

## *Unit: -IV*

# ***Node JS: Node Js - Basics and Setup, Node Js Console, Node Js Command Utilities, Node Js Modules, Node Js Concepts, Node Js Events, Node Js with Express Js, Node Js Database Access.***

## **❖ Node JS: Node Js - Basics and Setup**

### **✓ Node Js - Basics and Setup**

It is not P.L

It is not a framework

It is not a library.

It is a basically runtime environment which includes everything to run/execute a JS program.

Node **Js** can connect with databases.

Code and syntax are very similar to Java Script.

Node Js use Chrome's V8 to execute code.

### **✓ Node Js Setup**

#### **Step 1: Install Node.js**

1. Visit the official Node.js website at <https://nodejs.org>.
2. On the website's homepage, you'll find two versions to choose from:
  - LTS (Long Term Support): This version is recommended for most users as it is stable and receives updates for an extended period.
  - Current: The current version contains the latest features and improvements, but it may not be as stable as the LTS version.
3. Choose the version you prefer and download the installer for your operating system (Windows, macOS, or Linux).
4. Run the installer and follow the installation instructions. The installation process will also include NPM (Node Package Manager), which is used to manage Node.js packages and dependencies.
5. To check if Node.js and NPM are installed successfully, open your terminal or command prompt and run the following commands:

```
node -v
```

```
npm -v
```

You should see the versions of Node.js and NPM printed to the console.

#### **Step 2: Writing Your First Node.js Program**

1. Create a new directory for your Node.js project and navigate to it using your terminal or command prompt.
2. Create a new JavaScript file, for example, "hello.js," and open it in a code editor of your choice.
3. In "hello.js," write a simple Node.js program. For example:

```
console.log("Hello, Node.js!");
```

4. Save the file.

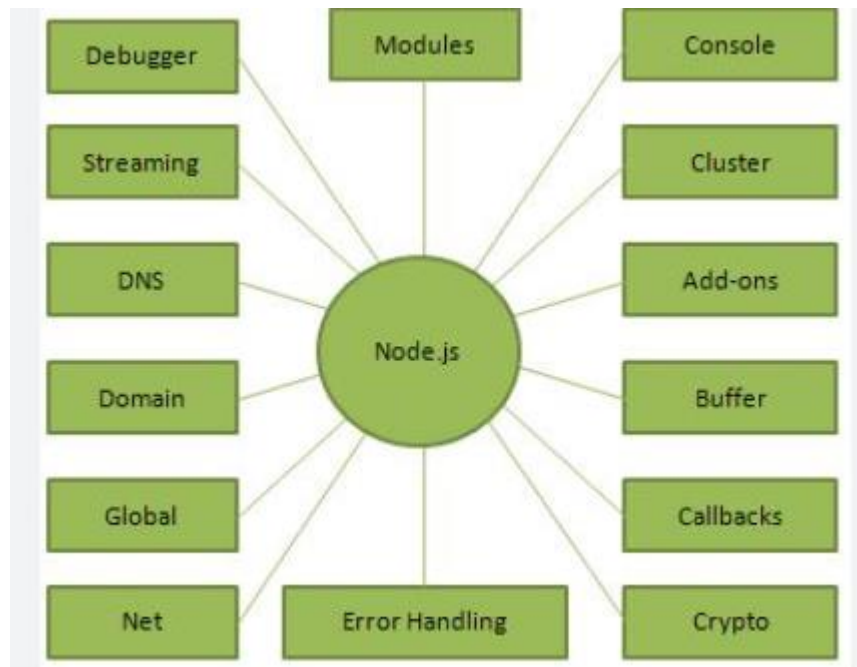
#### **Step 3: Running Your Node.js Program**

1. In the terminal, navigate to the directory where you saved your "hello.js" file.
2. Run your Node.js program using the following command:

node hello.js

3. You should see the output "Hello, Node.js!" printed to the console.

## ❖ Node Js Concepts



### ✓ Differences between jQuery, JavaScript & Node.js.

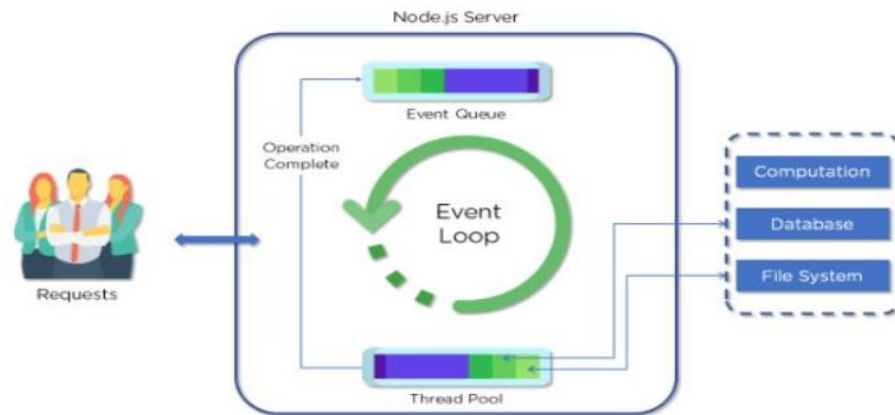
	JavaScript	jQuery	Node.js
<b>Type</b>	JavaScript is a general-purpose programming language used for both client-side and server-side development. It's not limited to web development and can be used in various contexts.	jQuery is a JavaScript library used for front-end web development. It's not a separate language or runtime environment but an additional set of functions and methods built on top of JavaScript.	Node.js is a runtime environment that allows you to execute JavaScript code on the server side. It's not a language, library, or framework; it's a runtime environment.
<b>Purpose</b>	JavaScript can be used both on the client side (in web browsers) and on the server side (using technologies like Node.js).	jQuery simplifies and enhances client-side web development by providing a wide range of features, such as DOM manipulation, event handling, and AJAX functionality.	Node.js is used for building server-side applications, such as web servers, APIs, and backend services. It allows developers to write server-side code in JavaScript.

<b>Usage</b>	JavaScript is used for creating interactive and dynamic web applications in web browsers. It's also used for server-side programming to handle tasks like server-side (By using Node JS technology) logic, data processing, and database interactions.	jQuery is used to create interactive and dynamic user interfaces, perform asynchronous requests (AJAX), and simplify many common front-end tasks.	Node.js code runs on the server, handling tasks like serving web pages, processing data, and communicating with databases.
<b>Event Handling</b>	JavaScript is used for handling events in the browser, making web pages interactive and responsive to user actions.	jQuery focuses on making it easier to select and manipulate HTML elements in the Document Object Model (DOM).	Node.js code runs on the server, handling tasks like serving web pages, processing data, and communicating with databases.

#### ❖ Node Js Architecture

##### ✓ Node.js Server Architecture

Node.js uses the “Single Threaded Event Loop” architecture to handle multiple concurrent clients. Node.js Processing Model is based on the JavaScript event-based model along with the JavaScript callback mechanism.



