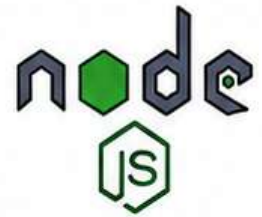


Chapter 12: Node.js Fundamentals

12.1 Introduction to Node.js

12.1.1 What is Node.js?

- * Node.js is a **JavaScript runtime** built on Chrome's V8 JavaScript engine.
- * It allows JavaScript to be used on the **server-side**.
- * Node.js is **open-source**, cross-platform and highly scalable.
- * It is used to build **fast, scalable network applications**.



12.1.2 Why Use Node.js?

- * **JavaScript Everywhere** – Use JavaScript for both frontend and backend.
- * **High Performance** – Built on an event-driven, non-blocking I/O model.
- * **Lightweight and Efficient** – Consumes less memory and handles many connections simultaneously.
- * **Large Ecosystem** – Huge collection of libraries in npm.

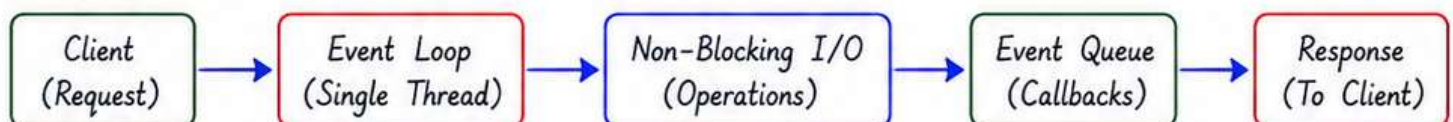


12.1.3 Features of Node.js

Event-Driven	Non-Blocking I/O	Single Threaded	Highly Scalable	Cross-Platform
				
Handles multiple requests using event loop mechanism.	Does not wait for API responses, improves performance.	Uses a single thread with event loop for handling requests.	Can handle thousands of connections concurrently.	Runs on Windows, Linux, macOS and other platforms.

12.1.4 Node.js Architecture

Node.js follows an **event-driven, non-blocking I/O** architecture.



Key Points:

- ✓ Node.js is a JavaScript runtime built for server-side development.
- ✓ It uses an event-driven, non-blocking I/O model.
- ✓ It is fast, scalable, lightweight and efficient.
- ✓ It has a rich ecosystem of libraries through npm.

npm

Node Package Manager
(Default for Node.js libraries)

Key Takeaway:

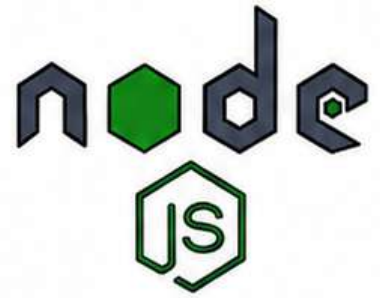


Node.js allows developers to build **fast, scalable** and **efficient** server-side applications using JavaScript.

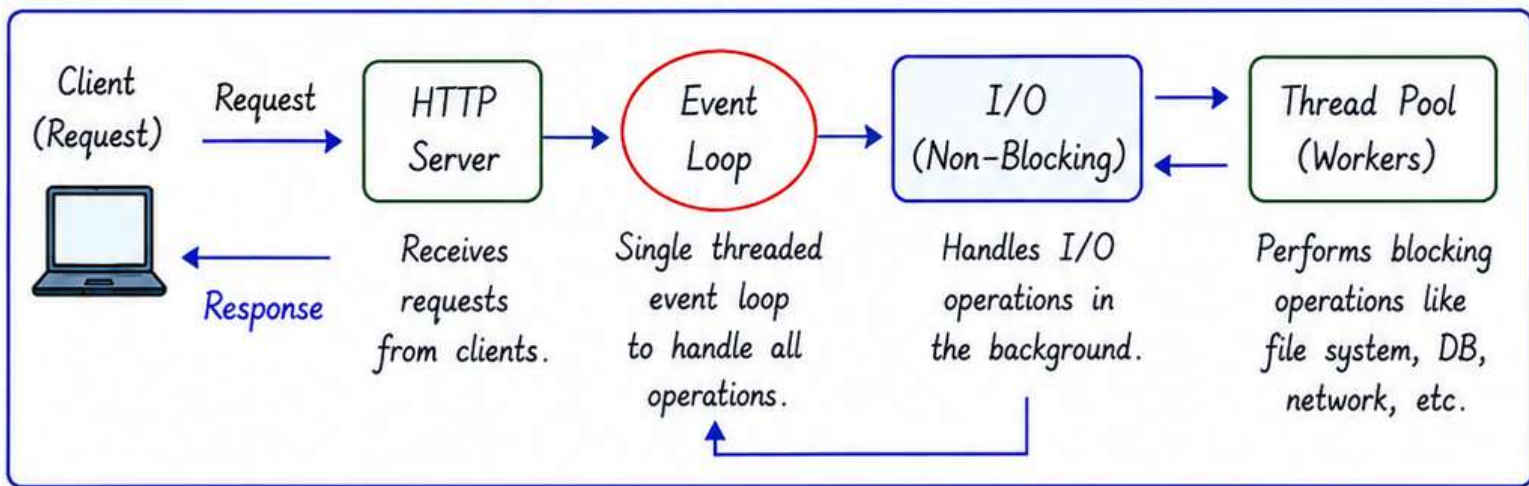
12.2 Node.js Architecture

12.2.1 Overview

- * Node.js is built on Chrome's **V8 JavaScript engine**.
- * It uses an **event-driven, non-blocking I/O** model.
- * Node.js architecture is designed to handle multiple connections concurrently with **high performance**.
- * It consists of the following main components.

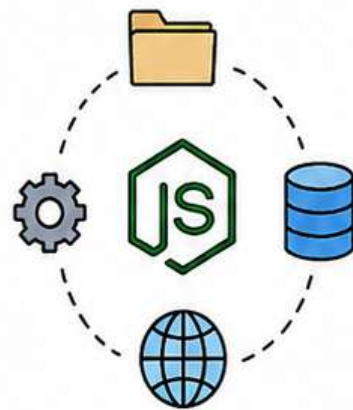


12.2.2 Architecture Diagram



12.2.3 How It Works?

- * A client sends a **request** to the Node.js application.
- * The **HTTP server** receives the request and hands it over to the **event loop**.
- * The event loop processes the request and initiates **non-blocking I/O** operations.
- * If a **blocking task** is required (e.g., reading a file, querying a database, network call), it is delegated to the **thread pool**.
- * Once the task is completed, the **result** is returned to the event loop.
- * The event loop sends the **response** back to the client.



