the Project:**

   Create a new file, app.js, and import the events module:

```
const EventEmitter = require('events');
const eventEmitter = new EventEmitter();
```

2. **Define Event Listeners**:

```
eventEmitter.on('taskComplete', (taskName) => {
console.log(`The task "${taskName}" has been completed!`);
});
```

   Emit Events:

   Simulate task completion and trigger the event:

```
function completeTask(taskName) {
console.log(`Working on task: ${taskName}...`);
```

```
setTimeout(() => {
eventEmitter.emit('taskComplete', taskName);
}, 2000); // Simulate a delay
}

completeTask('Clean the room');
completeTask('Write Node.js code');
```

3. **Run the Application**:

   Execute the script:

```
node app.js
```

   Output:

```
Working on task: Clean the room...
Working on task: Write Node.js code...
The task "Clean the room" has been completed!
The task "Write Node.js code" has been completed!
```

Key Takeaways for Freshers

1. **Event-Driven Programming**: Focuses on handling events and executing code in response to those events.
2. **The Node.js Event Loop**: Efficiently manages asynchronous operations in a single-threaded environment.
3. **events Module**: Enables creating and emitting custom events for flexibility in application design.
4. **Asynchronous Behavior**: Master callbacks, promises, and async/await for writing clean, non-blocking code

Assignment 1: Creating and Emitting Custom Events

Question:

1. Create a new Node.js script that imports the `events` module and creates an instance of `EventEmitter`.
2. Write a custom event called `greet` that emits a message `"Hello, Welcome to Node.js Events!"`.
3. Add a listener for the `greet` event to log the message to the console.
4. Emit the `greet` event and ensure the listener executes.

Bonus:

- Modify the event to accept a parameter (e.g., `name`) and display a personalized message: `"Hello, [name]! Welcome to Node.js Events!"`.

Assignment 2: Listening to Multiple Events

Question:

1. Create an event emitter with the following events:
    - `start`: Logs `"The process has started."`.
    - `process`: Logs `"Processing data..."`.
    - `end`: Logs `"The process has ended."`.
2. Write a script that emits the events in the sequence: `start`, `process`, and `end`.

Bonus:

- Delay each event by 1 second using `setTimeout`.

Assignment 3: Handling Event Listeners Dynamically

Question:

1. Create an event emitter that listens to a `data` event.
2. Add the following listeners dynamically:
    - The first listener logs `"Data received: Starting processing."`.
    - The second listener logs `"Data received: Logging details."`.
    - The third listener logs `"Data received: Completing processing."`.
3. Emit the `data` event and observe how all listeners are executed.

Bonus:

- Remove one listener dynamically after it executes for the first time.

Assignment 4: Event Listener Count and Error Handling

Question:

1. Create an event emitter with an event named `error`.
2. Add two listeners to the `error` event. Each listener should log an appropriate error message.
3. Emit the `error` event and capture the messages.
4. Use the `emitter.listenerCount()` method to count how many listeners are attached to the `error` event.

Bonus:

- Emit an error object (e.g., `{code: 500, message: 'Internal Server Error'}`) and display its properties in the listener.

Assignment 5: Creating a Simple Event-Driven Program

Question:

1. Create an event emitter that simulates the following workflow:
    - A `userRegistered` event that logs `"User has been registered."`.
    - A `sendWelcomeEmail` event that logs `"Welcome email has been sent."`.
    - A `userLoggedIn` event that logs `"User has logged in."`.
2. Emit the events in this sequence:
    - First, `userRegistered`.
    - After 2 seconds, `sendWelcomeEmail`.
    - After 1 more second, `userLoggedIn`.

Bonus:

- Use `once()` to ensure the `sendWelcomeEmail` event is triggered only once, even if emitted multiple times.

Module 5: Exploring Core Node.js Modules

Exploring Core Node.js Modules

Node.js includes a set of built-in modules that enable developers to perform common tasks efficiently. This guide introduces the most commonly used core modules, including fs, path, http, and url, with hands-on examples for freshers.

---

1. File System Operations Using the fs Module

The fs module provides methods to interact with the file system for reading, writing, and manipulating files and directories.

Key Methods:

- **Reading a File**:
- Asynchronous Reading

```
const fs = require('fs');
fs.readFile('example1.txt', 'utf8', (err, data) => {
if (err) {
console.error(err);
return;
}
console.log(“Async Reading 1”+data);
});
});

fs.readFile('example2.txt', 'utf8', (err, data) => {
if (err) {
console.error(err);
return;
}
console.log((“Async Reading 2”+data);
});
});

fs.readFile('example3.txt', 'utf8', (err, data) => {
if (err) {
console.error(err);
return;
}
console.log((“Async Reading 3”+data);
});
});
```

- **Reading a File**:

- Synchronous Reading

```
var data = fs.readFileSync("file1.txt", "utf8")
console.log("Sync read 1" + data.toString());
var data = fs.readFileSync("file2.txt", "utf8");
console.log("Sync read 2" + data.toString());
var data = fs.readFileSync("file3.txt", "utf8");
console.log("Sync read 3" + data.toString());
var data = fs.readFileSync("file1.txt", "utf8");
console.log("Sync read 1" + data.toString());
var read = fs.readFileSync("file1.txt");
fs.writeFileSync("file4.txt", "This is my file4");
console.log("Sync write completed");
var read = fs.readFileSync("file4.txt");
console.log(read.toString());
console.log("Program end");
```

- **Writing to a File**:

```
fs.writeFile('example.txt', 'This is a new file!', (err) => {
```

```
if (err) {
console.error(err);
return;
}
console.log('File created and written successfully!');
});
```

- **Appending Data to a File**:

```
fs.appendFile('example.txt', '\nAppended content!', (err) => {
if (err) {
console.error(err);
return;
}
console.log('Data appended to file!');
});
```

- **Deleting Files**

1. **Asynchronous Deletion**:

```
fs.unlink('example.txt', (err) => {
  if (err) {
    console.error('Error deleting file:', err);
  } else {
    console.log('File deleted successfully');
  }
});
```

2. **Synchronous Deletion**:

```
try {
  fs.unlinkSync('example.txt');
  console.log('File deleted successfully');
} catch (err) {
  console.error('Error deleting file:', err);
}
```

---

e. Renaming Files

1. **Asynchronous Renaming**:

```
fs.rename('example.txt', 'renamed.txt', (err) => {
  if (err) {
    console.error('Error renaming file:', err);
  } else {
    console.log('File renamed successfully');
  }
});
```

2. **Synchronous Renaming**:

```
try {
  fs.renameSync('renamed.txt', 'example.txt');
  console.log('File renamed successfully');
} catch (err) {
  console.error('Error renaming file:', err);
}
```

---

f. Checking File Existence

- **Asynchronous Method**:

```
fs.access('example.txt', fs.constants.F_OK, (err) => {
  if (err) {
    console.log('File does not exist');
  } else {
    console.log('File exists');
  }
});
```

- **Synchronous Method**:

```
if (fs.existsSync('example.txt')) {
  console.log('File exists');
} else {
  console.log('File does not exist');
}
```

---

3. Directory Operations

Creating a Directory

1. **Asynchronous Creation**:

```
fs.mkdir('example-dir', (err) => {
  if (err) {
    console.error('Error creating directory:', err);
  } else {
    console.log('Directory created successfully');
  }
});
```

2. **Synchronous Creation**:

```
try {
  fs.mkdirSync('example-dir');
  console.log('Directory created successfully');
} catch (err) {
  console.error('Error creating directory:', err);
}
```

Reading Directory Contents

- **Asynchronous Reading**:

```
javascript
Copy code
```

```
fs.readdir('example-dir', (err, files) => {
  if (err) {
    console.error('Error reading directory:', err);
  } else {
    console.log('Directory contents:', files);
  }
});
```

- **Synchronous Reading**:

```
javascript
Copy code
try {
  const files = fs.readdirSync('example-dir');
  console.log('Directory contents:', files);
} catch (err) {
  console.error('Error reading directory:', err);
}
```

Deleting a Directory

1. **Asynchronous Deletion**:

```
fs.rmdir('example-dir', (err) => {
  if (err) {
    console.error('Error deleting directory:', err);
  } else {
    console.log('Directory deleted successfully');
  }
});
```

2. **Synchronous Deletion**:

```
try {
  fs.rmdirSync('example-dir');
  console.log('Directory deleted successfully');
} catch (err) {
  console.error('Error deleting directory:', err);
}
```

2. Working with File Paths Using the path Module

The path module is used to handle and manipulate file and directory paths.

