NodeJS.

[aswnrh.dev@gmail.com](mailto:aswnrh.dev@gmail.com)

**Node.js is an open source, cross platform JavaScript runtime environment.**

it is not a language and it is not a framework.

Runs on V8 engine. Executes JS on outside web browser.

When we say that **Node.js allows you to run JavaScript code outside of a web browser, it means that you can execute JavaScript on the server or on your local machine**, not just within the context of a webpage in a browser.

Why node?

1. Can create dynamic pages and content.
2. Can create, open, read, delete and close files on the server.

**JavaScript can’t access the database but Nodejs can.**

**REPL (Read evaluate print loop)** basically a console window.

**Modules**

**Simple or complex functionalities in single or multiple JS files that can be reused throughout node.js.**

Three types

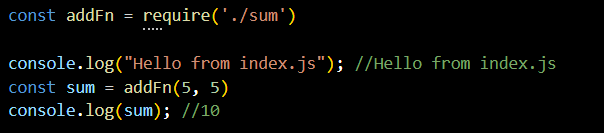
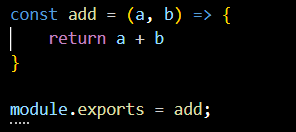
1. **Local modules** (modules that we create in our application)
2. **Core modules/built in modules** (modules that are part of the platform and comes with the node.js installation)
3. **Third party modules/npm** (it allows you to install and use third part npm libraries)

**Common JS**

**Common JS is a standard that states how a module should be structured and shared.**

**Local Modules**

Modules that we create in our application.



The add module is created by we and we have used it in index.js, and we have assigned it to a constant variable.

After creating a variable use module.exports to export a module.

**Module Scope**

**module scope refers to the scope or context within which variables, functions, and other code elements defined in a module are accessible**. **Each Node.js file (also known as a module) has its own scope**, meaning that variables and functions defined within a module are not accessible from other modules unless explicitly exported.

**Module wrapper**In Node.js, **every file you create gets a special cover called a "module wrapper."** **This cover keeps your code organized and safe.** Inside this wrapper, you have tools like `require()` to bring in other files and `exports` to share stuff with them. Additionally, it includes special variables such as \_\_filename and \_\_dirname, which tell the file where it's located on the computer. So, **think of the module wrapper as a protective layer that makes sure your code runs smoothly and doesn't mess with other parts of your project.**

**Core modules**

Module that are part of the platform and comes with node.js installation. Also known as built in modules.

**Npm/ third party modules**

Allow you to use third party npm libraries. npm i cli-color(now we can use the features of cli color in our application)

**Npm i nodemon (used to automatically save the changes)**

**OS Module**

Used to **retrive information from the under laying OS and the computer. The program runs on and interact with it.**

**Var OS = require(‘os’)**

**OS.version(), OS.release() etc.**

**FS Module**

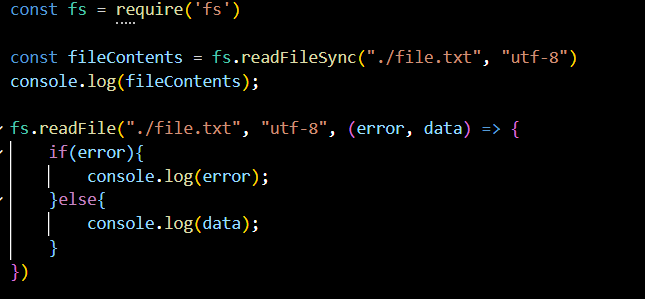
**Allow you to work with file system on your computer.**

Uses -> read, create, update, delete and rename.

Read file

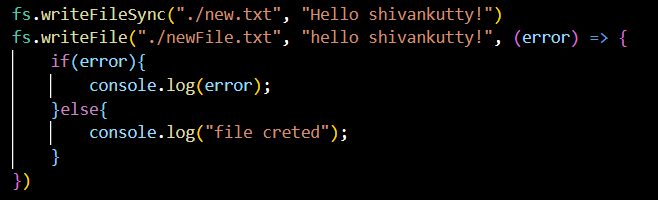
We can create it using readFileSync and readFile