Key Points:

- ✓ Node.js uses a **single threaded** event loop.
- ✓ It follows a **non-blocking I/O** model.
- ✓ Blocking operations are handled by the **thread pool**.
- ✓ This architecture allows Node.js to handle many connections **efficiently**.
- ✓ It is **lightweight, fast** and **highly scalable**.



Key Takeaway:



Node.js architecture is based on an event-driven, non-blocking I/O model that makes it **fast, scalable** and suitable for modern web applications.

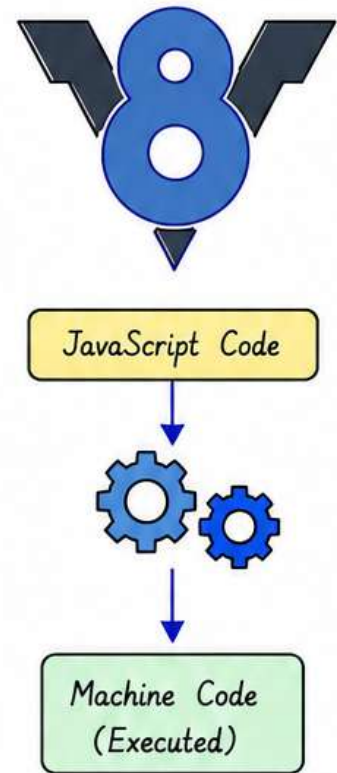
12.3 V8 Engine

12.3.1 What is V8 Engine?

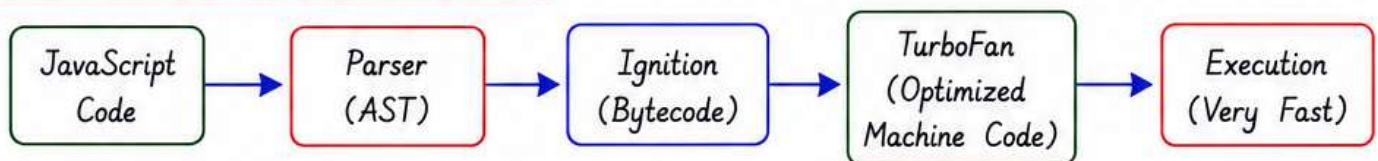
- * V8 is the high-performance JavaScript engine developed by **Google**.
- * It is written in **C++**.
- * V8 is used in **Chrome** and **Node.js**.
- * It compiles JavaScript code into **machine code** so that it can be executed very fast.

12.3.2 Key Components of V8 Engine

Component	Description
 Parser	Parses the JavaScript code and converts it into Abstract Syntax Tree (AST).
 Ignition Interpreter	Interprets the AST and generates bytecode . Executes the bytecode.
 TurboFan Compiler	Optimizes frequently used code and compiles it into highly optimized machine code .
 Memory Heap	Manages memory allocation for objects.
 Garbage Collector	Automatically reclaims memory that is no longer in use.

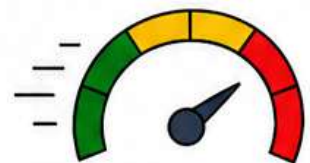


12.3.3 How V8 Engine Works?



Key Points:

- ✓ V8 is the JavaScript engine used in **Chrome** and **Node.js**.
- ✓ It compiles JavaScript into **optimized machine code**.
- ✓ It uses a just-in-time (JIT) **compilation** strategy.
- ✓ It has automatic memory management through **garbage collection**.
- ✓ This makes JavaScript execution very fast and efficient.



Key Takeaway:

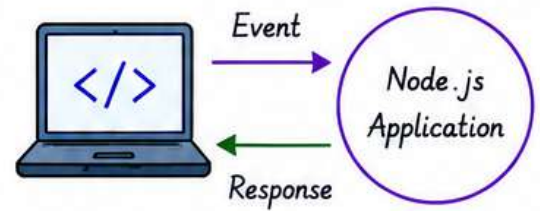


The V8 engine is the heart of Node.js. It makes JavaScript **fast** by compiling code, optimizing performance and managing memory **efficiently**.

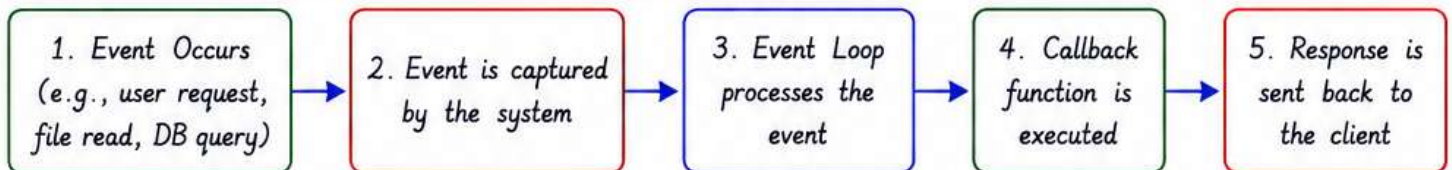
12.4 Event-Driven Architecture

12.4.1 What is Event-Driven Architecture?

- * Event-driven architecture is a design pattern where the flow of the program is determined by **events**.
- * In Node.js, almost **everything is an event**.
- * The application **waits for events** and **responds** when those events occur.
- * This makes Node.js highly **efficient** and **scalable**.