Key Methods:

- **Getting the Base Name of a File**:

```
const path = require('path');
const filePath = '/user/home/example.txt';
console.log(path.basename(filePath)); // Output: example.txt
```

- **Getting the Directory Name**:

```
console.log(path.dirname(filePath)); // Output: /user/home
```

- **Joining Paths**:

```
const newPath = path.join('/user', 'home', 'example.txt');
console.log(newPath); // Output: /user/home/example.txt
```

3. Creating and Managing Servers with the http Module

The http module enables you to create servers to handle HTTP requests and responses.

Basic HTTP Server:

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

const server = http.createServer((req, res) => {
res.writeHead(200, { 'Content-Type': 'text/plain' });
res.end('Hello, World!');
});

const PORT = 3000;
```

```
server.listen(PORT, () => {
console.log(`Server running at http://localhost:${PORT}`);
});
```

4. Parsing URLs Using the url Module

The url module allows parsing and working with URLs.

Parsing a URL:

```
const url = require('url');
const myUrl = 'http://example.com:8080/path?name=John&age=30';

const parsedUrl = url.parse(myUrl, true);

console.log(parsedUrl.hostname); // Output: example.com
console.log(parsedUrl.port);     // Output: 8080
console.log(parsedUrl.query);    // Output: { name: 'John', age: '30' }
```

---

5. Hands-On Activity:

Objective: Create a program that combines the fs, path, and url modules to read, write, and append data to files dynamically.

Steps:

1. **Set Up the Project**: Create a file named fileHandler.js in your project folder.
2. **Write the Code**:

```
const fs = require('fs');
const path = require('path');
const url = require('url');
// Define a file path using the path module
const filePath = path.join(__dirname, 'data.txt');
// Write data to the file
fs.writeFile(filePath, 'Initial content.', (err) => {
  if (err) {
 console.error('Error writing to file:', err);
 return;
}
 console.log('File created and written successfully.');
   // Append data to the file
fs.appendFile(filePath, '\nAppended content.', (err) => {
if (err) {
 console.error('Error appending to file:', err);
  return
   }
console.log('Content appended successfully.');
 // Read the file
fs.readFile(filePath, 'utf8', (err, data) => {
 if (err) {
```

```
console.error('Error reading file:', err);
  return;
 }
   ('File Content:');
  console.log(data);
```

// Parse a URL (as an example)

```
const myUrl = 'http://example.com:8080/path?name=John&age=30';
        const parsedUrl = url.parse(myUrl, true);

        console.log('Parsed URL Hostname:', parsedUrl.hostname);
        console.log('Parsed URL Query:', parsedUrl.query);
      });
   });
});
```

3. **Run the Program**: Execute the script:

```
node fileHandler.js
```

4. **Expected Output**:

```
File created and written successfully.
Content appended successfully.
File Content:
Initial content.
Appended content.
Parsed URL Hostname: example.com
Parsed URL Query: { name: 'John', age: '30' }
```

Assignment 1: File Operations with the `fs` Module

Question:

1. Create a Node.js script that performs the following file operations using the `fs` module:
    - Create a new file named `example.txt` and write the text: `"Hello, this is a sample file!"`.
    - Append the text: `" Additional content added."` to the file.
    - Read and display the content of the file in the console.
2. Verify that the file's content is updated after the operations.

Bonus:

- Delete the file after reading its content.

Assignment 2: Working with the `path` Module

Question:

1. Write a Node.js script that demonstrates the following:
    - Resolve the absolute path of the file `example.txt` (from Assignment 1).
    - Extract and display the directory name, base name, extension, and file name of the resolved path.
    - Join two paths (e.g., `'/user'` and `'data/info.txt'`) into a single valid path.

Bonus:

- Normalize a path that contains redundant segments (e.g., `/user/docs/../data/info.txt`).

Assignment 3: Parsing and Formatting URLs with the `url` Module

Question:

1. Create a Node.js script that performs the following tasks:
    - Parse the URL: https://www.example.com:8080/products?id=123&category=books.
    - Display the protocol, hostname, port, pathname, and query parameters separately.
    - Modify the query parameter id to 456 and format the updated URL as a string.

Bonus:

- Add a new query parameter sort=asc to the URL before formatting it.

Assignment 4: Combining fs, path, and url Modules

Question:

1. Create a Node.js script that:
    - Parses the URL: http://localhost:3000/files/readme.txt.
    - Extracts the pathname (/files/readme.txt) and resolves the absolute file path using the path module.
    - Reads the content of the resolved file path using the fs module and displays it in the console.

Bonus:

- Handle errors gracefully, such as if the file does not exist.

Assignment 5: Building a File Manager with Core Modules

Question:

1. Write a Node.js script that functions as a simple file manager using the fs, path, and url modules:
    - Parse a URL to determine the file operation (read, write, or delete) and file path. Example:
        - URL: http://localhost:3000?operation=read&file=notes.txt
    - Based on the operation query parameter:
        - read: Read and display the file's content.
        - write: Write the text "Node.js is amazing!" to the specified file.
        - delete: Delete the specified file.
2. Test the file manager by creating different URLs and executing the corresponding operations.

Bonus:

- Add an append operation that appends new content to the specified file.

Understanding Node.js Stream Module: Explained for Freshers

In Node.js, the **Stream module** is a powerful way to handle reading and writing data in a non-blocking and efficient manner. Streams allow you to process large files or data sources efficiently without loading everything into memory at once. This makes Node.js an excellent choice for working with big files or real-time data like video, audio, or API responses.

Types of Streams in Node.js

1. **Readable Streams**: These are used for reading data. For example, reading from a file or receiving data over a network.
2. **Writable Streams**: These are used for writing data. For example, writing data to a file or sending data over a network.
3. **Duplex Streams**: These are both readable and writable. For example, a TCP socket.
4. **Transform Streams**: These are streams that can modify data as it's being read and written. For example, a compression stream.

Examples of Stream Usage

---

1. Reading Data with Readable Streams

A **Readable Stream** is used when you need to read data from a source like a file, or network socket.

Example: Reading Data from a File with fs.createReadStream(

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

// Create a readable stream from a file
const readableStream = fs.createReadStream('example.txt', 'utf8');

// Listen for the 'data' event to read chunks of data
readableStream.on('data', (chunk) => {
    console.log('New Chunk:', chunk);  // Outputs a chunk of the file's
content
});

// Listen for the 'end' event when the entire file is read
readableStream.on('end', () => {
    console.log('File reading completed!');
});

// Listen for the 'error' event in case there's an error
readableStream.on('error', (err) => {
    console.error('Error:', err);
});
```

Explanation:

- fs.createReadStream() creates a readable stream from a file (example.txt in this case).
- The data event is emitted every time a chunk of data is available.
- The end event is emitted when the stream has finished reading all the data.
- The error event is emitted if there's an error in the stream (e.g., the file is not found).