#### ❖ <https://www.simplilearn.com/understanding-node-js-architecture-article>

Parts of the Node.js Architecture:

**Requests:** -Incoming requests can be blocking (complex) or non-blocking (simple), depending upon the tasks that a user wants to perform in a web application

**Node.js Server:** -Node.js server is a server-side platform that takes requests from users, processes those requests, and returns responses to the corresponding users

**Event Queue:** -Event Queue in a Node.js server stores incoming client requests and passes those requests one-by-one into the Event Loop

**Thread Pool:** -Thread pool consists of all the threads available for carrying out some tasks that might be required to fulfill client requests

**Event Loop:** -Event Loop indefinitely receives requests and processes them, and then returns the responses to corresponding clients

**External Resources:** -External resources are required to deal with blocking client requests. These resources can be for computation, data storage, etc.

### ✓ **FEATURES OF Node Js**

1. **Extremely fast:** Node.js is built on Google Chrome's V8 JavaScript Engine, so its library is very fast in code execution.
2. **I/O is Asynchronous and Event Driven:** All APIs of Node.js library are asynchronous i.e. non-blocking. So a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call. It is also a reason that it is very fast.
3. **Single threaded:** Node.js follows a single threaded model with event looping.
4. **Highly Scalable:** Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way.
5. **No buffering:** Node.js cuts down the overall processing time while uploading audio and video files. Node.js applications never buffer any data. These applications simply output the data in chunks.
6. **Open source:** Node.js has an open source community which has produced many excellent modules to add additional capabilities to Node.js applications.
7. **License:** Node.js is released under the MIT license.

### ❖ **Node Js Console**

#### ✓ **What is Node.js Console?**

The Node.js Console module provides a way to interact with the system console, allowing developers to log information, warnings, errors, and other messages during the execution of a Node.js application. It offers various methods to output different types of data and provides basic debugging capabilities.

#### ✓ **Why Use Node.js Console?**

##### 1. **Logging Information:**

- The primary purpose of the console is to log information about the application's execution. This helps developers understand what is happening at different points in the code.

##### 2. **Debugging:**

- Developers use the console for debugging by outputting variable values, checking the flow of the program, and identifying errors.
3. **Error Handling:**
    - The console is crucial for logging errors and exceptions, making it easier to diagnose and fix issues in the code.
  4. **Performance Monitoring:** Measure execution times to identify bottlenecks.

**There are mainly three main console methods that are used to write any node.js stream:**

1. `console.log ()`
2. `console. Error ()`
3. `console. Warn ()`

The Node.js runtime includes a built-in module called "console" that provides methods for printing messages and debugging information to the console, which is typically the command line or terminal window where your Node.js application is running. It allows developers to log, debug, and monitor their Node.js applications by providing various methods like `console.log ()`, `console. Error ()`, and `console. Debug ()` for different logging and debugging purposes.

1. **`console.log([data][, ...args])`:** This method is used for general-purpose logging. It prints messages to the console, and you can pass one or more values as arguments. It is commonly used for debugging and providing information about the execution of your program.

**Example: -**

```
console.log ("Hello, Node.js!");
console.log ("Value of x:", x);
```

2. **`console. Error([data][, ...args])`:** Similar to **`console.log()`**, but it logs messages as errors. This method is typically used to report errors and problems in your application. The output is often highlighted or formatted differently to indicate an error.

**Example: -**

```
console. Error ("An error occurred!");
console. Error ("Error details:", error);
```

3. **`console. Warn([data][, ...args])`:** Outputs warning messages to the console. Use this method when you want to indicate potential issues or non-critical problems in your code. Warnings often have a distinct format or color.

Some More Important Node.js console methods: -

4. **`console.info([data][, ...args])`:** Logs informational messages to the console. This is useful for providing non-error-related information. It is often used for general status updates and informative messages.
5. **`console. Debug([data][, ...args])`:** This method is used for debugging purposes. It's similar to **`console.log()`**, but it's intended for developer-specific debugging output. You can use it to print debugging information during development, and it can be turned off in production environments.
6. **`console. time(label)`:** Starts a timer with a given label. You can use **`console. Time End(label)`** to stop the timer and measure the time it takes to execute a specific block of code. It's valuable for profiling and performance optimization.

7. **console. Trace([message]):** This method prints a stack trace to the console, showing the call hierarchy and execution path of your code. It's useful for debugging and understanding the flow of your program, especially when you want to trace how a particular function was called.

## ❖ Node Js Command Utilities

- Node.js command utilities are command-line tools that come with the Node.js ecosystem to perform various tasks related to package management, process management, testing, and code quality enforcement. They enhance the development workflow and provide essential functionalities for developers working with Node.js applications
- Node.js command utilities are command-line tools and features provided by Node.js itself and its ecosystem. These tools are designed to help developers perform a wide range of tasks related to JavaScript and Node.js development. They include the Node.js interpreter, package manager (NPM), the Read-Eval-Print Loop (REPL), debuggers, profilers, and more.

Node.js command utilities are useful for a variety of purposes, including:

- Running JavaScript code on the server-side using the Node.js interpreter.
- Managing project dependencies and scripts with NPM.
- Interactively experimenting with JavaScript code in the REPL.
- Debugging Node.js applications to find and fix issues.
- Profiling and optimizing the performance of Node.js code.
- Executing project-specific command-line tools provided by Node.js packages.

Node.js command utilities can be used in different contexts, including:

- Local development on your computer to run and test Node.js applications.
- In continuous integration and deployment pipelines to manage project dependencies.
- Debugging and profiling applications during development and troubleshooting.
- Server environments where Node.js applications are deployed.
- Script automation for various tasks, such as building, testing, or deployment.

**How to Use Node.js, Where to Use Command Utilities:** Here's a brief overview of how to use some common Node.js command utilities:

### 1. npm (Node Package Manager):

**Where to Use:** Package management: npm is used to install, manage, and share JavaScript packages (libraries and tools).

**How to Use:**

- Install a package: **npm install <package>**
- Initialize a new Node.js project: **npm init**
- Start the application defined in the "scripts" section: **npm start**

#### Why to Use:

- Simplifies dependency management and allows easy sharing of code with other developers.

#### 2. node:

- Execute JavaScript code outside of a web browser using the Node.js runtime.
- Execute a script: **node myScript.js**
- Allows running JavaScript code on the server-side, enabling the development of server applications.

#### 3. npx:

- Execute npm package binaries without installing them globally.
- Run a package binary: **npx <package>**
- Provides a way to use tools without installing them globally, reducing potential conflicts.

#### 4. nodemon:

- Development workflow to automatically restart the server on file changes.
- Run a script with nodemon: **nodemon myServer.js**
- Saves time by avoiding manual server restarts during development.

#### 5. npm-run-all:

- Run multiple npm scripts sequentially or in parallel.
- Run scripts concurrently: **npm-run-all script1 script2**
- Streamlines complex build or development workflows.

#### 6. pm2:

- Process management for Node.js applications in production.
- Start a Node.js application: **pm2 start myApp.js**
- Ensures continuous uptime, automatic restarts, and load balancing in production environments.