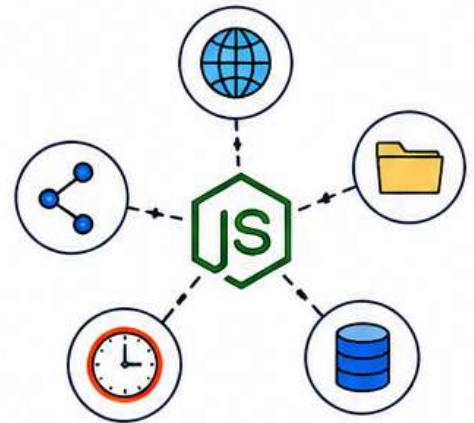


12.4.2 How Event-Driven Architecture Works?

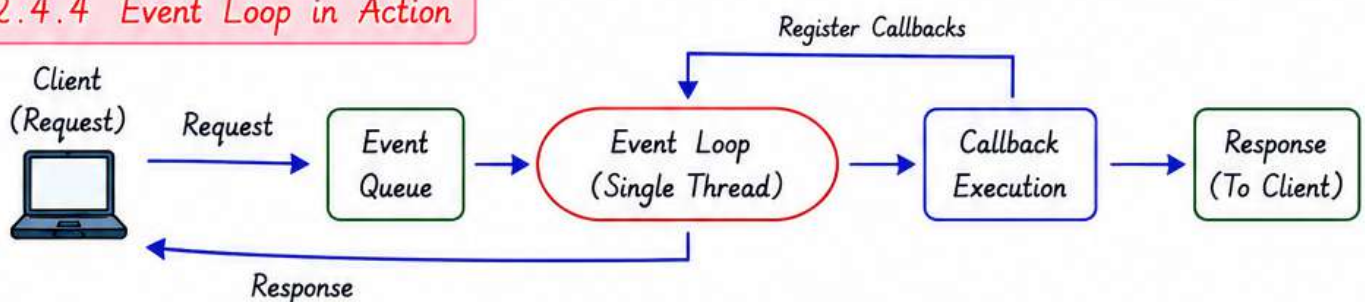


12.4.3 Common Events in Node.js

Event	Description
 Request	Triggered when a client sends an HTTP request.
 File Read / Write	Triggered when a file operation is completed.
 Database Query	Triggered when a database query is executed.
 Timer	Triggered by <code>setTimeout()</code> or <code>setInterval()</code> .
 Network Event	Triggered by network activities like connection, disconnection, data received, etc.



12.4.4 Event Loop in Action



Key Points:

- ✓ Node.js uses an **event-driven, non-blocking I/O** model.
- ✓ It uses a **single-threaded** event loop to handle multiple requests.
- ✓ **Callbacks** are executed when the events occur.
- ✓ This architecture improves **performance, scalability** and **real-time** capabilities.



Key Takeaway:

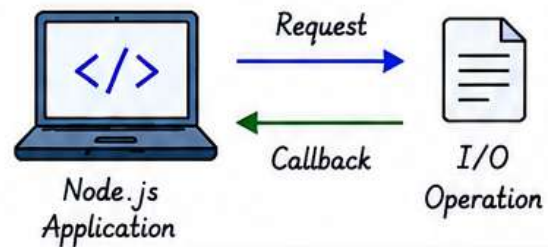


Event-driven architecture is the core of Node.js. It allows the application to handle **many events concurrently** with **high performance** using the event loop.

12.5 Non-Blocking I/O

12.5.1 What is Non-Blocking I/O?

- * Non-blocking I/O allows Node.js to perform I/O operations **without waiting** for the operation to complete.
- * Instead of blocking the program, Node.js **continues** to execute other tasks.
- * Once the I/O operation is completed, a **callback function** is executed.
- * This makes applications **faster, efficient** and **scalable**.

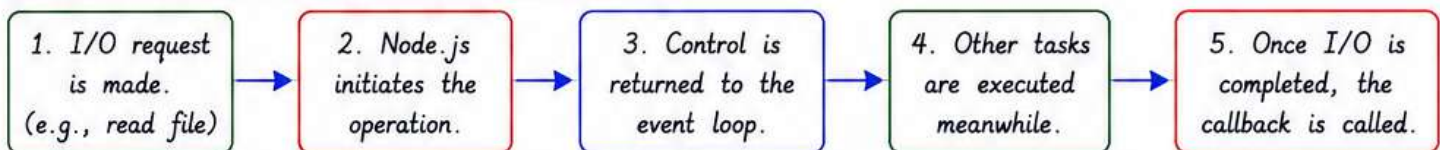


Node.js does not wait.
It continues with other tasks.

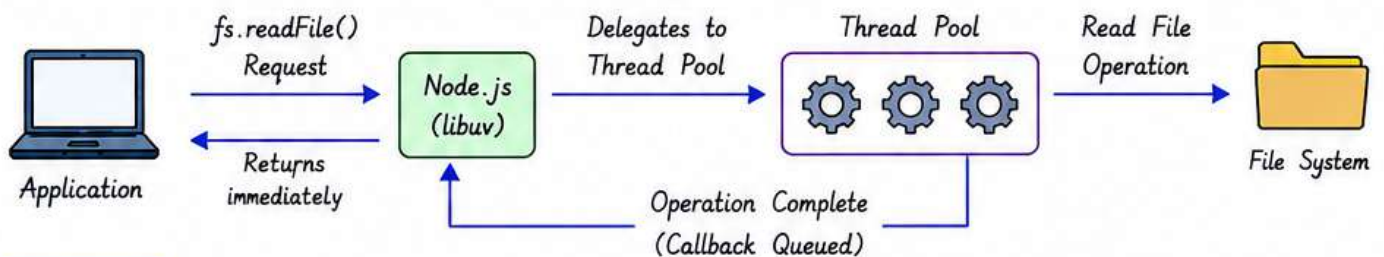
12.5.2 Blocking vs Non-Blocking I/O

Blocking I/O	Non-Blocking I/O
* The program waits for the I/O operation to complete. * Other tasks cannot be executed meanwhile. * Creates a bottleneck and reduces performance. * Suitable for small, simple applications.	* The program does not wait for the I/O operation to complete. * Other tasks can be executed meanwhile. * Improves performance and scalability. * Suitable for large, high-performance applications.

12.5.3 How Non-Blocking I/O Works?



12.5.4 Example: Read File (Non-Blocking)



Key Points:

- ✓ Non-blocking I/O allows Node.js to handle **many requests concurrently**.
- ✓ It uses the **event loop** and **thread pool** to perform I/O operations.
- ✓ The program **does not wait** for I/O operations to complete.
- ✓ **Callbacks** are executed once the operations are finished.
- ✓ This leads to better **performance** and **scalability**.

Example Code (Non-Blocking)