---

2. Writing Data with Writable Streams

A **Writable Stream** is used to send data to a destination like a file, or a network.

Example: Writing Data to a File with fs.createWriteStream()

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

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

// Write data to the file
writableStream.write('Hello, Stream!\n');
writableStream.write('This is some text being written to the file.\n');

// End the writing process
writableStream.end();

// Listen for the 'finish' event when the writing is completed
writableStream.on('finish', () => {
    console.log('Data has been written to the file successfully!');
});
```

Explanation:

- fs.createWriteStream() creates a writable stream to write data to a file (output.txt).
- The write() method is used to send chunks of data to the file.
- The end() method signals that no more data will be written.
- The finish event is triggered when the write operation is complete.

---

3. Piping Data from Readable to Writable Stream

Streams can be **piped** together, meaning that the output of one stream (Readable) is automatically fed into another stream (Writable). This makes it easy to perform data transformations or move data between different parts of your application.

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

// Create a readable stream from the source file
const readableStream = fs.createReadStream('example.txt', 'utf8');

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

// Pipe the data from the readable stream to the writable stream
readableStream.pipe(writableStream);

writableStream.on('finish', () => {
    console.log('Data has been copied successfully!');
});
```

Example: Piping Data from One File to Another

Explanation:

- pipe() is a built-in method that connects a readable stream to a writable stream.
- The data from example.txt is read and directly written to output.txt.

---

4. Transform Streams: Modifying Data

Transform Streams allow you to modify the data while it's being read and written. This is useful for tasks like compression, encryption, or converting data formats.

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

// Create a readable stream from the source file
const readableStream = fs.createReadStream('example.txt');

// Create a writable stream for the destination compressed file
const writableStream = fs.createWriteStream('example.txt.gz');

// Create a transform stream that compresses the data
const gzip = zlib.createGzip();

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

writableStream.on('finish', () => {
    console.log('File has been compressed!');
});
```

Example: Using zlib to Compress Data

Explanation:

- The zlib.createGzip() method creates a transform stream that compresses data using the GZIP format.
- The data is read from example.txt, compressed, and then written to example.txt.gz.

---

5. Handling Stream Errors

Streams are event-driven, and they can encounter errors during reading, writing, or transformation. You need to listen for the error event to handle these situations.

Example: Handling Stream Errors

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

// Create a readable stream from a non-existent file
const readableStream = fs.createReadStream('nonexistent.txt', 'utf8');

readableStream.on('data', (chunk) => {
    console.log(chunk);
});

readableStream.on('error', (err) => {
    console.error('Stream Error:', err.message);  // Handling error
gracefully
});
```

Explanation:

- The error event is emitted if there's a problem with the stream (in this case, trying to read from a non-existent file).
- The error message is logged to the console.

---

6. Piping Multiple Streams Together

You can pipe multiple streams together to create a chain of operations, like reading, transforming, and writing data.

Example: Chaining Streams to Compress and Write Data

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

// Create a readable stream from a file
const input = fs.createReadStream('largefile.txt');

// Create a writable stream for the compressed file
const output = fs.createWriteStream('largefile.txt.gz');

// Create a transform stream to compress the data
const gzip = zlib.createGzip();

// Chain the streams together using pipe
input.pipe(gzip).pipe(output);

output.on('finish', () => {
    console.log('File has been compressed and saved!');
});
```

Explanation:

- The input file largefile.txt is read, then compressed using the gzip transform stream, and finally written to the output file largefile.txt.gz.
- This is an example of chaining streams together to perform complex operations efficiently.

Summary of Key Concepts

Stream Type	Description	Example Use Case
Readable	Streams used to read data from a source (e.g., files, sockets)	Reading data from a file, receiving HTTP requests
Writable	Streams used to write data to a destination (e.g., files, sockets)	Writing data to a file, sending HTTP responses
Duplex	Streams that are both readable and writable (e.g., network connections)	Managing bi-directional communication (e.g., TCP socket)
Transform	Streams that modify data as it is being read or written	Data compression, encryption, or converting data formats

Assignment 1: Reading and Writing Files Using Streams

Question:

1. Write a Node.js script that:
    - Creates a readable stream to read the content of a file named input.txt.
    - Creates a writable stream to write the content into a new file named output.txt.
    - Pipe the readable stream into the writable stream to copy the content of input.txt to output.txt.
2. Verify that the content of output.txt matches the content of input.txt.

Bonus:

- Log a message when the data has been successfully copied.

Assignment 2: Transform Streams for Data Manipulation

Question:

1. Create a Node.js script that:
    - Reads data from a file named data.txt using a readable stream.
    - Implements a transform stream that converts all the text to uppercase.
    - Writes the transformed data to a new file named uppercase.txt.
2. Use the stream.Transform class to implement the transform stream.

Bonus:

- Modify the transform stream to reverse the text instead of converting it to uppercase.

Assignment 3: Creating a Simple HTTP Server with Streaming

Question:

1. Write a Node.js script that:
    - Creates an HTTP server that listens on port 3000.
    - When the user accesses the endpoint /file, stream the content of a file (largeFile.txt) to the response using a readable stream.
2. Test the server by accessing http://localhost:3000/file in a browser or Postman.

Bonus:

- Add error handling to handle cases where the file does not exist.

Assignment 4: Duplex Stream for Logging Input and Output

Question:

1. Create a Node.js script that:
    - Reads user input from the console using the `process.stdin` stream.
    - Implements a duplex stream that logs the input data to a file named `log.txt` and also outputs the same data to the console.
    - Write the modified data (e.g., appending `"[LOGGED]"`) to `log.txt`.

Bonus:

- Stop the input process when the user types "exit".

Assignment 5: Streaming JSON Data

Question:

1. Write a Node.js script that:
    - Streams a large JSON file (`data.json`) using a readable stream.
    - Parses each JSON object in the stream and logs it to the console.
    - Counts the total number of objects in the JSON file while streaming.
2. Use the `stream` module to handle the streaming.

Bonus:

- Write the parsed JSON objects to a new file (`parsedData.json`) using a writable stream.

Node.js Buffer Module

A **Buffer** in Node.js is a special object used to handle binary data directly. Buffers are often used when dealing with I/O operations, like reading from a file or sending data over the network.

What is a Buffer?

Buffers allow you to work with raw binary data without relying on JavaScript's native strings, which are UTF-16 encoded. Buffers are useful when you need to manipulate binary data or interact with non-text data, such as images or video files.

Creating a Buffer

A Buffer is a raw binary representation of data. You can create buffers in several ways:

```
// Create a buffer with a specific size
const buffer1 = Buffer.alloc(10);  // 10-byte buffer initialized with 0
console.log(buffer1);  // Outputs: <Buffer 00 00 00 00 00 00 00 00 00
00>

// Create a buffer from an existing array or array-like object
const buffer2 = Buffer.from([1, 2, 3, 4]);
console.log(buffer2);  // Outputs: <Buffer 01 02 03 04>

// Create a buffer from a string
const buffer3 = Buffer.from('Hello, Node.js!');
console.log(buffer3);  // Outputs: <Buffer 48 65 6c 6c 6f 2c 20 4e 6f 64
65 2e 6a 73 21>
```

Working with Buffers

- **Accessing Data**: Buffers allow you to read and write data at specific byte offsets.

