What is Node.Js?

Node.js is an open-source, server-side JavaScript runtime environment built on Chrome's V8 JavaScript engine. It allows developers to run JavaScript code outside of a web browser, enabling them to build scalable and efficient network applications.

Node.js utilizes an event-driven, non-blocking I/O model, which means it can handle multiple concurrent connections and perform asynchronous operations efficiently. This design makes it particularly well-suited for building real-time applications, such as chat servers, streaming platforms, or any application that requires handling a large number of concurrent connections.

One of the key advantages of Node.js is its ability to use JavaScript on both the client-side and server-side, enabling developers to share code and easily transition between different parts of the application. This streamlines the development process and promotes code reusability.

Initial Release Date: **27 may 2009**
Licence: **MIT licence**
Operating System: **Microsoft Windows, Linux, SmartOS, MacOS, freeBSD, OpenBSD, IBM AIX**

Why Node.js?

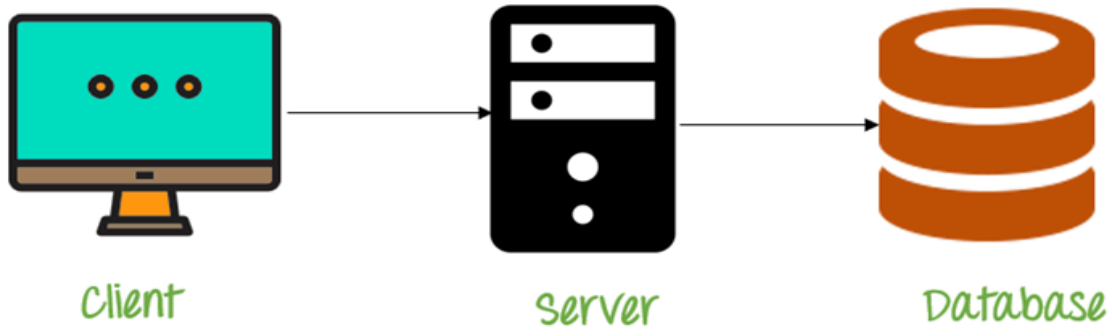
Node.js allows developers to use JavaScript on both the client-side and server-side. This means that developers can write their entire application using a single language, making it easier to share code, maintain consistency, and transition between front-end and back-end development. JavaScript is also a widely-used and well-documented language, with a large community of developers and extensive libraries and frameworks available.

Difference Between Websites and Web Application

Website	Web Application
It basically contains static content.	It is designed for interaction with end-users.
The user of the website can only read the content of the website but is unable to manipulate it.	The users of the web applications are able to read the content of the web application or are able to manipulate the data.
This does not need to be precompiled.	This site should be precompiled before the deployment.
The website's function is quite simple.	The web application's function is quite complex.
The website is not interactive for users.	The web application is interactive for users.
The browser capability included with the website is high.	The browser capabilities included with a web application are high.
Authentication is not necessary on the website.	Authentication must be required in mostly web applications.
For the website, integration is simple.	For web applications, integration is complex because it has complex functionality.
Ex :- Breaking news, wikipedia, etc.	Ex :- Flipkart, Amezon, etc.

What is HTTP?

HTTP (Hypertext Transfer Protocol) is an application protocol used for transmitting hypermedia documents, such as HTML files, over the internet. It is the foundation of data communication on the World Wide Web. HTTP defines how clients (such as web browsers) and servers communicate with each other to request and deliver resources.



HTTP Request / Response:-
Communication between clients and servers is done by request and response.

- A client (browser) sends an **HTTP request** to the web.
- A web server receives the request.
- The server runs an application to process the request.
- The server returns an **HTTP response** to the browsers.
- The client (browser) receives the response.

HTTP Request / Response Circle:
The browser requests an **HTML** page. The server returns an **HTML** file.
The browser requests a **stylesheet**. The server returns a **CSS** file.
The browser requests a **JPG** image. The server returns a **JPG** file.
The browser requests **javascript** code. The server returns a **JS** file.
The browser requests **data**. The server returns **data** (**XML** or **JSON**)

What is a Module in Node.js?
Consider modules to be the same javascript libraries. A set of functions you want to include in your application.

Each module in Node.js has its own scope, meaning the variables, functions, and other declarations inside a module are not accessible from outside unless explicitly exported.

Node.js provides a set of built-in modules that offer commonly used functionality. These modules are available without requiring any additional installation and can be accessed using the **require()** function. Examples: include the http module for creating HTTP servers, the fs module for file system operations, and the path module for working with file paths.

1. HTTP/HTTPs Module: The HTTP (Hypertext Transfer Protocol) and HTTPS (HTTP Secure) modules in Node.js are used to handle server-side communication over the HTTP and HTTPS protocols, respectively.