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

console.log('This line is executed first!');
```



Key Takeaway:

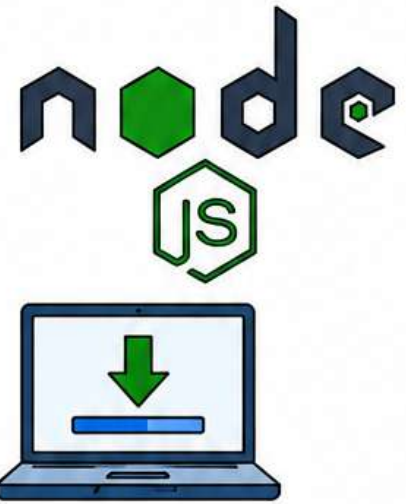


Non-blocking I/O is the core strength of Node.js. It allows the server to handle multiple requests simultaneously without waiting, making applications **fast, scalable** and **efficient**.

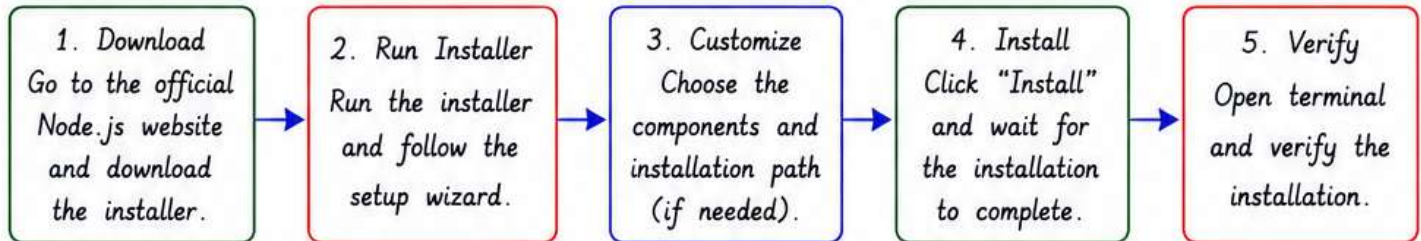
12.6 Node.js Installation

12.6.1 What is Node.js?

- * Node.js is a JavaScript runtime built on Chrome's **V8 engine**.
- * It allows you to run JavaScript on the **server side**.
- * Node.js comes with **npm** (Node Package Manager) to manage packages and dependencies.
- * It is **cross-platform** and **open source**.



12.6.2 Steps to Install Node.js



12.6.3 Verify the Installation

- * Open Command Prompt (Windows) or Terminal (macOS/Linux) and run the following commands:

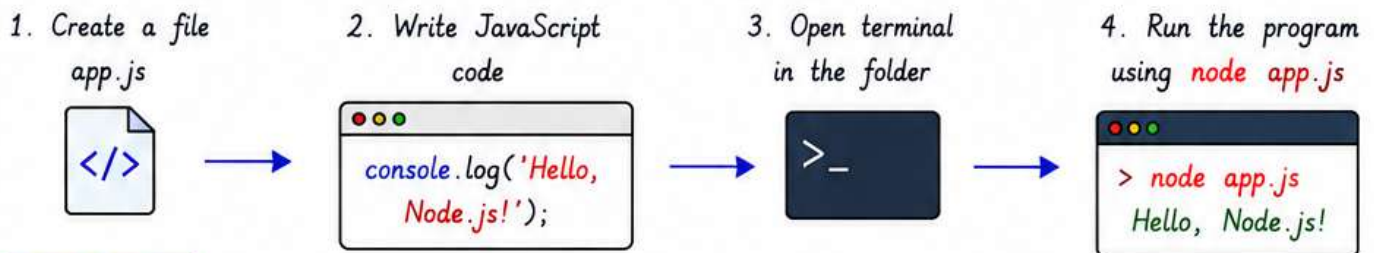
Command	Purpose	Example Output
<code>node -v</code>	Checks the installed Node.js version.	<pre>> v20.11.1</pre>
<code>npm -v</code>	Checks the installed npm version.	<pre>> 10.2.4</pre>

Note:

If you see version numbers, Node.js and npm are installed successfully.



12.6.4 Example: Run a Simple Node.js Program



Key Points:

- ✓ Node.js is built on Chrome's **V8 JavaScript engine**.
- ✓ It includes **npm** to manage packages and dependencies.
- ✓ Node.js is **cross-platform** and **open source**.
- ✓ Always verify the installation using `node -v` and `npm -v`.
- ✓ You can run JavaScript files directly using the `node` command.



Key Takeaway:

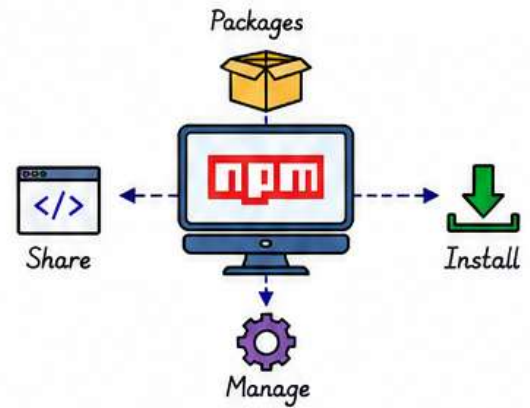


Node.js installation is simple and quick. Once installed, you can run JavaScript on the server side, manage packages with **npm** and build powerful applications.

12.7 npm

12.7.1 What is npm?

- * **npm** (Node Package Manager) is the default package manager for **Node.js**.
- * It is used to **install**, **share** and **manage** packages (modules) written in JavaScript.
- * It is the largest ecosystem of **open source** libraries in the world.
- * npm comes **bundled** with Node.js installation.

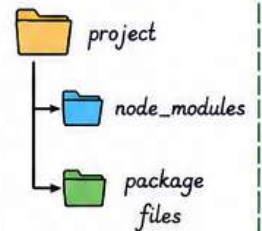


12.7.2 Common npm Commands

Command	Description	Example
<code>npm init</code>	Initializes a new npm project and creates package.json file.	<code>npm init</code>
<code>npm install <package></code>	Installs a package and adds it to dependencies.	<code>npm install express</code>
<code>npm install</code>	Installs all dependencies listed in package.json.	<code>npm install</code>
<code>npm uninstall <package></code>	Uninstalls a package.	<code>npm uninstall express</code>
<code>npm list</code>	Lists all installed packages.	<code>npm list</code>
<code>npm update</code>	Updates all installed packages.	<code>npm update</code>
<code>npm search <package></code>	Searches for a package in the npm registry.	<code>npm search lodash</code>
<code>npm -v</code>	Checks the npm version.	<code>npm -v</code>
<code>npm help</code>	Shows help information.	<code>npm help</code>

Note:

Packages installed locally are stored in the **node_modules** folder.



12.7.3 package.json File

- * Every npm project has a **package.json** file.
- * It contains metadata about the project and its dependencies.
- * It includes project name, version, scripts, dependencies, etc.

Note:

Dependencies listed in **package.json** are installed inside the **node_modules** folder.

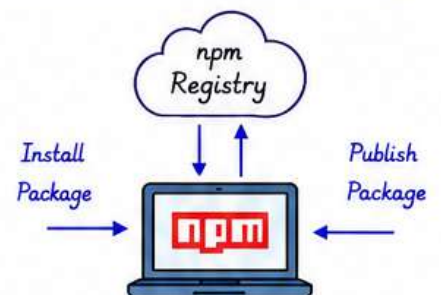
```
{
  "name": "my-app",
  "version": "1.0.0",
  "description": "A sample Node.js application",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

12.7.4 Local vs Global Installation

Local Installation	Global Installation
* Installed in the current project directory. * Listed in package.json under dependencies. * Used by the project only. * Example: <code>npm install express</code>	* Installed globally on the system. * Not listed in package.json. * Can be used in any project. * Example: <code>npm install -g nodemon</code>

Key Points:

- ✓ **npm** is the default package manager for **Node.js**.
- ✓ It helps to **install**, **manage** and **share** JavaScript packages.
- ✓ **package.json** stores project metadata and dependencies.
- ✓ Use **npm** commands to simplify development and dependency management.
- ✓ Always run `npm install` to install project dependencies.