```
const buffer = Buffer.from('Node.js Buffer Example');

// Reading a single byte
console.log(buffer[0]);  // Outputs: 78 (ASCII for 'N')

// Writing to a specific index
buffer[0] = 90;  // Modify first byte
console.log(buffer);  // Outputs: <Buffer 5a 6f 64 65 2e 6a 73 20 42 75
66 66 65 72 20 45 78 61 6d 70 6c 65>
```

```
const buffer = Buffer.from('Node.js Buffer Example');
const slice = buffer.slice(0, 4);  // Get first 4 bytes
console.log(slice.toString());  // Outputs: 'Node'
console.log(buffer.length);  // Outputs: 22
```

- **Slicing Buffers**: You can slice a buffer into smaller parts.
- **Buffer Length**: You can check the length of a buffer with the .length property.

Assignment 1: Creating and Manipulating Buffers

Question:

1. Write a Node.js script that:
    - Creates a buffer of size 10 and initializes it with random data.
    - Converts the buffer data into a string and logs it to the console.
    - Overwrites the first 5 bytes of the buffer with the string "Hello" and logs the updated buffer.

Bonus:

- Check the length of the buffer before and after modifications.

Assignment 2: Converting Data into Buffers

Question:

1. Create a Node.js script that:
    - Converts the string "Learning Node.js is fun!" into a buffer.
    - Logs the original buffer and its hexadecimal representation.
    - Converts the buffer back into a string and logs it.

Bonus:

- Use different encoding types (e.g., utf8, hex, base64) to encode the buffer and observe the differences.

Assignment 3: Buffer Comparison and Concatenation

Question:

1. Write a Node.js script that:
    - Creates two buffers, buffer1 containing "Hello " and buffer2 containing "World!".
    - Concatenates these two buffers into a single buffer using Buffer.concat() and logs the result.
    - Compares the two buffers using Buffer.compare() and explains the output.

Bonus:

- Modify one of the buffers and observe how the comparison result changes.

Assignment 4: Working with Buffer Slices

Question:

1. Create a Node.js script that:
    - Creates a buffer from the string "Node.js Buffer Module".
    - Extracts a slice of the buffer containing only the word "Buffer" and logs it as a string.
    - Modify the slice and check if the changes reflect in the original buffer.

Bonus:

- Create a copy of the buffer slice and modify it without affecting the original buffer.

Assignment 5: Using Buffers for File Operations

Question:

1. Write a Node.js script that:
    - Reads a file named example.txt using the fs module.
    - Converts the file content into a buffer and logs its size.
    - Writes the buffer data into a new file named copy_example.txt.

Bonus:

- Use the Buffer.from() method to create a buffer containing metadata (e.g., file size and creation date) and prepend it to the content before writing to copy_example.txt.

3. Node.js Timer Module

The **Timer module** in Node.js allows you to execute functions at specific times or after certain intervals. It's similar to how JavaScript's setTimeout() and setInterval() work in the browser but is optimized for server-side use in Node.js.

Common Timer Functions

1. **setTimeout()**: Executes a function after a specified delay (in milliseconds).

```
// Delayed execution with setTimeout()
setTimeout(() => {
    console.log('This message is shown after 2 seconds');
}, 2000);  // 2000 milliseconds (2 seconds)
```

Explanation:

- setTimeout() takes two arguments: a callback function and a time (in milliseconds).
- The callback function will be executed after the specified time elapses.

2. **setInterval()**: Executes a function repeatedly at specified intervals (in milliseconds).

```
// Repeated execution with setInterval()
let counter = 0;
const intervalId = setInterval(() => {
    counter++;
    console.log(`This message is shown every 1 second. Count:
${counter}`);

    if (counter === 5) {
        clearInterval(intervalId);  // Stops the interval after 5
repetitions
    }
}, 1000);  // 1000 milliseconds (1 second)
```

Explanation:

- setInterval() works similarly to setTimeout() but repeats the execution of the callback function at regular intervals.
- The clearInterval() function stops the repeated execution, as shown in the example.

3. **setImmediate()**: Executes a function after the current event loop cycle (as soon as possible

```
// Executes a function immediately after the current event loop
setImmediate(() => {
    console.log('This runs immediately after the current event loop');
});
```

Explanation:

- setImmediate() ensures that the function is executed after the current event loop, but before any timers (like setTimeout() or setInterval()).

Examples of Combining Buffer and Timer

Here's an example where we use both Buffers and Timers together in Node.js:

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

// Read data from a file using a buffer
const buffer = fs.readFileSync('example.txt');

// Use setTimeout to print the buffer contents after 3 seconds
setTimeout(() => {
    console.log('File content after 3 seconds:');
    console.log(buffer.toString());  // Convert the buffer to a string
and print it
}, 3000);  // Wait for 3 seconds before printing the content
```

Example: Reading a Buffer and Using setTimeout() for Delayed Execution

Explanation:

- We use fs.readFileSync() to read the content of the file into a Buffer.
- setTimeout() delays the printing of the buffer contents for 3 seconds.

Assignment 1: Using setTimeout

Question:

1. Write a Node.js script that:
    - Logs the message "This message is displayed after 3 seconds" to the console after a delay of 3 seconds using setTimeout.
    - Cancels the timer if the user types "cancel" within the 3-second window using process.stdin.

Bonus:

- Add a feature to dynamically modify the delay time using user input.

Assignment 2: Using `setInterval`

Question:

1. Create a Node.js script that:
    - Logs the current time to the console every 2 seconds using `setInterval`.
    - Stops the interval after 10 seconds.

Bonus:

- Modify the script to accept the interval duration as a command-line argument.

Assignment 3: Using `setImmediate`

Question:

1. Write a Node.js script that:
    - Logs `"Executing setImmediate callback"` to the console using `setImmediate`.
    - Logs `"Synchronous code executed"` before the `setImmediate` callback to demonstrate the asynchronous behavior of `setImmediate`.

Bonus:

- Add a comparison to show the execution order of `setImmediate`, `setTimeout(0)`, and `process.nextTick`.

Assignment 4: Scheduling and Clearing Timers

Question:

1. Create a Node.js script that:
    - Schedules a timer using `setTimeout` to log `"This will not execute"` after 5 seconds.
    - Cancels the timer immediately using `clearTimeout` and logs `"Timer has been cleared"`.

Bonus:

- Demonstrate a similar behavior with `setInterval` and `clearInterval`.

Assignment 5: Timer Module in a Real-World Use Case

Question:

1. Write a Node.js script that:
    - Simulates a countdown timer using setInterval that counts down from 10 to 0.
    - Logs "Happy New Year!" when the countdown reaches 0 and stops the interval.

Bonus:

- Implement a pause and resume feature for the countdown timer using process.stdin to capture user input.

Summary of Key Concepts

Feature	Description	Example Use Case
Buffer	Handles binary data efficiently in memory.	Storing file data or processing binary information.
setTimeout()	Executes a function once after a specified delay.	Delaying actions like logging or file operations.
setInterval()	Repeatedly executes a function at specified intervals.	Running periodic tasks like monitoring or updates.
setImmediate()	Executes a function after the current event loop cycle.	Executes a task as soon as the event loop is free.

Key Takeaways for Freshers:

- **fs Module**:
    - Perform file operations like reading, writing, and appending.
- **Stream Module:**
    - Perform stream operations like reading,writing,piping data in chunks
- **path Module**:
    - Handle file paths dynamically and resolve path issues.
- **http Module**:
    - Create and manage basic HTTP servers.
- **url Module**:
    - Parse URLs and extract useful components like query parameters.
- **Buffer Module:**
    - Buffers allow you to efficiently manage raw data, especially in I/O operations
- **Timer Module:**
    - Timers enable you to schedule tasks for delayed or repeated execution.