One is synchronous and one is asynchronous

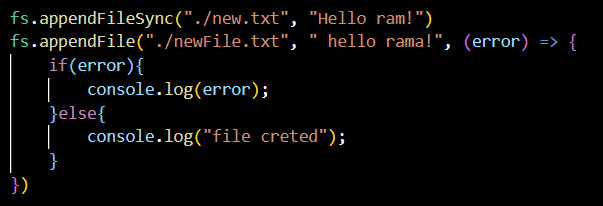


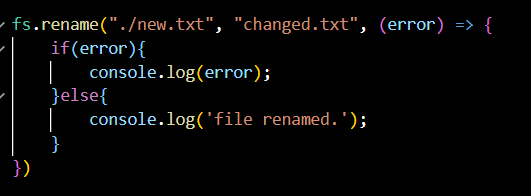
Write file

WriteFile and writeFileSync

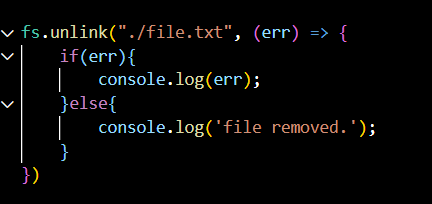


Instead of using write file after the file creation we use appendFile or appendFileSync because if we use write file it will overwrite the data so append is the one we have to use to update the value.



Rename  


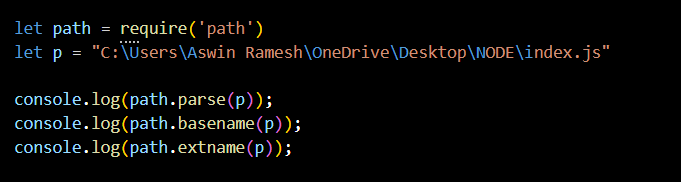
Unlink (Delete)



**Path module**

**Allow you to interact with file path easily.**

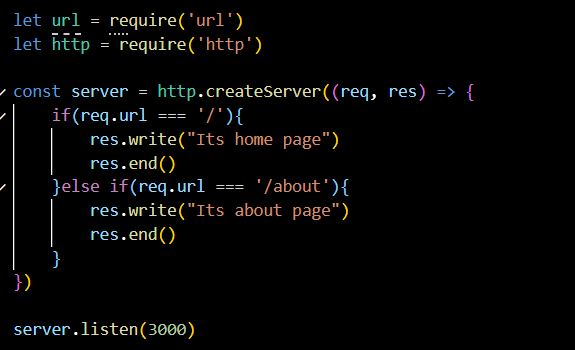
It has many useful properties and methods to access and manipulate path in fs.



**http modules**

**allow you to transfer data over the http. Can create http server that listen to the server port and gives a response back to the client.**

Interaction between browser and the server



**Stream**

**Stream is a sequence of data that is being moved from one point to another over time.** If you’re transferring file contents from fileA to fileB, you don’t wait for entire fileA content to be saved in temporary memory before moving it into fileB. Instead the contents is transferred in chunks over time which prevents unnecessary memory usage.  
**mechanism for reading and writing data in chunks rather than loading the entire item into memory. It is transferred as buffer of data.**

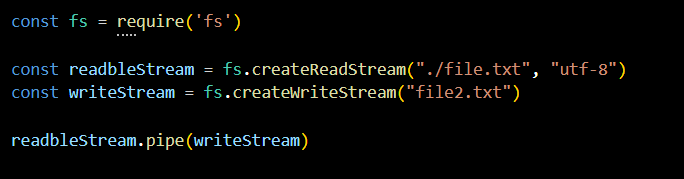
**Ex: watching a video on you tube**

**The data arrives in chunks and you watch in chunks while the rest of the data arrives over time.**

**Buffer: temporary storage area of raw binary data.**

**Types:**

1. **Readable stream**
2. **Writeable stream**
3. **Duplex**
4. **Transform**

****

**Complete read and write operation**

****

**Status code**

Describes the type of code that sent to the browser.

200 OK, 301 Resource moved, 404 Not Found, 500 Internal server error.

100 informational range, 200 success, 300 redirection range, 400 client error, 500 server error.

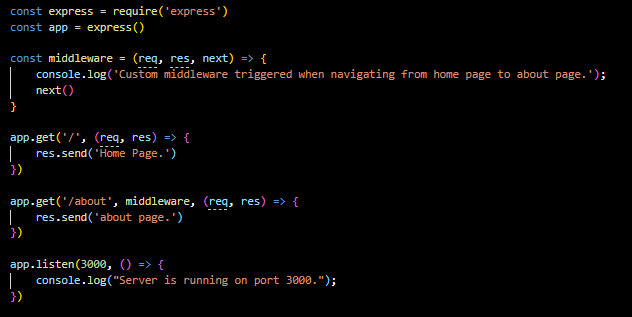
**View Engine**

**view engine is used to render dynamic HTML pages.** It processes template files containing static HTML and placeholders for dynamic data, converting them into complete HTML pages that can be sent to a web browser. **Common view engines include EJS, Pug, and Handlebars.**

**Middleware**

Code that runs between **getting a request and sending a response.** Just like a middleman.