Key Takeaway:

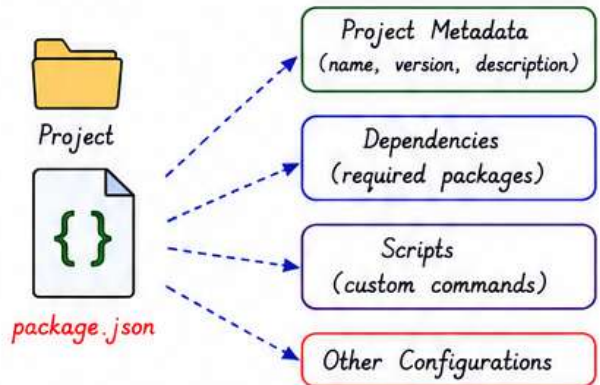


npm is a powerful tool that makes it **easy** to **install**, **manage** and **share** packages. It simplifies dependency management and improves development **productivity**.

12.8 package.json

12.8.1 What is package.json?

- * **package.json** is a configuration file used in **Node.js** projects.
- * It contains **metadata** about the project and its **dependencies**.
- * It is automatically created when you run **npm init**.
- * It helps **npm** to manage project information, **scripts** and **packages**.
- * It is the heart of every **Node.js** project.



12.8.2 Structure of package.json

```
{
  "name": "my-app",
  "version": "1.0.0",
  "description": "A sample Node.js application",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "nodemon": "^2.0.22"
  },
  "author": "John Doe",
  "license": "ISC"
}
```

Field	Description
name	The name of the project.
version	The version of the project.
description	A short description of the project.
main	The entry point file of the application.
scripts	Custom commands to run tasks.
dependencies	List of packages required for the project to run in production.
devDependencies	List of packages required for development and testing.
author	The author of the project.
license	The license under which the project is distributed.

12.8.3 How to Create package.json?

There are two ways to create a **package.json** file:

1. Using **npm init** (Interactive)

- * Open terminal in your project directory.
- * Run the command: **npm init**
- * Answer the questions, **npm** will create **package.json**.

```
$ npm init
This utility will walk you through creating a package.json file.
Press ^C at any time to quit.
package name: (my-app)
version: (1.0.0)
description: A sample Node.js application
entry point: (index.js)
test command:
git repository:
keywords:
author: John Doe
license: (ISC)
```

2. Using **npm init -y** (Default)

- * Open terminal in your project directory.
- * Run the command: **npm init -y**
- * Creates a default **package.json** file with default values.

```
$ npm init -y
Wrote to /path/my-app/package.json:

{
  "name": "my-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```



Creates
package.json

12.8.4 Key Points

- ✓ **package.json** stores all important information about your project.
- ✓ It helps in **dependency management** and **script execution**.
- ✓ It is automatically used by **npm** to install packages and run scripts.
- ✓ Always include **package.json** in your project and in version control (**git**).



Key Takeaway:

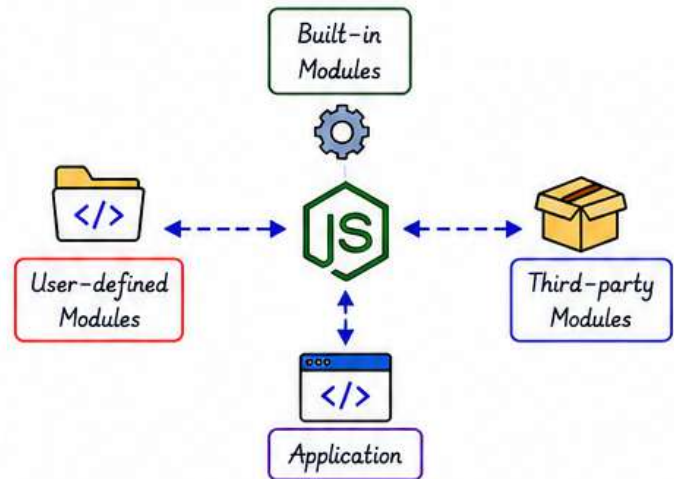


package.json is a must-have file in every **Node.js** project. It contains project metadata, scripts, and dependencies that help **npm** manage everything **efficiently**.

12.9 Modules

12.9.1 What is a Module?

- * A module is a **reusable** piece of JavaScript code.
- * Node.js has a **modular** architecture.
- * Modules help in **organizing** code into separate files.
- * They improve code **reusability**, **maintainability** and **scalability**.
- * Node.js supports both **built-in** modules and **user-defined** modules.



12.9.2 Types of Modules

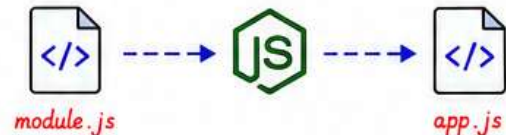
1. Built-in Modules

- * Provided by Node.js.
- * No need to install separately.
- * Example: `fs`, `path`, `http`, `os`, `url`, etc.



2. User-defined Modules

- * Created by the developers.
- * Used to split the code into multiple files.
- * Export from one file and use in another.



3. Third-party Modules

- * Created by other developers and published on **npm**.
- * Need to install using `npm install <module>`.
- * Example: `express`, `lodash`, `mongoose`, `chalk`, etc.



12.9.3 How Modules Work in Node.js?

Modules in Node.js follow the concept of **exporting** from one file and **importing** in another.

1. Export (from module file)

Use `module.exports` or `exports` to export values, functions or objects.

```
// math.js
function add(a, b) {
  return a + b;
}
module.exports = add;
```

2. Import (in another file)

Use `require()` to import the module and use it.

```
// app.js
const add = require('./math');
console.log(add(5, 3)); // 8
```

3. Use

The imported module can be used in the application.