Module 6: Exploring Third party(MySql and MongoDB) Modules for Database connectivity

Node.js Database Connectivity: A Beginner's Guide

Node.js can connect to various types of databases such as MySQL, MongoDB, PostgreSQL, and SQLite. In this guide, we will focus on explaining **database connectivity** in Node.js, specifically with **MySQL** and **MongoDB**, as these are the most commonly used databases. You'll learn how to connect to databases, perform basic CRUD operations (Create, Read, Update, Delete), and understand the essential concepts involved in database connectivity.

---

1. Setting Up MySQL Database Connection in Node.js

MySQL is a popular relational database that stores data in tables, allowing for complex queries. To connect to a MySQL database from Node.js, you typically use the **mysql2** or **mysql** package.

Steps to Set Up MySQL Database Connection in Node.js

1. **Install MySQL Database Client:** First, you need to install the mysql2 package to communicate with MySQL databases in Node.js.

   bash
   Copy code
   npm install mysql2

2. **Create a Database and Table:** For this example, let's assume we have a database named testdb and a table named users.

```
CREATE DATABASE testdb;
USE testdb;

CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(100)
);
```

3. **Connecting to MySQL Database:** Now, you can write a simple Node.js script to connect to the database and perform operations.

```
const mysql = require('mysql2');

// Create a connection to the database
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',  // Replace with your MySQL username
  password: '',  // Replace with your MySQL password
  database: 'testdb'
});

// Connect to the MySQL server
connection.connect(err => {
  if (err) {
    console.error('error connecting: ' + err.stack);
    return;
  }
  console.log('connected as id ' + connection.threadId);
});
```

4. **CRUD Operations in MySQL:**
   - **Create Data** (Insert):

```
const query = 'INSERT INTO users (name, email) VALUES (?,
?)';
connection.query(query, ['John Doe', 'john@example.com'],
(err, results) => {
  if (err) throw err;
  console.log('User added with ID:', results.insertId);
});
```

   - **Read Data** (Select):

```
connection.query('SELECT * FROM users', (err, results) => {
  if (err) throw err;
  console.log(results);  // Display all users
});
```

   - **Update Data**:

```
const query = 'UPDATE users SET name = ? WHERE id = ?';
connection.query(query, ['Jane Doe', 1], (err, results) => {
  if (err) throw err;
  console.log('Updated rows:', results.affectedRows);
});
```

   - **Delete Data**:

```
const query = 'DELETE FROM users WHERE id = ?';
connection.query(query, [1], (err, results) => {
  if (err) throw err;
  console.log('Deleted rows:', results.affectedRows);
});
```

5. **Close the Connection:** After finishing your operations, you should always close the connection.

```
connection.end();
```

2. Setting Up MongoDB Database Connection in Node.js

MongoDB is a NoSQL database that stores data in JSON-like documents. Unlike relational databases, MongoDB is schema-less and offers high flexibility, which is great for handling unstructured data.

Steps to Set Up MongoDB Database Connection in Node.js

1. **Install MongoDB Client (Mongoose):** To connect to MongoDB, we use the **Mongoose** library, which provides a simple interface to interact with MongoDB.

```
npm install mongoose
```

2. **Connect to MongoDB:** Mongoose makes it easy to connect to a MongoDB database using a simple connection URI.

```
const mongoose = require('mongoose');
  //Replace with your MongoDB URI
mongoose.connect('mongodb://localhost:27017/testdb', {
useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log('Connected to MongoDB!'))
  .catch(err => console.log('Error connecting to MongoDB:', err));
```

3. **Define a Model:** In MongoDB, documents are stored in collections. You define the

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

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

schema (structure) of the documents using Mongoose models.

4. **CRUD Operations in MongoDB:**
   - **Create Data** (Insert):

```
const newUser = new User({ name: 'John Doe', email:
'john@example.com' });

newUser.save()
  .then(user => console.log('User created:', user))
  .catch(err => console.log('Error creating user:', err));
```

```
User.find({}, (err, users) => {
  if (err) throw err;
  console.log('Users:', users);
});
```

- **Read Data** (Find):
- **Update Data** (Update):

```
User.findByIdAndUpdate('userId', { name: 'Jane Doe' })
  .then(result => console.log('Updated User:', result))
  .catch(err => console.log('Error updating user:', err));
```

- **Delete Data** (Delete):

```
User.findByIdAndDelete('userId')
  .then(result => console.log('Deleted User:', result))
  .catch(err => console.log('Error deleting user:', err));
```

```
mongoose.connection.close();
```

5. **Close the Connection:** When you are done, close the connection to MongoDB.

3. Advantages of Using Different Databases in Node.js

- **MySQL**:
  - Structured data storage using tables with relationships.
  - Suitable for applications requiring complex queries and transactions.
  - Uses SQL for querying, which is familiar to many developers.
- **MongoDB**:
  - Flexible and schema-less structure allows easy scaling and quick development.
  - Great for handling large amounts of unstructured or semi-structured data.
  - Uses JSON-like documents, which integrates well with JavaScript applications.

---

4. Best Practices for Database Connectivity in Node.js

- **Error Handling**: Always handle errors properly to ensure that your application doesn't crash. Use try-catch for synchronous operations and catch blocks for promises.
- **Use Environment Variables**: Never hardcode sensitive information like database credentials. Use environment variables to securely store them. Libraries like dotenv can be helpful.
- **Use Prepared Statements**: In SQL, avoid directly inserting values into queries to prevent **SQL injection** attacks. Use parameterized queries instead.
- **Close Connections**: Always close the database connection when done to avoid memory leaks or connection issues.

Assignment 1: Connecting to MySQL Database

Question:

1. Write a Node.js script that:
    - Connects to a MySQL database using the `mysql` module.
    - Logs a success message (`"Connected to MySQL!"`) upon successful connection.
    - Handles errors if the connection fails.

Bonus:

- Modify the script to create a database named `student_records` if it doesn't already exist.

Assignment 2: CRUD Operations in MySQL

Question:

1. Write a Node.js script that performs the following:
    - Creates a table named `students` with columns `id`, `name`, and `age`.
    - Inserts a few records into the `students` table.
    - Retrieves and logs all records from the `students` table.
    - Updates the `age` of a specific student by `id`.
    - Deletes a student record by `id`.

Bonus:

- Add functionality to fetch the records sorted by the `name` column.

Assignment 3: Connecting to MongoDB

Question:

1. Write a Node.js script that:
    - Connects to a MongoDB database named `school` using the `mongodb` or `mongoose` module.
    - Logs a success message (`"Connected to MongoDB!"`) upon successful connection.
    - Handles connection errors.

Bonus:

- Add functionality to close the database connection after logging the success message.

Assignment 4: CRUD Operations in MongoDB

Question:

1. Write a Node.js script that:

- Creates a collection named `students`.
- Inserts multiple documents into the `students` collection with fields `name`, `age`, and `grade`.
- Retrieves all documents from the collection and logs them.
- Updates the `grade` of a specific student by their `name`.
- Deletes a document based on the student's `name`.

Bonus:

- Add pagination to retrieve the documents in batches of 2.

Assignment 5: Real-World Use Case - A Unified API for MySQL and MongoDB

Question:

1. Write a Node.js REST API that:
   - Provides an endpoint `/add-student` to add a student to both MySQL and MongoDB databases.
   - Provides an endpoint `/get-students` to fetch all students from both MySQL and MongoDB and display them together in the API response.
   - Handles errors gracefully if either database operation fails.