#### 7. eslint:

- Code quality enforcement and identifying issues in JavaScript code.
- Lint a file: **eslint myFile.js**
- Helps maintain a consistent and high-quality codebase.

#### 8. ava:

- Test runner for Node.js applications.
- Run tests: **ava test.js**
- Facilitates writing and running test cases, ensuring code reliability.

#### 9. forever:

- Ensure that a given script runs continuously as a persistent process.
- Start a script with forever: **forever start myScript.js**
- Useful for running scripts persistently, especially in server environments.

#### Node.js Command Line Options

Index	Option	Description
-------	--------	-------------

1.	v, --version	It is used to print node's version.
2.	-h, --help	It is used to print node command line options.
3.	-e, --eval "script"	It evaluates the following argument as JavaScript. The modules which are predefined in the REPL can also be used in script.
4.	-p, --print "script"	It is identical to -e but prints the result.
5.	-c, --check	Syntax check the script without executing.
6.	-i, --interactive	It opens the REPL even if stdin does not appear to be a terminal.
7.	-r, --require module	It is used to preload the specified module at startup. It follows require()'s module resolution rules. Module may be either a path to a file, or a node module name.
8.	--no-deprecation	Silence deprecation warnings.
9.	--trace-deprecation	It is used to print stack traces for deprecations.
10.	--throw-deprecation	It throws errors for deprecations.
11.	--no-warnings	It silence all process warnings (including deprecations).
12.	--trace-warnings	It prints stack traces for process warnings (including deprecations).
13.	--trace-sync-io	It prints a stack trace whenever synchronous i/o is detected after the first turn of the event loop.
14.	--zero-fill-buffers	Automatically zero-fills all newly allocated buffer and slowbuffer instances.
15.	--track-heap-objects	It tracks heap object allocations for heap snapshots.
16.	--prof-process	It processes V8 profiler output generated using the v8 option --prof.
17.	--V8-options	It prints V8 command line options.



18.	--tls-cipher-list=list	It specifies an alternative default tls cipher list. (requires node.js to be built with crypto support. (default))
19.	--enable-fips	It enables fips-compliant crypto at startup. (requires node.js to be built with ./configure --openssl-fips)
20.	--force-fips	It forces fips-compliant crypto on startup. (cannot be disabled from script code.) (same requirements as --enable-fips)
21.	--icu-data-dir=file	It specifies ICU data load path. (Overrides node_icu_data)

Examples: -

To see the version of the running Node:

Open Node.js command prompt and run command `node -v` or `node --version`

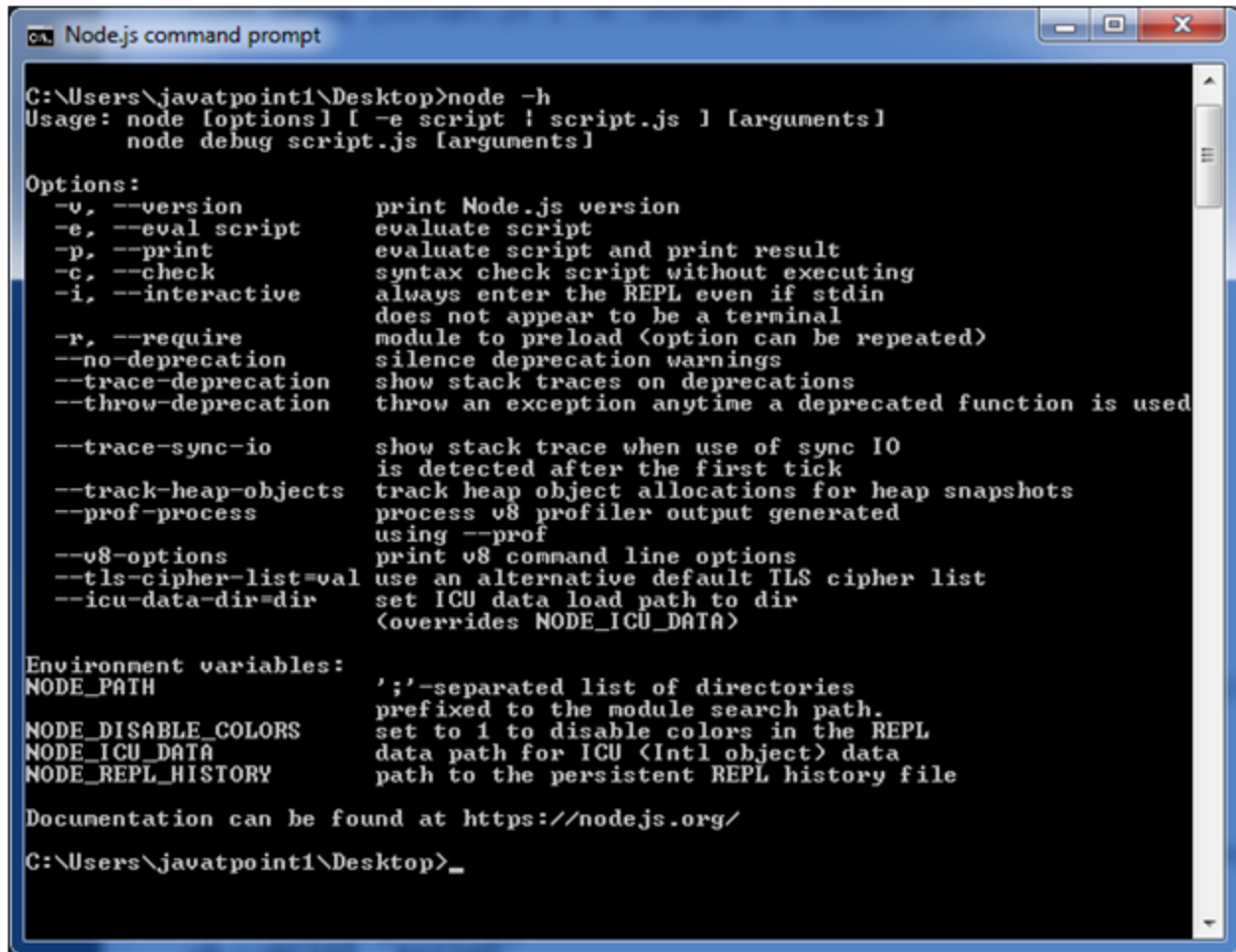
```

C:\Users\javatpoint1>cd desktop
C:\Users\javatpoint1\Desktop>node -v
v4.4.2
C:\Users\javatpoint1\Desktop>node --version
v4.4.2
C:\Users\javatpoint1\Desktop>

```

For Help:

Use command `node? h` or `node --help`



```
Node.js command prompt

C:\Users\javatpoint1\Desktop>node -h
Usage: node [options] [ -e script | script.js ] [arguments]
       node debug script.js [arguments]

Options:
  -v, --version           print Node.js version
  -e, --eval script       evaluate script
  -p, --print             evaluate script and print result
  -c, --check             syntax check script without executing
  -i, --interactive       always enter the REPL even if stdin
                          does not appear to be a terminal
  -r, --require           module to preload (option can be repeated)
  --no-deprecation        silence deprecation warnings
  --trace-deprecation     show stack traces on deprecations
  --throw-deprecation     throw an exception anytime a deprecated function is used

  --trace-sync-io         show stack trace when use of sync IO
                          is detected after the first tick
  --track-heap-objects    track heap object allocations for heap snapshots
  --prof-process          process v8 profiler output generated
                          using --prof
  --v8-options            print v8 command line options
  --tls-cipher-list=val   use an alternative default TLS cipher list
  --icu-data-dir=dir      set ICU data load path to dir
                          <overrides NODE_ICU_DATA>

Environment variables:
NODE_PATH                ';' separated list of directories
                          prefixed to the module search path.
NODE_DISABLE_COLORS      set to 1 to disable colors in the REPL
NODE_ICU_DATA            data path for ICU <Intl object> data
NODE_REPL_HISTORY        path to the persistent REPL history file

Documentation can be found at https://nodejs.org/

C:\Users\javatpoint1\Desktop>_
```

To evaluate an argument (but not print result):

Use command `node -e, --eval "script"`

To evaluate an argument and print result also:

Use command `node -p "script"`

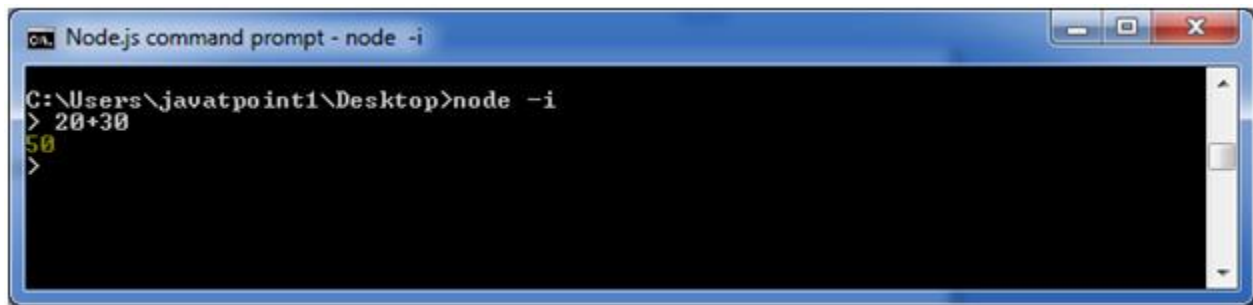


```
Select Node.js command prompt

C:\Users\javatpoint1\Desktop>node -e 10+10
20
C:\Users\javatpoint1\Desktop>node -p 10+10
20
C:\Users\javatpoint1\Desktop>_
```

To open REPL even if stdin doesn't appear:

Use command node -i, or node --interactive

A screenshot of a Windows command prompt window titled "Node.js command prompt - node -i". The window shows the following text:

```
C:\Users\javatpoint1\Desktop>node -i
> 20+30
50
>
```

## ❖ Node Modules

### ➤ Node JS Modules

- In Node.js, modules are a fundamental concept that allows you to organize code into reusable and maintainable units.
- Modules can be a single file or a collection of multiple files/folders.
- Programmers are heavily reliant on modules is because of their reusability as well as the ability to break down a complex piece of code into manageable chunks.
- The Common JS pattern, using **exports** and **require**, is used for defining and importing modules.

#### ✓ **Basic Structure of a Module:**

A Node.js module typically consists of the following elements:

#### 1. **Exports:**

- Objects, functions, or variables that are made available to other modules.
- Defined using **module. Exports** or **exports**.

#### 2. **Require:**

- Allows a module to include and use functionality from another module.
- Achieved using the **require** function.

#### ✓ **Modules are of three types:**

#### ✓ Core Modules

#### ✓ local Modules

#### ✓ Third-party Modules

#### ✓ **Core Modules**

- Core modules in Node.js are built-in modules provided by the Node.js runtime itself.
- These modules offer essential functionalities for common tasks, such as working with the file system, handling HTTP requests, managing paths, and more.
- Core modules are part of the Node.js standard library and are available for use without requiring installation via npm.
- To use a core module, you simply need to use the **require** function.
- **Syntax:** `const module = require('module_name');`

Here are some examples of core modules in Node.js:

### 1. fs (File System):

- Provides methods for interacting with the file system.

- **Example:**

```
const fs = require('fs'); // Reading a file fs.readFile('file.txt', 'utf8', (err, data) => { if (err)
throw err; console.log(data); });
```

### 2. http:

- Allows you to create an HTTP server and make HTTP requests.

- **Example:**

```
const http = require('http'); // Creating a simple HTTP server const server =
http.createServer((req, res) => { res.writeHead(200, { 'Content-Type': 'text/plain' });
res.end('Hello, World!'); }); server.listen(3000, () => { console.log('Server listening on port
3000'); });
```

### 3. path:

- Provides utilities for working with file and directory paths.

```
const path = require('path'); const fullPath = path.join(__dirname, 'files', 'example.txt');
console.log(fullPath);
```

### 4. os:

- Offers operating system-related utility methods.

- **Example:**

```
const os = require('os'); console.log('Platform:', os.platform()); console.log('Architecture:',
os.arch()); console.log('Total Memory:', os.totalmem());
```

### 5. events:

- Provides an event-driven programming interface.

- **Example:**

```
const EventEmitter = require('events'); class MyEmitter extends EventEmitter { } const
myEmitter = new MyEmitter(); myEmitter.on('event', () => { console.log('Event
occurred!'); }); myEmitter.emit('event');
```

### 6. util:

- Provides various utility functions.

- **Example:**

```
const util = require('util'); const myFunction = util.promisify(someAsyncFunction);
```

### 7. querystring:

- Provides methods for working with URL query strings.

- **Example:**

```
const querystring = require('querystring'); const params = { name: 'John', age: 30 }; const
queryString = querystring.stringify(params); console.log(queryString);
```

### 8. crypto:

- Implements cryptographic functionality.

- **Example:**

```
const crypto = require('crypto'); const hash = crypto.createHash('sha256');
hash.update('Hello, Node.js!'); const hashedData = hash.digest('hex');
console.log(hashedData);
```

### 9. zlib:

- Provides compression and decompression functionalities.

- **Example:**

```
const zlib = require('zlib'); const fs = require('fs'); const gzip = zlib.createGzip(); const
readStream = fs.createReadStream('file.txt'); const writeStream =
fs.createWriteStream('file.txt.gz'); readStream.pipe(gzip).pipe(writeStream);
```

- ✓ **local Modules**

- local modules are created locally in your Node.js application
- Local modules in Node.js refer to custom modules created by developers within their project or application. These modules encapsulate related functionalities, making the code more modular, organized, and maintainable.
- Local modules are particularly useful for separating concerns and promoting code reuse.

Stepwise explanation of how to create and use a local module in Node.js:

**1. Create a Local Module:**

Let's create a simple local module named **mathOperations.js** that exports basic mathematical operations:

```
// mathOperations.js
// Exporting addition function
exports.add = function (a, b) { return a + b; };
// Exporting subtraction function
exports.subtract = function (a, b) { return a - b; };
```

In this example, the module exports two functions, **add** and **subtract**.