1. **Application-level Middleware**: Functions that handle requests globally or for specific routes.
2. **Router-level Middleware**: Functions that handle requests for specific route paths using express.Router().
3. **Error-handling Middleware**: Functions that handle errors and have four arguments: err, req, res, and next.
4. **Built-in Middleware**: Predefined Express functions for common tasks like serving static files or parsing request bodies.
5. **Third-party Middleware**: Community-provided functions for tasks like handling cookies, sessions, or CORS.
6. **Authentication Middleware**: Functions that verify user identity by checking credentials.
7. **Authorization Middleware**: Functions that check if authenticated users have permission to access specific resources.
8. **Custom Middleware**: User-defined functions for specific application needs like logging or data validation.



**Static files**

Static files refer to any **files in a web application that remain unchanged during runtime.** These files are **typically served directly to the client without any processing by the server.** Common examples of static files include HTML, CSS, JavaScript, images, fonts, and other assets required by a web page or application.

**MVC**

**Model:** Model handles the data and the logic behind the scenes. For example, it knows how to save and retrieve information from a database.

**View:** View is responsible for showing the user interface. It displays the data from the Model in a way that users can understand and interact with, like showing web pages, forms, or images.

**Controller:** Controller receives input from the user via the View, figures out what needs to be done, and tells the Model to update or retrieve data accordingly. Then, it decides which View to show next based on what the user wants.

**Module caching**

**Built in mechanism that allows the runtime to store and reuse previously loaded modules.**

**JSON:** data interchange format commonly used in a web browser.

**Watch mode:** constantly watched the process and restart the updates.

**Event module:** allow us to work with node.js. an action or an occurrence that has happened in one application that we respond to.

**Cluster module**

Nb: read this fully!

The Cluster module in Node.js is like **having a team of workers to handle tasks in your application.** Imagine you're running a restaurant and have multiple chefs in the kitchen. The Cluster module allows you to create copies of your main chef (the master process) called workers. Each worker (chef) can handle incoming orders (requests) independently, making use of all the available cooking stations (CPU cores) efficiently. When a new order comes in, the master chef decides which worker should take it based on who's available. If one of the chefs takes a break or burns the dish (crashes), the master chef quickly replaces them with a new chef to keep things running smoothly. This way, your restaurant (Node.js application) can serve more customers (handle more requests) without getting overwhelmed, leading to faster service and happier customers.

The Cluster module in Node.js is **a built-in module that allows a Node.js application to create multiple instances of itself, called worker processes, to handle incoming requests concurrently.** This module is particularly useful for applications running on multi-core systems, as it enables the application to utilize all available CPU cores effectively, thereby improving performance and scalability.

* **child processes are instances of the Node.js application that are spawned by the master process to serve as workers**. These child processes are separate instances of the same application running independently, each handling incoming requests.
* The main use of child processes in the Cluster module is to leverage the capabilities of multi-core systems effectively. **By creating multiple instances (child processes) of the Node.js application, the Cluster module allows the application to utilize all available CPU cores**, improving performance and scalability.
* **Each child process (worker) operates autonomously, handling incoming requests without interfering with other child processes**. This parallel processing approach enables the application to **handle a higher volume of requests concurrently**, leading to better responsiveness and throughput.
* Additionally, **child processes enable fault tolerance** and resilience in the application. **If one child process encounters an error or crashes, it does not affect the operation of other child processes.** The master process can detect such failures and spawn new child processes to replace the ones that have failed, ensuring continuous availability and reliability of the application

**spawn()**: **This method in the child process module is used to spawn a new process,** providing more control over input/output streams and allowing for the execution of external commands or scripts.

exec(): **Another method in the child process module that spawns a shell and executes a command within that shell,** buffering any generated output. It is suitable for executing simple commands but may not provide as much control over input/output streams as spawn().

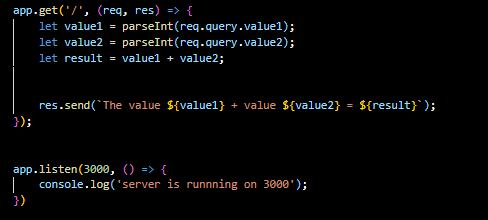
fork(): **This method is specifically designed for creating child processes that run Node.js modules.** It is **similar to spawn(), but it also establishes a communication channel between the parent and child processes** **using inter-process communication (IPC).**

Brief explanation

**Node.js is single threaded. No matter how many cores you have node only uses single core of your cpu. This is okey for IO operations, but if the code has long running and cpu intensive operations, your application struggles.**