```
> node app.js
8
```

12.9.4 Key Points

- ✓ Modules help in writing **clean**, **organized** and **reusable** code.
- ✓ **Built-in** modules are part of Node.js.
- ✓ **User-defined** modules are created by developers.
- ✓ **Third-party** modules are available via **npm**.
- ✓ Use `module.exports` to export and `require()` to import modules.



Key Takeaway:

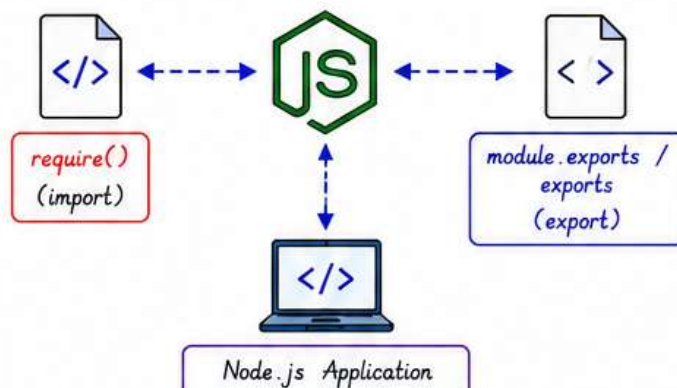


Modules are the building blocks of Node.js applications. They promote code **reusability**, better **organization** and easier **maintenance**.

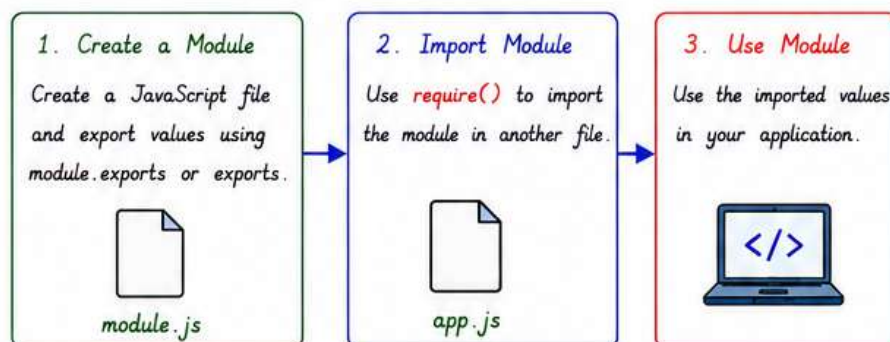
12.10 CommonJS

12.10.1 What is CommonJS?

- * CommonJS is a **module system** used in **Node.js**.
- * It allows you to organize code into separate files (modules) and reuse them.
- * It uses **require()** to import modules and **module.exports** or **exports** to export values.
- * CommonJS is the default module system in Node.js (prior to ES Modules).



12.10.2 How CommonJS Works



12.10.3 Export in CommonJS

Syntax	Description
<code>module.exports = value;</code>	Export a single value (object, function, string, etc.).
<code>module.exports = { a: 1, b: 2 };</code>	Export an object with multiple values.
<code>exports.name = value;</code>	Export multiple values using exports object.
<code>exports.name1 = value1; exports.name2 = value2;</code>	Another way to export multiple values.

12.10.4 Example: Exporting a Module

```
1. module.js
// Export a function
function add(a, b) {
  return a + b;
}
// Export using module.exports
module.exports = add;
```

```
2. app.js
// Import the module
const add = require('./module');

// Use the module
const result = add(5, 3);
console.log(result); // 8
```

```
3. Output (Terminal)
> node app.js
8
```

12.10.5 Exporting Multiple Values

```
1. math.js
// Export multiple values using exports
exports.add = function(a, b) {
  return a + b;
};
exports.sub = function(a, b) {
  return a - b;
};
```

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

console.log(math.add(5, 3)); // 8
console.log(math.sub(5, 3)); // 2
```

```
3. Output (Terminal)
> node app.js
8
2
```

12.10.6 Key Points

- ✓ CommonJS is the default module system in Node.js.
- ✓ Use **require()** to import modules.
- ✓ Use **module.exports** or **exports** to export values.
- ✓ Modules help in writing **modular**, **reusable** and **maintainable** code.
- ✓ CommonJS is **synchronous** and works on the **server side**.



Key Takeaway:

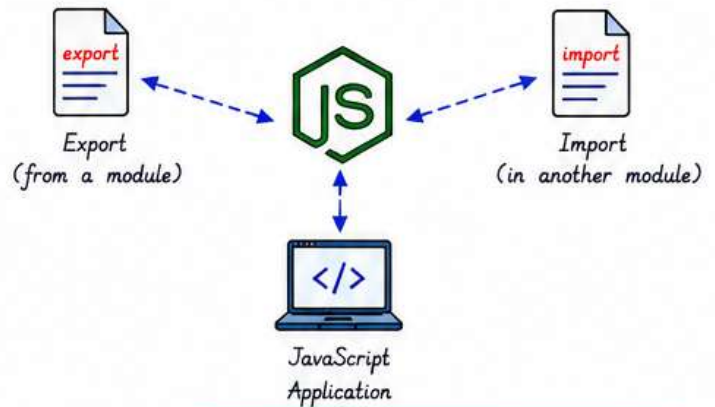


CommonJS is the traditional module system in Node.js.
It makes code modular and reusable using **require()** and **module.exports**.

12.11 ES Modules

12.11.1 What are ES Modules?

- * ES Modules (ECMAScript Modules) are the **official** module system in JavaScript.
- * They were introduced in **ES6 (ES2015)**.
- * They use **import** and **export** keywords.
- * They are **static**, **standardized** and work in both browsers and Node.js.
- * ES Modules are the **future** of JavaScript modules.



12.11.2 ES Modules vs CommonJS

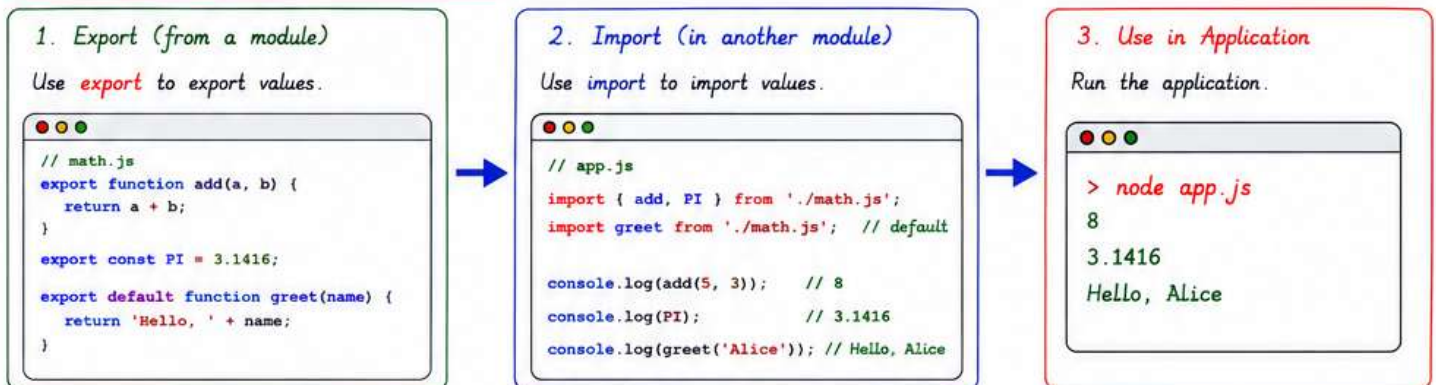
Feature	CommonJS	ES Modules
Syntax	<code>require()</code> , <code>module.exports</code>	<code>import</code> , <code>export</code>
Standard	Not officially standardized	Official ECMAScript standard
Loading	Dynamic (runtime)	Static (compile-time)
Browser Support	Limited (needs bundlers)	Native support in modern browsers
Default in Node.js	Yes (older versions)	Yes (with "type": "module" or .mjs)