**2. Use the Local Module in Another File:**

Now, let's create a file (**main.js**) where we'll use the **mathOperations** module:

```
// main.js
// Requiring the local module
const math = require('./mathOperations');
// Using the exported functions
console.log(math.add(5, 3));
// Output: 8 console.log(math.subtract(8, 3));
// Output: 5
```

In this file, we use **require** to import the **mathOperations** module. We then use the exported functions (**add** and **subtract**) as part of our application.

**3. Run the Application:**

Save both files (**mathOperations.js** and **main.js**) in the same directory and run **main.js** using the **node** command:

```
node main.js
```

You should see the output:

```
8
5
```

*Explanation(for understanding purpose)*

- **Creating the Local Module (mathOperations.js):**

- The module exports two functions (**add** and **subtract**) using **exports**.
- **Using the Local Module (main.js):**
  - The **require** function is used to import the local module (**mathOperations**).
  - The exported functions (**add** and **subtract**) are used in the application.
  - Rvy
- **Relative Path:**
  - When using **require** for local modules, provide a relative path to the module file. In this example, **./mathOperations** specifies that the module is in the same directory.
- **File Extensions:**
  - If the module file has a **.js** extension, it's typically omitted in the **require** statement.
- **Exporting Objects or Variables:**
  - Besides functions, local modules can export objects, variables, or any other valid JavaScript entities.
- **Organizing Modules:**
  - As projects grow, it's common to organize modules into subdirectories based on functionality or features.

#### ✓ **Third-party Modules**

Third-party modules in Node.js are modules created by external developers and are not part of the Node.js core. These modules are hosted on the npm (Node Package Manager) registry, which is a central repository for Node.js packages. Third-party modules provide additional functionalities and tools that you can integrate into your Node.js projects, saving development time and effort.

### **How to Use Third-Party Modules:**

1. Install the Module:
  - Use the npm install command to install a third-party module. For example, to install the axios module for making HTTP requests:  
npm install axios
2. Require the Module:
  - In your Node.js script, use the require function to include the module. For example:  
const axios = require('axios');
3. Use the Module:
  - Once the module is installed and required, you can use its functions, objects, or methods in your code. For example:  

```

axios.get('https://jsonplaceholder.typicode.com/users')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => { console.error('Error:', error.message); });

```

### **Where to Use Third-Party Modules:**

1. Web Development:
  - Use third-party modules for tasks such as handling HTTP requests (e.g., axios, request-promise), creating web servers (e.g., express, koa), and managing authentication (e.g., passport).
2. Database Integration:

- Third-party modules like mongoose are commonly used for connecting to and interacting with databases (e.g., MongoDB).
- 3. Utility Functions:
  - Use modules like lodash for utility functions that simplify common programming tasks.
- 4. Testing:
  - Third-party testing frameworks (e.g., mocha, jest) and assertion libraries (e.g., chai, assert) can be employed for writing and running tests.
- 5. Authentication and Authorization:
  - Modules like passport are widely used for implementing authentication strategies in web applications.
- 6. Logging and Debugging:
  - Utilize logging modules (e.g., winston, morgan) to enhance debugging and monitor application behavior.
- 7. File Uploads:
  - Modules like multer simplify handling file uploads in web applications.

### **Applications of using Third-Party Modules:**

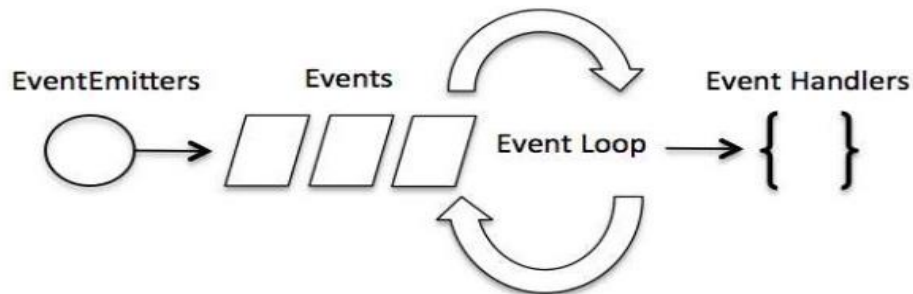
1. Time Efficiency:
  - Third-party modules can save development time by providing pre-built solutions for common tasks, allowing developers to focus on application-specific logic.
2. Code Quality:
  - Reputable third-party modules are often well-tested and maintained, contributing to overall code quality and reliability.
3. Community Collaboration:
  - Leveraging third-party modules allows you to benefit from the expertise and collaboration of the broader Node.js community.
4. Scalability:
  - Many third-party modules are designed to scale, enabling developers to build scalable applications without reinventing the wheel.
5. Feature Enhancement:
  - Third-party modules often provide advanced features or functionalities that might be complex to implement from scratch.
6. Security:
  - Popular and well-maintained modules are regularly updated to address security vulnerabilities, contributing to a more secure development environment.
7. Ecosystem Compatibility:
  - Third-party modules are designed to work seamlessly within the Node.js ecosystem, providing compatibility with other modules and tools.

### **List of some commonly used and popular third-party modules in the Node.js ecosystem:**

1. **Express:** *A minimal and flexible web application framework for building web and mobile applications.*
  - [Express on npm](#)
2. **Lodash:** *A utility library that provides helper functions for common programming tasks.*

- [Lodash on npm](#)
- 3. **Axios:** A promise-based HTTP client for making HTTP requests.
  - [Axios on npm](#)
- 4. **Mongoose:** An elegant MongoDB object modeling for Node.js.
  - [Mongoose on npm](#)
- 5. **Request:** Simplified HTTP request client.
  - [Request on npm](#)
- 6. **Body-parser:** Middleware for parsing incoming request bodies in a middleware-friendly way.
  - [Body-parser on npm](#)
- 7. **Multer:** Middleware for handling **multipart/form-data**, used for file uploads.
  - [Multer on npm](#)
- 8. **Passport:** Authentication middleware for Node.js.
  - [Passport on npm](#)
- 9. **Socket.io:** Enables real-time, bidirectional, and event-based communication.
  - [Socket.io on npm](#)
- 10. **Jest:** A delightful JavaScript testing framework.
  - [Jest on npm](#)
- 11. **Mocha:** A feature-rich JavaScript test framework.
  - [Mocha on npm](#)
- 12. **Chai:** Assertion library for node and browsers.
  - [Chai on npm](#)

## ❖ Node js Events



- Node.js has a built-in module, called "Events", where you can create-, fire-, and listen for- your own events.
- Every action on a computer is an event. Like when a connection is made or a file is opened.

### ✓ key concepts related to events in Node.js:

1. **Event Emitter:** The Event Emitter class is at the heart of the Node.js event system. It is part of the core events module. Objects in Node.js that need to emit events inherit from Event Emitter. This class provides methods for adding listeners to events, emitting events, and managing event listeners.