The HTTP module provides a set of functions and classes for creating HTTP servers and making HTTP requests. It allows you to listen for incoming HTTP requests on a specific port and handle them accordingly. You can create an HTTP server using the **http.createServer()** method and specify a callback function to handle each incoming request. This module also provides methods to send HTTP responses, set headers, handle URL routing, and handle data streams.

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

// Create a server
const app = http.createServer((request, response) => {
    response.write('Welcome to NodeJS');
    response.end();
});

app.listen(3000, ()=>{
    console.log(`Server connected at port 3000`)
});
```

2. File system: To handle file operations like creating, reading, deleting, etc., Node.js provides an inbuilt module called FS (File System). Node.js gives the functionality of file I/O by providing wrappers around the standard POSIX functions. All file system operations can have synchronous and asynchronous forms depending upon user requirements.

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

Open a File: The fs.open() method is used to create, read, or write a file. The fs.readFile() method is only for reading the file and fs.writeFile() method is only for writing to the file, whereas the fs.open() method does several operations on a file. First, we need to load the fs class which is a module to access the physical file system.

```
fs.open(path, flags, mode, callback)
```

Parameters:

- **path:** It holds the name of the file to read or the entire path if stored at other locations.
- **flags:** Flags indicate the behaviour of the file to be opened. All possible values are (r, r+, rs, rs+, w, wx, w+, wx+, a, ax, a+, ax+).
- **mode:** Sets the mode of file i.e. r-read, w-write, r+ -readwrite. It is set to default as readwrite.
- **err:** If any error occurs.
- **data:** Contents of the file. It is called after the open operation is executed.

Example:

```
var fs = require("fs");
// Asynchronous - Opening File
console.log("opening file!");
fs.open('input.txt', 'r+', (err, fd)=> {
    if (err) {
        return console.log(err);
    }
console.log("File open successfully");
});
```

Reading a File: The fs.readFile() method is an inbuilt method which is used to read the file. This method reads the entire file into a buffer. To load the fs module we use require() method.

fs.readFile(filename, encoding, callback_function)

Parameters:

- **filename:** It holds the name of the file to read or the entire path if stored at another location.
- **encoding:** It holds the encoding of a file. Its default value is 'utf8'.
- **callback_function:** It is a callback function that is called after reading a file. It takes two parameters:
  - **err:** If any error occurred.
  - **data:** Contents of the file.

Example:

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

1. fs.readFile('Demo.txt', 'utf8', (err, data)=>{

     // Display the file content
     console.log(data);
   });

   console.log('readFile called');

2. let readFile = fs.readFileSync('Demo.txt', 'utf8');

   console.log('readFile called: ' + readFile);
```

Writing to a File: This method will overwrite the file if the file already exists. The fs.writeFile() method is used to asynchronously write the specified data to a file. By default, the file would be replaced if it exists. The 'options' parameter can be used to modify the functionality of the method.

fs.writeFile(path, data, options, callback)

Parameters:

- **path:** It is a string, Buffer, URL, or file description integer that denotes the path of the file where it has to be written. Using a file descriptor will make it behave similarly to fs.write() method.
- **data:** It is a string, Buffer, TypedArray, or DataView that will be written to the file.
- **options**: It is a string or object that can be used to specify optional parameters that will affect the output. It has three optional parameters:
- **encoding:** It is a string value that specifies the encoding of the file. The default value is 'utf8'.
- **mode:** It is an integer value that specifies the file mode. The default value is 0o666.
- **flag**: It is a string value that specifies the flag used while writing to the file. The default value is 'w'.
- **callback:** It is the function that would be called when the method is executed.
    - **err:** It is an error that would be thrown if the operation fails.

Example:

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

fs.writeFile('input.txt', 'Hello World!',(err)=> {
    if (err)
        return console.error(err);
console.log("Data written successfully!");

fs.readFile('input.txt', 'utf-8' , (err, data)=> {
    if (err)
        return console.error(err);
console.log("Asynchronous read: " + data.toString());
});
});
```

Appending to a File: The fs.appendFile() method is used to synchronously append the data to the file.

fs.appendFile(filepath, data, options, callback); **or**
fs.appendFileSync(filepath, data, options);

Parameters:

- **filepath:** It is a String that specifies the file path.
- **data:** It is mandatory and it contains the data that you append to the file.
- **options:** It is an optional parameter that specifies the encoding/mode/flag.
- **Callback:** Function is mandatory and is called when appending data to file is completed.

Example: **1. Asynchronously appending**