**So node introduced cluster module, to solve this cluster module enables the creation of child process that run simultaneously. (also called workers)**

**Fetching value from the query and calculating.**

****

* **Query: Think of it like asking a question. Query parameters are added to the end of a URL after a ‘?’, like ‘?color=blue&size=large’. They're used to pass extra information to the server, like search terms or filters.**
* **Params: Params are like placeholders in the URL itself, marked by a ‘:’. For example, ‘/users/:id.’ They're used to capture specific values from the URL, like user IDs or product names.**

**Session**: A session is a **way to store user-specific information on the server side**. When a user visits a website, a unique session ID is created for that user. This ID is then stored as a cookie on the client-side, allowing the server to identify the user in subsequent requests. **Sessions are typically used to store sensitive data like user authentication status, shopping cart contents, or user preferences.**

**Cookie**: A cookie is a small piece of data sent from a website and stored on the user's device by the web browser. **Cookies can store information such as user preferences, login credentials, and shopping cart items.** They are commonly used for implementing features like persistent login sessions, tracking user behavior, and personalizing user experiences. **Cookies can have an expiration date, after which they are automatically deleted, or they can be set to expire when the browser session ends.**

**Difference between session storage and local storage**

**session storage**

1. **Data is stored for the duration of the page session**. It is cleared when the page session ends, which happens when the browser tab or window is closed.
2. **Data is specific to a single browser tab or window.** Different tabs or windows from the same origin cannot access each other’s session storage.
3. Typically has the same storage limits as local storage, but **it's mainly intended for temporary storage.**
4. **Ideal for storing temporary data that needs to be available for the duration of a page session**, like form inputs before submission or temporary state information.

**Local storage**

1. **Data is stored indefinitely, or until it is explicitly deleted by the user or the web application.** It persists even after the browser is closed and reopened.
2. **Data is shared across all tabs and windows from the same origin**. All instances of the browser can access the same data.
3. Usually has a **storage limit of about 5-10 MB per origin**, which is the same as session storage but designed for longer-term storage.
4. Suitable **for storing persistent data**, such as user preferences, themes, or other settings that need to be retained between sessions.

**Concurrency**

Concurrency in Node.js **refers to the ability to handle multiple tasks or operations simultaneously**, which is a crucial aspect of building efficient and high-performance applications. Node.js **achieves concurrency through its event-driven, non-blocking I/O model.**

**Key Concepts**

1. **Event-Driven Architecture**: Node.js uses an event loop to handle asynchronous events, such as I/O operations, without blocking the main thread.
2. **Non-Blocking I/O**: I/O operations (like reading files or querying a database) are executed without blocking the execution flow, allowing the application to continue processing other tasks.
3. **Event Loop**: The event loop manages the execution of asynchronous operations in phases (timers, pending callbacks, I/O events, etc.), enabling efficient task handling.
4. **Asynchronous Programming**: Node.js supports asynchronous programming via:

* **Callbacks**: Functions executed after an asynchronous operation completes.
* **Promises**: Objects representing the eventual completion or failure of an asynchronous operation.
* **Async/Await**: Syntactic sugar over promises for more readable asynchronous code.

**process.nextTick and setImmediate**

**Both process.nextTick and setImmediate are essential tools in Node.js for controlling the timing and order of callback execution, helping developers manage asynchronous operations more effectively.**

**process.nextTick**

* Executes immediately after the current operation completes.
* Higher priority over I/O and timer callbacks.
* Useful for deferring execution within the same phase of the event loop.

**setImmediate**

* Executes on the next iteration of the event loop, after I/O and timer events.
* Lower priority compared to process.nextTick.
* Useful for deferring execution to the next phase of the event loop.

**Buffer**

In Node.js, a Buffer is a **special way to handle raw binary data,** which is different from regular text data. **Buffers are fixed in size and designed for efficiency, making them ideal for tasks like reading and writing files, handling network data, and converting data formats**. They allow you to work directly with binary data, which is useful for performance-intensive operations. Buffers can be created from arrays, strings, or by allocating a specific amount of memory. Once created, you can read from or write to a buffer, and even manipulate its contents. Buffers are essential for efficiently managing binary data in Node.js applications.

**Crypto Module  
The `crypto` module in Node.js provides tools to secure your data. It allows you to encrypt and decrypt information, create hashes, and generate digital signatures**. This module helps protect data by ensuring it remains confidential and verifying its integrity and authenticity. **For example, you can use it to encrypt sensitive information like passwords, create unique hashes to check data integrity, and sign messages to confirm they come from a trusted source.** The `crypto` module is essential for building secure applications in Node.js.