**Methods:-**

Sr.No.	Method & Description
--------	----------------------



1	<b>addListener(event, listener)</b> Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained.
2	<b>on(event, listener)</b> Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained.
3	<b>once(event, listener)</b> Adds a one time listener to the event. This listener is invoked only the next time the event is fired, after which it is removed. Returns emitter, so calls can be chained.
4	<b>removeListener(event, listener)</b> Removes a listener from the listener array for the specified event. <b>Caution</b> – It changes the array indices in the listener array behind the listener. removeListener will remove, at most, one instance of a listener from the listener array. If any single listener has been added multiple times to the listener array for the specified event, then removeListener must be called multiple times to remove each instance. Returns emitter, so calls can be chained.
5	<b>removeAllListeners([event])</b> Removes all listeners, or those of the specified event. It's not a good idea to remove listeners that were added elsewhere in the code, especially when it's on an emitter that you didn't create (e.g. sockets or file streams). Returns emitter, so calls can be chained.
6	<b>setMaxListeners(n)</b> By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event. This is a useful default which helps finding memory leaks. Obviously not all Emitters should be limited to 10. This function allows that to be increased. Set to zero for unlimited.
7	<b>listeners(event)</b> Returns an array of listeners for the specified event.
8	<b>emit(event, [arg1], [arg2], [...])</b> Execute each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise.

*Syntax:-*

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

Methods:

The EventEmitter class provides several methods for working with events.

**Here are some key methods with syntaxes:**

- `on(eventName, listener)`:- The `on` method is used to attach a listener to an event. It adds a listener function for the specified event.

Syntax: -

```
myEmitter.on('myEvent', (arg) => {
  console.log(`Event occurred with argument: ${arg}`);
});
```

- `emit(eventName, [arg1], [arg2], [...])`:-Emits the specified event, triggering all attached listeners. Optional arguments can be passed to the listeners.

Syntax: - `myEmitter.emit('myEvent', 'Hello, Node.js!');`

- `once(eventName, listener)`:-Adds a one-time listener function for the specified event. The listener is automatically removed after being called once.

```
Syntax: - myEmitter.once('oneTimeEvent', () => {
  console.log('This listener will be called only once. ');
});
```

- `removeListener(eventName, listener)`:

Removes a specific listener for the specified event.

```
Syntax: - const myListener = (arg) => {
  console.log(`Event occurred with argument: ${arg}`);
};
```

```
myEmitter.on('myEvent', myListener);
```

*// Remove the listener*

```
myEmitter.removeListener('myEvent', myListener);
```

- `removeAllListeners([eventName])`:

Removes all listeners for the specified event. If no event is provided, removes all listeners for all events.

Syntax: - `myEmitter.removeAllListeners('myEvent');` *// Remove all listeners for 'myEvent'*

`myEmitter.removeAllListeners();` *// Remove all listeners for all events*

### **Example:**

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

*// Create an instance of EventEmitter*

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

*// Add a listener for the 'myEvent' event*

```
myEmitter.on('myEvent', (arg) => {
  console.log(`Event occurred with argument: ${arg}`);
});
```

*// Emit the 'myEvent' event*

```
myEmitter.emit('myEvent', 'Hello, Node.js!');
```

In this example:

- An instance of `EventEmitter` is created with `new EventEmitter()`.
- A listener is added to the `'myEvent'` event using `on`.

- The emit method is used to trigger the 'myEvent' event, and the listener logs a message to the console with the provided argument.
2. **Event:** An event is a named signal that indicates that something of interest has happened. It can be a system event, user action, or any other occurrence in the application. Events are identified by string names.
  3. **Listener:** -It is executed when the event it is listening for is emitted.  
In Node.js, listeners are functions that are attached to specific events emitted by instances of the EventEmitter class. Listeners are executed when the corresponding event is emitted.

Syntax:-

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

```
// Adding a listener for the 'myEvent' event
myEmitter.on('myEvent', (arg) => {
  console.log(`Event occurred with argument: ${arg}`);
});
```

```
// Emitting the 'myEvent' event
myEmitter.emit('myEvent', 'Hello, Node.js!');
```

1. Create an instance of EventEmitter:

Import the EventEmitter class from the 'events' module.

Create an instance of EventEmitter using new EventEmitter().

2. Add a listener for an event:

Use the on method to add a listener for a specific event.

The event name ('myEvent' in this case) is a string that identifies the event.

The second argument is the listener function, which will be executed when the event is emitted.

3. Emit an event:

Use the emit method to trigger the specified event.

Optional arguments can be passed to the listeners.

Example:-

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

```
// Create an instance of EventEmitter
const myEmitter = new EventEmitter();
```

```
// Add a listener for the 'myEvent' event
myEmitter.on('myEvent', (arg) => {
  console.log(`Event occurred with argument: ${arg}`);
});
```

```
// Emit the 'myEvent' event with an argument  
myEmitter.emit('myEvent', 'Hello, Node.js!');
```

#### 4. **Emitter and Receiver:**

##### ✓ Emitter:

- An emitter is an object that has the capability to emit events. In Node.js, instances of the EventEmitter class are commonly used as emitters.
- Emitters are responsible for signaling that a certain event has occurred, and they provide a mechanism for listeners to respond to these events.
- Examples of emitters in Node.js include HTTP servers, file system modules, and custom objects that extend the EventEmitter class.

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

```
// Emit an event named 'myEvent'  
myEmitter.emit('myEvent', 'Event data');
```

##### ✓ Receiver (or Listener):

- A receiver, often referred to as a listener, is a function that "listens" for a specific event emitted by an emitter. When the event occurs, the listener is invoked.
- Receivers are attached to emitters, and they are designed to respond to specific events of interest.
- Multiple listeners can be attached to the same event, and they will all be executed in the order they were registered.

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

```
// Add a listener for the 'myEvent' event  
myEmitter.on('myEvent', (data) => {  
  console.log(`Event received with data: ${data}`);  
});
```

In the example above:

- myEmitter is the emitter, as it is an instance of EventEmitter.
- The on method is used to attach a listener to the 'myEvent' event.
- When myEmitter.emit('myEvent', 'Event data') is called, the listener function is executed with the provided data.

Example with Multiple Listeners:

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

```
// Listener 1  
myEmitter.on('myEvent', (data) => {  
  console.log(`Listener 1: Event received with data: ${data}`);  
});
```

```
// Listener 2
myEmitter.on('myEvent', (data) => {
  console.log(`Listener 2: Event received with data: ${data}`);
});

// Emit the 'myEvent' event with data
myEmitter.emit('myEvent', 'Event data');
```

5. **Event Loop:** Node.js uses an event loop to handle asynchronous operations. The event loop continually checks for events and executes the associated listeners.