12.11.3 Why Use ES Modules?

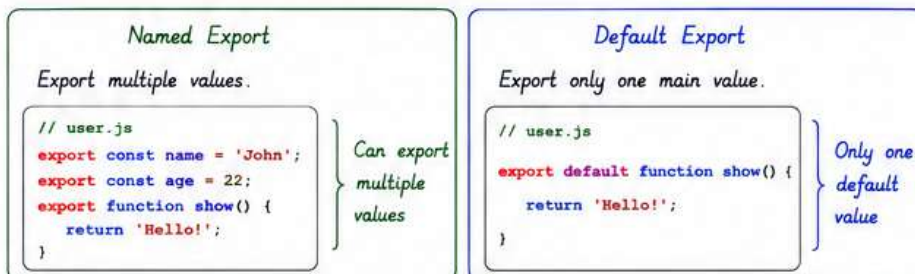
- * Static structure allows better **optimization**.
- * Better support for **tree shaking**.
- * Works **natively** in browsers.
- * Encourages **cleaner** and more maintainable code.
- * Future-proof and **standard** approach.



12.11.4 How ES Modules Work?



12.11.5 Named Export vs Default Export



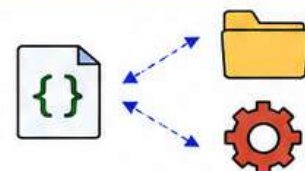
12.11.6 File Extensions & Config

- * Use **.mjs** extension OR
- * Add **"type": "module"** in `package.json`
- * Then use **.js** files

```
{
  "name": "my-app",
  "type": "module",
  "version": "1.0.0"
}
```

12.11.7 Key Points

- ✓ ES Modules use **import** and **export**.
- ✓ They are the **official standard** in JavaScript (ES6).
- ✓ Static structure improves **performance** and **optimization**.
- ✓ Works in both **browsers** and **Node.js**.
- ✓ Better for large applications and **modern** development.



Key Takeaway:

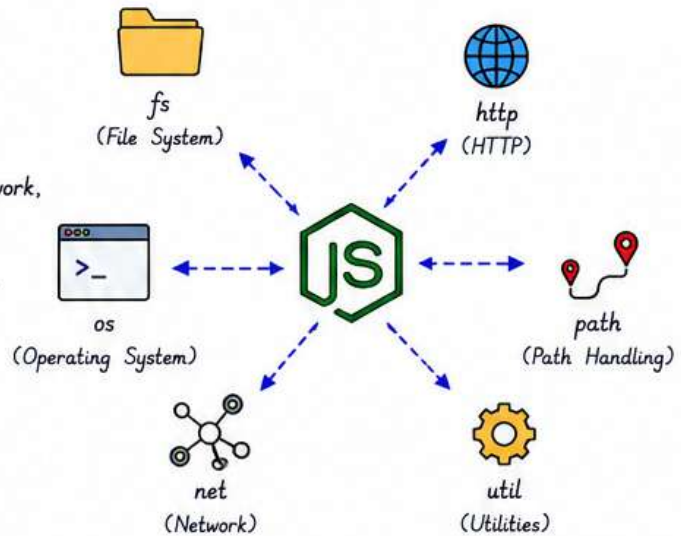


ES Modules are the modern, standard way to write modular JavaScript.
They make code **cleaner**, more **efficient** and **future-ready**.

12.12 Built-in Modules

12.12.1 What are Built-in Modules?

- * Built-in modules are included with **Node.js**.
- * No need to install them using **npm**.
- * They provide **core functionality** for file system, network, HTTP, path handling, OS information, etc.
- * We can use them by directly **requiring** or **importing**.
- * They are **fast**, **reliable** and **well-tested**.



12.12.2 Common Built-in Modules

Module	Description	Example Use
fs	Provides methods to interact with the file system.	Read/Write files, create folders, delete files.
path	Helps in working with file and directory paths.	Join paths, get file extension, resolve paths.
http	Allows creating HTTP servers and making HTTP requests.	Build web servers, handle requests.
os	Provides information about the OS.	CPU details, memory info, platform, uptime.
url	Parses URL strings and works with URLs.	Extract parts of a URL.
events	Provides event-driven programming capabilities.	Create and manage events.
stream	Handles streaming data.	Read/Write large data efficiently.
util	Provides utility functions.	Formatting, inspection, callbacks.
crypto	Provides cryptographic functionality.	Hashing, encryption, decryption.
buffer	Handles binary data directly.	Work with binary data.

12.12.3 Using Built-in Module (CommonJS)

Syntax:

```
const moduleName = require('module-name');
```

Example:

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

console.log(os.platform());
console.log(path.join('a', 'b', 'c'));
```

12.12.4 Using Built-in Module (ES Modules)

Syntax:

```
import moduleName from 'module-name';
```

Example:

```
import fs from 'fs';
import os from 'os';
import path from 'path';

console.log(os.platform());
console.log(path.join('a', 'b', 'c'));
```

12.12.5 Examples of Built-in Modules

1. fs Module (File System)

```
const fs = require('fs');
// Read a file
const data = fs.readFileSync('test.txt', 'utf8');
console.log(data);
```

2. path Module (Path Handling)

```
const path = require('path');
const filePath = path.join('folder', 'sub', 'file.txt');
console.log(filePath);
```

3. os Module (Operating System)

```
const os = require('os');
console.log('Platform:', os.platform());
console.log('CPUs:', os.cpus().length);
console.log('Free Mem:', os.freemem());
```

4. http Module (HTTP Server)