```
var fs = require('fs');
var data = "\nLearn Node.js";
// Append data to file
fs.appendFile('input.txt', data, 'utf8',
        (err)=> {                    // Callback function
                if (err) throw err;
        console.log("Data is appended to file successfully.")
});
```

2. synchronously appending

```
var fs = require('fs');
var data = "\nLearn Node.js";

// Append data to file
fs.appendFileSync('input.txt', data, 'utf8');
console.log("Data is appended to file successfully.")
```

Closing the File: The fs.close() method is used to asynchronously close the given file descriptor thereby clearing the file that is associated with it. This will allow the file descriptor to be reused for other files. Calling fs.close() on a file descriptor while some other operation is being performed on it may lead to undefined behavior.

fs.close(fd, callback)

Parameters:

- **fd:** It is an integer that denotes the file descriptor of the file for which to be closed.
- **callback:** It is a function that would be called when the method is executed.
    - **err:** It is an error that would be thrown if the method fails.

Example:

```
// Close the opened file.
fs.close(fd, function(err) {
    if (err) {
        console.log(err);
    }
    console.log("File closed successfully.");
}
```

Delete a File: The fs.unlink() method is used to remove a file or symbolic link from the filesystem. This function does not work on directories, therefore it is recommended to use fs.rmdir() to remove a directory.

fs.unlink(path, callback)

Parameters:

- **path:** It is a string, Buffer or URL which represents the file or symbolic link which has to be removed.
- **callback:** It is a function that would be called when the method is executed.
    - **err:** It is an error that would be thrown if the method fails.

Example:

```
var fs = require("fs");
console.log("deleting an existing file");
fs.unlink('input.txt', function(err) {
    if (err)
        return console.error(err);
console.log("File deleted successfully!");
});
```

1. Synchronous methods: Synchronous functions block the execution of the program until the file operation is performed. These functions are also called blocking functions. The synchronous methods have File Descriptor as the last argument. File Descriptor is a reference to opened files. It is a number or a reference id to the file returned after opening the file using fs.open() method of the fs module. All asynchronous methods can perform synchronously just by appending "Sync" to the function name.

2. Asynchronous methods: Asynchronous functions do not block the execution of the program and each command is executed after the previous command even if the previous command has not computed the result. The previous command runs in the background and loads the result once it has finished processing. Thus, these functions are called non-blocking functions. They take a callback function as the last parameter. Asynchronous functions are generally preferred over synchronous functions as they do not block the execution of the program whereas synchronous functions block the execution of the program until it has finished processing.

3. Operation System: OS is a node module used to provide information about the computer operating system.

Advantages:
It provides functions to interact with the operating system. It provides the hostname of the operating system and returns the amount of free system memory in bytes.

```
os.arch() -> cpu architecture
os.freemem() -> amount of free system memory in bytes
os.totalmem() -> total amount of system memory in bytes
os.networkInterfaces() -> list of network interfaces
os.tmpdir () -> operating systems default directory for temp
files
os.endianness() -> endianness of system
os.hostname() -> hostname of system
os.type() -> operating system name
os.platform() -> platform of os
os.release() -> operating systems release
```

4. Path :
The Path Module in Node.js provides the utilities for working with file and directory paths.

```
path.basename() -> get the filename portion of a path to the file
path.dirname() -> get the directory name of the given path
path.extname() -> get the extension portion of a file path
path.format() -> a path string from the given path object
path.isAbsolute() -> check whether the given path is an absolute path
or
                    not
path.join() -> join a number of path segments using the
               platform-specific delimiter to form a single path
path.normalize() -> normalize the given path
path.parse() -> object whose properties represent the given path
path.relative() -> find the relative path from a given path to another
                   path based on the current working directory
path.resolve() -> resolve a sequence of path segments to an absolute
                  path
path.toNamedspacePath() -> find the equivalent namespace-prefixed path
                           from the given path
```

Own Module Create and Export : The **module.exports** in Node.js is used to export any literal, function or object as a module. It is used to include JavaScript files into node.js applications. The module is similar to the variable that is used to represent the current module and exports is an object that is exposed as a module.

```
module.exports = literal | function | object
```

Module Wrapper Function: Under the hood, NodeJS does not run our code directly, it wraps the entire code inside a function before execution. This function is termed as Module Wrapper Function. Refer https://nodejs.org/api/modules.html#modules_the_module_wrapper for official documentation.

Use of Module Wrapper Function in NodeJS:

1. The top-level variables declared with var, const, or let are scoped to the module rather than to the global object.
2. It provides some global-looking variables that are specific to the module, such as:
    - The module and exports object that can be used to export values from the module.
    - The variables like __filename and __dirname, that tells us the module's absolute filename and its directory path.