Key concepts in Event Loop in Node.js:

- **Event Loop Phases:** The event loop in Node.js consists of several phases, and each phase has a specific set of tasks to perform. The main phases include timers, I/O callbacks, idle, prepare, poll, check, and close callbacks.
- **Timers:** Timers phase handles the execution of `setTimeout()` and `setInterval()` callbacks.
- **I/O Callbacks:** I/O callbacks phase executes callbacks related to I/O events, such as network requests or file system operations.
- **Idle and Prepare:** These are internal phases that are not often used in typical Node.js applications.
- **Poll:** The poll phase is responsible for retrieving new I/O events from the event queue. If there are no new events, it waits for events to be added. If there are pending callbacks, it executes them.
- **Check:** The check phase executes `setImmediate()` callbacks.
- **Close Callbacks:** The close callbacks phase executes callbacks for close events, such as socket or file closures.
- **Example:**

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

// Schedule a timer callback
setTimeout(() => {
  console.log('Timer callback executed.');
```

```
}, 1000);

// Read a file asynchronously
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    throw err;
  }
  console.log('File read callback executed.');
```

```
});

// Schedule a setImmediate callback
setImmediate(() => {
```

```
console.log('setImmediate callback executed.');
```

```
});  
  
// Log a message  
console.log('Main thread execution.');
```

*// This is a simple example to illustrate the event loop. In a real application, you would have more asynchronous operations.*

In this example:

- A timer callback is scheduled with `setTimeout`.
- A file read operation is performed asynchronously with `fs.readFile`.
- A set Immediate callback is scheduled with `setImmediate`.
- The main thread logs a message.

✓ **Examples of Events(For more once check here)**

<https://www.geeksforgeeks.org/node-js-events/>

## ❖ Node js with Express js

<https://www.javatpoint.com/expressjs-tutorial>

- Node.js, combined with the Express.js framework, is a powerful and popular combination for building web applications and APIs. Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It simplifies the process of building scalable and maintainable web applications by providing a set of conventions and middleware for common tasks.

### **Getting Started:**

#### **1. Install Node.js:**

- Ensure you have Node.js installed on your machine. You can download it from <https://nodejs.org/>.

#### **2. Initialize a Node.js Project:**

- Open a terminal and navigate to your project directory.
- Run the following command to initialize a new Node.js project and create a **package.json** file:

```
npm init -y
```

#### **3. Install Express:**

- Install Express.js using the following command:  

```
npm install express
```

### **Create a Simple Express App:**

Create a file (e.g., **app.js**) with the following content:

```
const express = require('express'); const app = express(); const port = 3000;  
// Define a route app.get('/', (req, res) => { res.send('Hello, Express!'); });  
// Start the server app.listen(port, () => { console.log(`Server listening at  
http://localhost:${port}`); });
```

### **Run the Express App:**

In the terminal, run the following command to start the Express app:  
node app.js

You should install the following important modules along with express –

- body-parser – This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.  
*\$ npm install body-parser --save*
- cookie-parser – Parse Cookie header and populate req.cookies with an object keyed by the cookie names.  
*\$ npm install cookie-parser --save*
- multer – This is a node.js middleware for handling multipart/form-data.  
*\$ npm install multer --save*

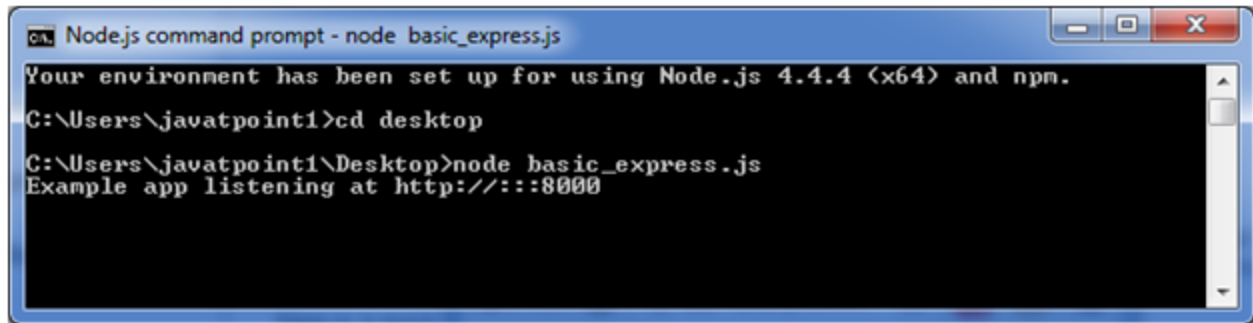
Visit <http://localhost:3000> in your web browser, and you should see "Hello, Express!".

Example:-

Let's see a basic Express.js app.

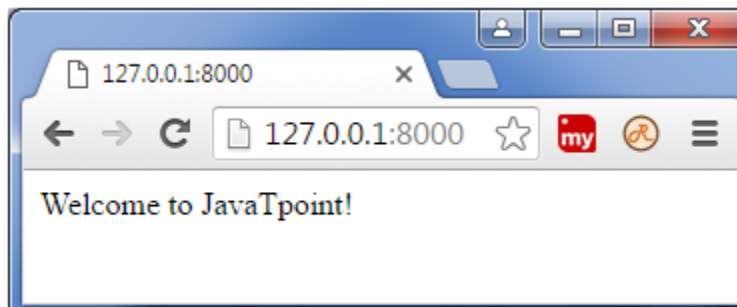
**File: basic\_express.js**

```
var express = require('express');
var app = express();
app.get('/', function (req, res) {
  res.send('Welcome to JavaTpoint!');
});
var server = app.listen(8000, function () {
  var host = server.address().address;
  var port = server.address().port;
  console.log('Example app listening at http://%s:%s', host, port);
});
```



```
Node.js command prompt - node basic_express.js
Your environment has been set up for using Node.js 4.4.4 (x64) and npm.
C:\Users\javatpoint1>cd desktop
C:\Users\javatpoint1\Desktop>node basic_express.js
Example app listening at http://:::8000
```

## Output:



## Key Express.js Concepts:

### 1. Routing:

- Define routes to handle HTTP requests and define the behavior of your application based on those routes.

### 2. Middleware:

- Express middleware functions are functions that have access to the request, response, and the next function in the application's request-response cycle. They can perform actions, modify the request or response objects, and terminate the request-response cycle.

### 3. Template Engines:

- Express can be used with various template engines (e.g., EJS, Pug) to dynamically generate HTML pages.

### 4. Static Files:

- Use **express.Static** middleware to serve static files like images, stylesheets, and scripts.

### 5. Middleware Stacks:

- Express allows you to stack middleware functions to execute in a specific order.

## ✓ Why Use Node.js with Express.js:

### • Fast and Lightweight:

- Express is designed to be minimal and flexible, allowing you to build scalable applications with less boilerplate code.

### • Middleware Ecosystem:

- A rich ecosystem of middleware simplifies handling common tasks, such as authentication, logging, and request processing.

### • Routing:



- Express provides a simple and powerful routing system for defining how an application responds to client requests.
- **Template Engines:**
  - Integrates seamlessly with various template engines for rendering dynamic views.
- **RESTful API Development:**
  - Well-suited for building RESTful APIs and serving JSON data.
- **Active Community:**
  - Large and active community support with a wealth of resources and third-party modules.
- **Flexibility:**
  - Offers the flexibility to structure your application according to your needs rather than imposing a strict architecture.

### ✓ Companies That Are Using Express JS

Netflix  
IBM  
ebay  
Uber

## ❖ Node.js Database Access

- Node.js database access refers to the ability of a Node.js application to interact with databases, whether they are relational databases like MySQL or NoSQL databases like MongoDB. The interaction typically involves tasks such as querying data, inserting records, updating information, and deleting data.

### ❖ MySQL Database Access with Node.js:

To access a MySQL database from Node.js, you can use the `mysql2` library, which is a fast and feature-rich MySQL library for Node.js.

1. Open a Terminal/Command Prompt:

On Windows, you can use Command Prompt or PowerShell.

On macOS or Linux, you can use Terminal.

2. Navigate to Your Project Directory:

*cd path/to/your/project*

3. Install the `mysql2` Package:

*npm install mysql2*

This command will download and install the `mysql2` package from the npm registry and add it to your project.

4. Verify Installation:

Once the installation is complete, you can check if the package is added to your `package.json` file and the node modules directory is created in your project folder.

*// Inside your package.json file*