Bonus:

- Add an endpoint `/delete-student/:id` to delete a student by `id` from both MySQL and MongoDB.

Module 7: Introduction to REST APIs

Introduction to REST APIs for Freshers

In modern web development, **APIs** (Application Programming Interfaces) are essential for enabling communication between different systems and services. A **REST API**

(Representational State Transfer) is a widely used type of web service that allows systems to interact over HTTP using standard methods like GET, POST, PUT, and DELETE.

In this guide, you will learn the basics of REST APIs, how to set up routes for different HTTP methods, send/receive JSON data, and test your API using tools like Postman and cURL. You will also complete a hands-on activity by building a simple REST API for a to-do list application.

---

1. What is a REST API and Why is it Important?

A **REST API** is an architectural style for designing networked applications. REST APIs use HTTP requests to perform operations on resources (data objects) and return responses in a format like JSON or XML. These APIs are stateless, meaning each request from a client must contain all the information necessary to process the request.

Key Concepts:

- **Resources**: A resource in a REST API represents data or functionality, such as a user, a product, or a to-do item.
- **HTTP Methods**:
  - **GET**: Retrieve data (e.g., get a list of items).
  - **POST**: Create new data (e.g., add a new item).
  - **PUT**: Update existing data (e.g., edit an item).
  - **DELETE**: Remove data (e.g., delete an item).

Why REST APIs are Important:

- They provide a standardized way for different applications to communicate with each other.
- REST APIs are simple, lightweight, and easy to integrate with various technologies (e.g., mobile apps, web apps, third-party services).

---

2. Setting Up Routes to Handle GET, POST, PUT, and DELETE Requests

Steps to Set Up Routes:

1. **Install Express**: First, make sure you have Node.js installed. Then, initialize a new project and install Express:

```
npm init -y
npm install express
```

2. **Create Routes**: Here's an example of setting up GET, POST, PUT, and DELETE routes for managing to-do list items:

```
const express = require('express');
const app = express();
app.use(express.json()); // Middleware to parse JSON bodies

let todos = []; // In-memory array to store todos

// GET route: Fetch all to-do items
app.get('/todos', (req, res) => {
    res.status(200).json(todos);
});

// POST route: Create a new to-do item
app.post('/todos', (req, res) => {
    const { title, description } = req.body;
    const newTodo = { id: Date.now(), title, description };
    todos.push(newTodo);
    res.status(201).json(newTodo);
});

// PUT route: Update a to-do item by ID
app.put('/todos/:id', (req, res) => {
    const { id } = req.params;
    const { title, description } = req.body;
    let todo = todos.find(todo => todo.id == id);
    if (todo) {
        todo.title = title;
        todo.description = description;
        res.status(200).json(todo);
    } else {
        res.status(404).json({ message: 'Todo not found' });
    }
});

// DELETE route: Delete a to-do item by ID
app.delete('/todos/:id', (req, res) => {
    const { id } = req.params;
    todos = todos.filter(todo => todo.id != id);
    res.status(204).end(); // No content in the response body
});

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

In this example:

- **GET** /todos: Returns all to-do items.
- **POST** /todos: Adds a new to-do item.
- **PUT** /todos/:id: Updates a specific to-do item.
- **DELETE** /todos/:id: Deletes a specific to-do item.

3. Sending and Receiving JSON Data

REST APIs often use JSON (JavaScript Object Notation) as the format for sending and receiving data. In the example above, the body of the POST and PUT requests contains JSON data, and the responses are also in JSON format.

Example JSON Data:

- **Request Body (POST/PUT)**:

```
{
  "title": "Learn Node.js",
  "description": "Understand REST APIs and Node.js fundamentals."
}
```

```
[
  {
    "id": 1,
    "title": "Learn Node.js",
    "description": "Understand REST APIs and Node.js fundamentals."
  }]
```

- **Response Body (GET)**:

In Express, you can use express.json() middleware to automatically parse incoming JSON data in request bodies.

---

4. Tools for Testing APIs

To test your REST API, you can use tools like **Postman** or **cURL** to send HTTP requests and view the responses.

Testing with Postman:

1. **GET Request**:
    - Open Postman and select **GET** as the HTTP method.
    - Enter http://localhost:3000/todos as the URL.
    - Click **Send** to see the list of to-do items.
2. **POST Request**:
    - Select **POST** as the HTTP method.

    - Enter http://localhost:3000/todos as the URL.
    - In the Body tab, select **raw** and choose **JSON**.
    - Enter the JSON data for the new to-do item and click **Send**.
3. **PUT Request**:
    - Select **PUT** as the HTTP method.
    - Enter http://localhost:3000/todos/1 (replace 1 with the ID of the to-do you want to update).
    - Provide the updated JSON data in the body and click **Send**.
4. **DELETE Request**:
    - Select **DELETE** as the HTTP method.
    - Enter http://localhost:3000/todos/1 (replace 1 with the ID of the to-do you want to delete).
    - Click **Send** to delete the to-do item.

curl -X POST http://localhost:3000/todos -H "Content-Type: application/json" -d '{"title": "Learn Node.js", "description": "Understand REST APIs and Node.js fundamentals."}'

---

5. Hands-On Activity: Build a Simple REST API for a To-Do List Application

Objective:

Build a basic REST API that allows users to create, view, update, and delete to-do items.

Steps:

1. **Set up your environment**:
    - Install **Node.js** and **Express**.
    - Create a new project directory and initialize a Node.js project using npm init -y.
    - Install **Express** with npm install express.
2. **Create the API**:
    - Create a file named app.js and set up the routes for **GET**, **POST**, **PUT**, and **DELETE** as shown above.
3. **Test your API**:
    - Use **Postman** or **cURL** to interact with the API and ensure it's working.

Assignment 1: Connecting to MySQL Database

Question:

1. Write a Node.js script that:
    - Connects to a MySQL database using the `mysql` module.
    - Logs a success message (`"Connected to MySQL!"`) upon successful connection.
    - Handles errors if the connection fails.

Bonus:

- Modify the script to create a database named `student_records` if it doesn't already exist.

Assignment 2: CRUD Operations in MySQL

Question:

1. Write a Node.js script that performs the following:
    - Creates a table named `students` with columns `id`, `name`, and `age`.
    - Inserts a few records into the `students` table.
    - Retrieves and logs all records from the `students` table.
    - Updates the `age` of a specific student by `id`.
    - Deletes a student record by `id`.

Bonus:

- Add functionality to fetch the records sorted by the `name` column.

Assignment 3: Connecting to MongoDB

Question:

1. Write a Node.js script that:
    - Connects to a MongoDB database named `school` using the `mongodb` or `mongoose` module.
    - Logs a success message (`"Connected to MongoDB!"`) upon successful connection.
    - Handles connection errors.

Bonus:

- Add functionality to close the database connection after logging the success message.

Assignment 4: CRUD Operations in MongoDB

Question:

1. Write a Node.js script that:
    - Creates a collection named `students`.
    - Inserts multiple documents into the `students` collection with fields `name`, `age`, and `grade`.
    - Retrieves all documents from the collection and logs them.
    - Updates the `grade` of a specific student by their `name`.
    - Deletes a document based on the student's `name`.

Bonus:

- Add pagination to retrieve the documents in batches of 2.

Assignment 5: Real-World Use Case - A Unified API for MySQL and MongoDB

Question:

1. Write a Node.js REST API that:
    - Provides an endpoint `/add-student` to add a student to both MySQL and MongoDB databases.
    - Provides an endpoint `/get-students` to fetch all students from both MySQL and MongoDB and display them together in the API response.
    - Handles errors gracefully if either database operation fails.

Bonus:

- Add an endpoint `/delete-student/:id` to delete a student by `id` from both MySQL and MongoDB.

Key Takeaways for Freshers:

- **REST API Basics**: Understand the HTTP methods (GET, POST, PUT, DELETE) and how they correspond to CRUD operations.
- **Setting Up Routes**: Learn how to set up routes using Express to handle these HTTP methods.
- **JSON**: Practice sending and receiving JSON data, which is a common format in REST APIs.
- **Testing**: Use tools like Postman and cURL to test your API.

Module 8: Basic Error Handling and Debugging

Basic Error Handling and Debugging in Node.js for Freshers

When developing applications in Node.js, it's essential to handle errors and debug issues effectively to ensure your application runs smoothly. This guide will introduce you to common types of errors in Node.js, how to handle them, and tools for debugging. By the end of this guide, you'll be able to manage errors in your code and debug common issues, making your application more robust and reliable.

---

1. Types of Errors in Node.js

There are three main types of errors you will encounter in Node.js:

a. Syntax Errors:

- Occur when there's a mistake in the structure of your code.
- These are typically caught by the JavaScript engine at the time of code compilation.
- Example:

```
let message = 'Hello, World!'; // Missing closing quote here will cause a syntax error
```

b. Runtime Errors:

- These occur during the execution of the program, often due to unforeseen issues like accessing undefined variables, trying to call methods on null, or dividing by zero.
- Example:

```
let result = 10 / 0; // This causes Infinity, which might not be handled correctly in your code
```

c. Logical Errors:

- These are errors in the logic of your program, causing the program to behave incorrectly but without throwing an error.
- Logical errors are the hardest to spot because they don't necessarily result in any immediate crashes or exceptions.
- Example:

```
let a = 5;
```

```
let b = 10;
console.log(a + b);  // Expecting multiplication, but it's just addition
```

2. Handling Errors with try...catch

In JavaScript (and Node.js), the try...catch block is used to handle errors gracefully and prevent your program from crashing.

Syntax:

```
try {
// Code that may cause an error
} catch (error) {
// Code to handle the error
console.error('An error occurred:', error);
}
```

Example:

```
function divideNumbers(a, b) {
try {
if (b === 0) {
throw new Error('Cannot divide by zero');
}
return a / b;
} catch (error) {
console.error(error.message);  // Handle the error by logging it
}
}

divideNumbers(10, 0);  // Output: Cannot divide by zero
```

In this example, if b is 0, an error is thrown, and the catch block handles the error, preventing the program from crashing.

Handling Asynchronous Errors:

Node.js heavily relies on asynchronous operations (e.g., file system operations, HTTP requests). In such cases, you can use try...catch within async functions combined with await:

```
async function fetchData() {
try {
let data = await fetch('https://api.example.com/data');
let json = await data.json();
console.log(json);
} catch (error) {
console.error('Error fetching data:', error);
}
}
```

3. Using Node.js Debugging Tools

Node.js offers a variety of tools for debugging code and understanding what's going wrong in your application.

a. console.log:

- The simplest and most commonly used debugging tool.
- Allows you to log values at various points in your application.

Example:

```
let x = 10;
console.log(x);  // Output: 10
```

- You can log objects, arrays, or any other data type to understand the state of variables.

b. debugger:

- You can use the debugger keyword to pause your code and inspect the program's state.
- This is useful for stepping through code to understand how it's executing.

Example:

```
function add(a, b) {
   debugger; // Code execution will pause here
   return a + b;
}
add(3, 4);
```

- To run this, start your Node.js program with the inspect flag: node --inspect app.js.
- Open Chrome and navigate to chrome://inspect to interact with the debugger.

c. Node Inspector:

- A more advanced tool that allows you to step through your code line by line, set breakpoints, and inspect variables.
- To use Node Inspector, run:

```
node --inspect-brk app.js
```

Then open Chrome and go to chrome://inspect to start debugging.

---

4. Best Practices for Managing Errors and Logging

a. Error Handling Best Practices:

- **Always handle errors**: Use try...catch for synchronous code and handle asynchronous errors with .catch() or async/await with try...catch.
- **Use meaningful error messages**: Include enough detail to understand what caused the error (e.g., the variable values, function name).
- **Don't expose stack traces to users**: In production, stack traces should be hidden, as they can give away sensitive details about the app. Log them instead.

b. Logging Best Practices:

- **Use a logging library** like winston or morgan for structured logging in production.
- **Log errors with severity levels** (e.g., info, warn, error).
- **Log to a file or an external service** for better tracking and analysis.

Example using winston:

```
npm install Winston

const winston = require('winston');

const logger = winston.createLogger({
level: 'info',
transports: [
new winston.transports.Console(),
new winston.transports.File({ filename: 'app.log' })
]
});

logger.info('This is an info log');
logger.error('This is an error log');
```

c. Use process.exit() for Critical Errors:

If your app encounters a critical error that it cannot recover from, you can exit the application gracefully using process.exit(1) to indicate an error.

Example:

```
try {
// Code that might cause a fatal error
throw new Error('Critical failure');
} catch (error) {
console.error(error);
process.exit(1);  // Exit with a non-zero status code
}
```

5. Hands-On Activity: Add Error Handling to the REST API and Debug Common Issues

Objective:

Enhance the to-do list REST API (from the previous activity) by adding proper error handling and debugging.

Steps:

1. **Add Error Handling to Routes**: Update your routes to handle errors gracefully. For instance, in the POST route, check if the required fields are provided:

```
app.post('/todos', (req, res) => {
try {
const { title, description } = req.body;
if (!title || !description) {
throw new Error('Title and description are required');
}
const newTodo = { id: Date.now(), title, description };
todos.push(newTodo);
res.status(201).json(newTodo);
} catch (error) {
res.status(400).json({ message: error.message });
}
});
```

2. **Add Debugging Statements**: Insert console.log() or use debugger to inspect variables and identify issues.

Example:

```
app.get('/todos', (req, res) => {
console.log('Fetching all todos');
console.log('Current todos:', todos);
res.status(200).json(todos);
});
```

3. **Test Your API**: Use **Postman** or **cURL** to test the routes and make sure error handling works. For example:
    - Try sending a POST request without the title or description fields and ensure the error is handled correctly.
    - Use **console logs** to check the flow of the application and troubleshoot issues.

Assignment 1: Using `try...catch` for Synchronous Error Handling

Question:

1. Write a Node.js script that:

- Reads the content of a file named `data.txt` using the `fs.readFileSync` method.
- Handles the error using a `try...catch` block if the file does not exist.
- Logs either the file content or an error message like `"File not found."`.

Bonus:

- Add a feature to create the file with default content if it doesn't exist.

Assignment 2: Handling Errors in Asynchronous Operations

Question:

1. Write a Node.js script that:
   - Reads the content of a file named `data.json` using the `fs.readFile` method (asynchronous).
   - Handles the error using a callback function if the file does not exist or is invalid.
   - Logs `"Error reading file: <error message>"` if there's an error and logs the file content if successful.

Bonus:

- Validate whether the file contains valid JSON data and log an error if it doesn't.

Assignment 3: Debugging with `console.log` and `debugger`

Question:

1. Create a Node.js script that:
   - Simulates a function to calculate the factorial of a number.
   - Intentionally introduces a logical error (e.g., wrong calculation in the loop).
   - Uses `console.log` to print the value of variables inside the loop for debugging.
   - Runs the script with `node inspect` and uses the `debugger` statement to pause execution and inspect variables.

Bonus:

- Log the stack trace using `console.trace` when an error occurs.

Assignment 4: Centralized Error Handling

Question:

1. Write a Node.js script that:
    - Creates a custom function `readFileSafely(filePath)` that:
        - Reads a file using `fs.promises.readFile`.
        - Returns the file content if successful or throws an error if the file doesn't exist.
    - Use `try...catch` to call `readFileSafely` and log appropriate messages for success or failure.

Bonus:

- Add logging to write the errors into a log file named `error.log`.

Assignment 5: Error Handling in an HTTP Server

Question:

1. Create a Node.js HTTP server that:
    - Listens on port 3000.
    - Responds with the content of a file requested by the URL query parameter (e.g., `/file?name=data.txt`).
    - Handles errors when:
        - The query parameter is missing.
        - The file does not exist or cannot be read.
    - Sends appropriate HTTP status codes (e.g., 400 for missing parameters, 404 for file not found).

Bonus:

- Add a custom error page for the client when an error occurs.

Key Takeaways for Freshers:

- **Error Types**: Learn the difference between syntax, runtime, and logical errors.
- **Error Handling**: Use try...catch to handle both synchronous and asynchronous errors.
- **Debugging Tools**: Use console.log(), debugger, and Node Inspector to step through your code and identify problems.
- **Best Practices**: Ensure robust error handling and logging in your applications to keep them reliable and easy to maintain.

By adding error handling and debugging capabilities to your application, you ensure that your code is more resilient, easier to maintain, and less prone to crashing in production. Happy coding!

Project Work: Building a Basic Web Application

In this final project, you will apply everything you've learned about Node.js to build a simple web application. This project will involve setting up an HTTP server, handling file operations, creating RESTful endpoints for CRUD (Create, Read, Update, Delete) operations, and implementing error handling and logging to make the app robust and reliable.

Project Overview

The goal of this project is to create a basic to-do list application that allows users to:

1. **Create** new to-do items.
2. **Read** a list of all to-do items.
3. **Update** an existing to-do item.
4. **Delete** a to-do item.

You will also need to ensure proper error handling and logging throughout the application.

---

1. Set Up the HTTP Server with Routes

Start by setting up a basic HTTP server that listens for incoming requests and serves the necessary responses.

Steps:

1. **Create the server.js file**:
    - This file will contain the HTTP server logic.
2. **Set up routes**:
    - The server will listen for requests and respond to different routes (for the CRUD operations).

Code Example:

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

// Create an HTTP server
const server = http.createServer((req, res) => {
    const parsedUrl = url.parse(req.url, true);
    const path = parsedUrl.pathname;
    const method = req.method;

    // Handling routes
    if (path === '/todos' && method === 'GET') {
        // Handle GET request to fetch all todos
        fs.readFile('./todos.json', 'utf8', (err, data) => {
            if (err) {
                res.statusCode = 500;
                return res.end(JSON.stringify({ message: 'Unable to read todos' }));
            }
            res.setHeader('Content-Type', 'application/json');
            res.end(data);
        });
    } else if (path === '/todos' && method === 'POST') {
        // Handle POST request to create a new todo
        let body = '';
        req.on('data', chunk => {
            body += chunk;
        });
        req.on('end', () => {
            const newTodo = JSON.parse(body);
            fs.readFile('./todos.json', 'utf8', (err, data) => {
                if (err) {
                    res.statusCode = 500;
                    return res.end(JSON.stringify({ message: 'Unable to read todos' }));
                }
                const todos = JSON.parse(data);
                todos.push(newTodo);
                fs.writeFile('./todos.json', JSON.stringify(todos, null, 2), err => {
                    if (err) {
                        res.statusCode = 500;
                        return res.end(JSON.stringify({ message: 'Unable to save todo' }));
                    }
                    res.statusCode = 201;
                    res.end(JSON.stringify(newTodo));
                });
            });
        });
    } else {
        // Handle undefined routes
        res.statusCode = 404;
        res.end(JSON.stringify({ message: 'Route not found' }));
    }
});

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

- The server listens for two routes:
    - GET /todos: Returns a list of all to-dos.
    - POST /todos: Adds a new to-do.

2. Use Core Modules to Handle File Operations

Node.js provides built-in modules like fs (File System) to handle file operations such as reading, writing, and appending data to files.

File Operations in this Project:

- **Reading Data**: Use fs.readFile() to load existing to-dos from a todos.json file.
- **Writing Data**: Use fs.writeFile() to save the updated list of to-dos after adding a new one.

Example:

```
fs.readFile('./todos.json', 'utf8', (err, data) => {
if (err) {
console.error('Error reading file:', err);
return;
}
console.log('Data read:', data);
});
```

3. Create RESTful Endpoints for CRUD Operations

A RESTful API allows you to perform operations (CRUD) over HTTP requests. You will create four main operations for this project:

- **Create** (POST): Add a new to-do item.
- **Read** (GET): Fetch all to-do items.
- **Update** (PUT): Modify an existing to-do item.
- **Delete** (DELETE): Remove a to-do item.

For simplicity, you will implement just GET and POST for now, but you can extend this by adding PUT and DELETE later.

Example for the GET method (to fetch all to-dos):

```
if (path === '/todos' && method === 'GET') {
fs.readFile('./todos.json', 'utf8', (err, data) => {
if (err) {
res.statusCode = 500;
return res.end(JSON.stringify({ message: 'Unable to read todos' }));
}
res.setHeader('Content-Type', 'application/json');
```

```
res.end(data);
});
}
```

4. Implement Error Handling and Logging

Proper error handling is essential to make sure your application doesn't crash unexpectedly. You will also want to log important events and errors for debugging and monitoring purposes.

Error Handling:

- Use try...catch blocks for synchronous code.
- Handle asynchronous errors inside callbacks (like fs.readFile()).

Logging:

- Use console.log() for simple logging.
- You can enhance logging by using a logging library like winston.

Example of error handling:

```
fs.readFile('./todos.json', 'utf8', (err, data) => {
if (err) {
console.error('Error reading todos file:', err);
res.statusCode = 500;
return res.end(JSON.stringify({ message: 'Unable to read todos' }));
}
res.setHeader('Content-Type', 'application/json');
res.end(data);
});
```

Example of using winston for logging:

npm install Winston

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

// Create a logger
const logger = winston.createLogger({
    level: 'info',
    transports: [
        new winston.transports.Console(),
        new winston.transports.File({ filename: 'app.log' })
    ]
});

// Example of logging an error
logger.error('Something went wrong!');
```

Steps to Complete the Project:

1. **Create the Server**:
    - Set up an HTTP server that listens for GET and POST requests.
2. **Create the To-Do List Data File**:
    - Create a todos.json file to store the to-do items. This file will be read and written to by the server.
3. **Set Up CRUD Operations**:
    - Implement the GET and POST routes to fetch and create to-do items, respectively.
4. **Implement Error Handling**:
    - Handle file read/write errors, invalid requests, and missing data.
5. **Add Logging**:
    - Use console.log() or a logging library like winston to log errors and events.

Final Project Output:

- A basic Node.js application that can:
    - Accept requests to add and retrieve to-do items.
    - Handle file system operations using fs for reading and writing the to-do list.
    - Provide proper error responses and log information for debugging.

By the end of this project, you will have a good understanding of how to use Node.js for building simple web applications, how to work with core modules, and how to implement RESTful APIs with proper error handling.