```
"dependencies": {
  "mysql2": "^2.3.0"
}
```

5. Require `mysql2` in Your Code:

In your Node.js application code, require the `mysql2` module to use it.

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

#### 6. Create a Connection:

You can then use the create Connection method to establish a connection to your MySQL database.

```
const connection = mysql.createConnection({  
  host: 'your_database_host',  
  user: 'your_database_user',  
  password: 'your_database_password',  
  database: 'your_database_name'  
});
```

Replace the placeholder values with your actual database connection details.

#### 7. Execute Queries:

You can use the connection to execute SQL queries.

For example:

```
connection.query('SELECT * FROM your_table', (error, results, fields) => {  
  if (error) {  
    console.error(error);  
  } else {  
    console.log(results);  
  }  
  connection.end(); // Close the connection when done  
});
```

This is a basic example, and you should handle errors and close the connection appropriately in a production environment.

- ❖ To create, update, and delete tables in a database using Node.js, we'll use the **mysql2** library for MySQL as an example. The process involves executing SQL queries to the database

#### 8. npm install mysql2

```
const mysql = require('mysql2');  
  
// Create a connection to the MySQL database  
const connection = mysql.createConnection({  
  host: 'your-mysql-host',  
  user: 'your-mysql-username',  
  password: 'your-mysql-password',  
  database: 'your-mysql-database',  
});  
  
// Connect to the database  
connection.connect((err) => {  
  if (err) {  
    console.error('Error connecting to MySQL:', err);  
    return;  
  }  
  console.log('Connected to MySQL database');  
});
```

```

// 1. Create Table
const createTableQuery = `
CREATE TABLE IF NOT EXISTS users (
id INT PRIMARY KEY AUTO_INCREMENT,
username VARCHAR(255) NOT NULL,
email VARCHAR(255) NOT NULL
)
`;

connection. Query(createTableQuery, (err, results) => {
if (err) throw err;
console.log ('Table created or already exists:', results);
});

// 2. Insert Data
const insertDataQuery = `
INSERT INTO users (username, email) VALUES
('john_doe', 'john@example.com'),
('jane_doe', 'jane@example.com')
`;

connection.query(insertDataQuery, (err, results) => {
if (err) throw err;
console.log('Data inserted:', results);
});

// 3. Update Data
const updateDataQuery = `
UPDATE users SET email = 'new_email@example.com' WHERE username =
'john_doe'
`;

connection.query(updateDataQuery, (err, results) => {
if (err) throw err;
console.log('Data updated:', results);
});

// 4. Delete Data
const deleteDataQuery = `
DELETE FROM users WHERE username = 'jane_doe'
`;

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

```

```
if (err) throw err;
console.log('Data deleted:', results);
});
```

```
// Close the connection
```

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

*(Make sure to replace placeholder values such as 'your-mysql-host', 'your-mysql-username', etc., with your actual database details.)*

### ✓ **Mongoose (MongoDB) Database Access with Node.js:**

To access MongoDB from Node.js, you can use Mongoose, an ODM (Object Data Modeling) library for MongoDB and Node.js.

1. Install the mongoose package:

```
npm install mongoose
```

2. Example Code for MongoDB Access:

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

```
// Connect to MongoDB
```

```
mongoose.connect('mongodb://localhost:27017/your-mongodb-database', { useNewUrlParser: true, useUnifiedTopology: true });
```

```
// Define a mongoose schema
```

```
const userSchema = new mongoose.Schema({
  username: String,
  email: String,
});
```

```
// Define a mongoose model
```

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

```
// Create a new user
```

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

```
// Save the user to the database
```

```
newUser.save((err, savedUser) => {
  if (err) throw err;
  console.log('User saved:', savedUser);
});
```

```
// Find users in the database
```

```
User.find({}, (err, users) => {
  if (err) throw err;
```

```
console.log ('Users:', users);  
});
```

## ***Assignment questions***

### **Node.js Basics and Setup:**

1. Explain what Node.js is and why it is used for server-side development and explain the installation steps.
2. What is the Node Package Manager (npm), and how is it used in Node.js development?

### **Node.js Console and Command Utilities:**

3. What is the Node.js console, and how can you interact with it?
4. How do you print messages to the console using console.log () in Node.js?
5. Explain the purpose of the process object in Node.js and provide an example of its usage.
6. Describe the role of the os module in Node.js and provide an example of its usage.
7. What is the purpose of the fs module in Node.js, and how can you read a file using this module?

### **Node.js Modules and Concepts:**

8. Explain the concept of Commons modules in Node.js and provide an example of creating and using a module.
9. Describe the difference between built-in core modules and external modules in Node.js.
10. What is the purpose of the module. Exports object, and how is it used to export functionality from a module?
11. What is callback hell in Node.js, and how can it be mitigated using callbacks and Promises?
12. Define the Singleton pattern and explain how it can be applied in Node.js modules.
13. Describe the Event Loop in Node.js and how it enables asynchronous, non-blocking behavior in your applications.

### **Node.js Events:**

14. What are events in Node.js, and how do they facilitate asynchronous programming?
15. Explain the role of the Event Emitter class in Node.js, and provide an example of creating and emitting custom events.
16. Describe the difference between synchronous and asynchronous event handling in Node.js, and provide use cases for each.
17. What is the purpose of the once method in the Event Emitter class, and when might you use it in your code?

### **Node.js with Express.js:**

18. What is Express.js, and how does it simplify building web applications in Node.js?
19. Explain the role of middleware in an Express.js application, and provide an example of using middleware for request processing.
20. How can you handle HTTP requests in an Express.js application, and provide an example of routing for GET and POST requests?

### **Node.js Database Access:**

21. Discuss the importance of database access in Node.js applications and the types of databases commonly used.
22. How do you establish a connection to a relational database, such as MySQL or PostgreSQL, using Node.js and a popular library like mysql2?

23. Describe the concept of NoSQL databases and provide an example of accessing and querying a NoSQL database, such as MongoDB, in a Node.js application.