```
const http = require('http');
const server = http.createServer((req, res) => {
  res.end('Hello from Node.js');
});
server.listen(3000);
console.log('Server running on port 3000');
```

12.12.6 Key Points

- ✓ Built-in modules come with Node.js.
- ✓ No installation **required**.
- ✓ Use **require()** in CommonJS and **import** in ES Modules.
- ✓ Provide essential features like file handling, OS info, HTTP, path, etc.
- ✓ They are **efficient** and optimized for performance.



Key Takeaway:



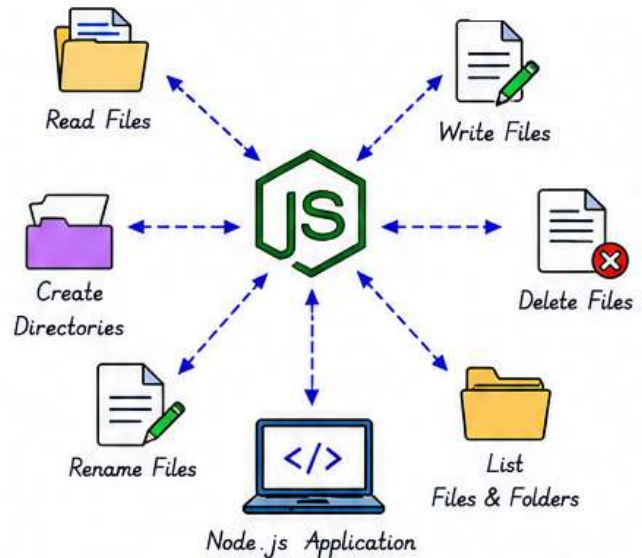
Built-in modules are **ready-to-use** modules provided by Node.js.

They **save time**, reduce dependencies and provide **reliable** core features for applications.

12.13 File System

12.13.1 What is File System (fs) Module?

- * The **fs** (File System) module in Node.js allows interaction with the file system on your computer.
- * It provides methods to **read, write, update, delete, rename** and manage files and directories.
- * We can use **fs** module to perform both **synchronous** (blocking) and **asynchronous** (non-blocking) file operations.
- * It is a **built-in** module, no need to install.



12.13.2 Common fs Operations (Asynchronous)

Operation	Method	Description
Read File	<code>fs.readFile()</code>	Reads the content of a file.
Write File	<code>fs.writeFile()</code>	Writes data into a file (overwrites).
Append File	<code>fs.appendFile()</code>	Appends data to a file.
Delete File	<code>fs.unlink()</code>	Deletes a file.
Rename File	<code>fs.rename()</code>	Renames or moves a file.
Create Folder	<code>fs.mkdir()</code>	Creates a new directory.
Read Directory	<code>fs.readdir()</code>	Reads the contents of a directory.
Delete Folder	<code>fs.rmdir()</code>	Deletes an empty directory.

12.13.3 Example: Read and Write File

1. Write to File (writeFile)

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

const data = 'Hello, Node.js!';

fs.writeFile('example.txt', data, (err) => {
  if (err) {
    console.log('Error:', err);
  } else {
    console.log('File written successfully!');
  }
});
```

2. Read from File (readFile)

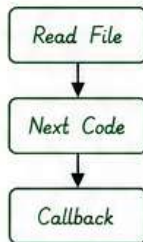
```
const fs = require('fs');

fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.log('Error:', err);
  } else {
    console.log('File Content:');
    console.log(data);
  }
});
```

12.13.4 Synchronous vs Asynchronous

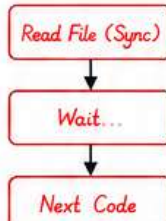
1. Asynchronous (Non-blocking)

- * Default methods in **fs** module.
- * Uses **callbacks**.
- * Does not block the next code from executing.
- * Better for performance.



2. Synchronous (Blocking)

- * Methods with **Sync** suffix.
- * Blocks the next code until the operation is complete.
- * Simple to write but not efficient for large applications.



12.13.5 Examples of Other fs Operations

1. Append to File

```
fs.appendFile('example.txt',
  '\nThis is appended text.',
  (err) => {
    if (err) console.log(err);
    else console.log('Appended!');
  });
```



2. Delete File

```
fs.unlink('example.txt', (err) => {
  if (err) console.log(err);
  else console.log('File deleted!');
});
```



3. Rename File

```
fs.rename('old.txt', 'new.txt', (err) => {
  if (err) console.log(err);
  else console.log('Renamed!');
});
```



4. Create Directory

```
fs.mkdir('myFolder', (err) => {
  if (err) console.log(err);
  else console.log('Folder created!');
});
```



5. Read Directory

```
fs.readdir('.', (err, files) => {
  if (err) console.log(err);
  else console.log(files);
});
```



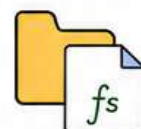
6. Delete Directory

```
fs.rmdir('myFolder', (err) => {
  if (err) console.log(err);
  else console.log('Folder deleted!');
});
```



12.13.6 Key Points

- ✓ **fs** module allows working with files and directories in Node.js.
- ✓ Supports both synchronous and asynchronous operations.
- ✓ Async methods are recommended for better performance.
- ✓ Common operations: read, write, append, delete, rename, create folder, read folder.
- ✓ Always handle errors using callbacks or `try...catch` (for sync).



Key Takeaway:



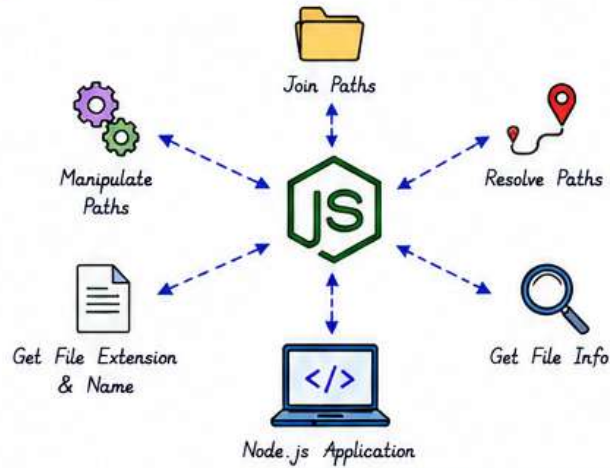
The **fs** module is essential for file handling in Node.js.

Use **asynchronous** methods for non-blocking operations and better scalability.

12.14 Path Module

12.14.1 What is Path Module?

- * The **path** module provides utilities for working with file and directory paths.
- * It helps in handling and transforming paths in a **cross-platform** way.
- * It is a **built-in** module, no need to install.
- * Works on **Windows**, **macOS** and **Linux**.



12.14.2 Common path Methods

Method	Description	Example
<code>path.join(...paths)</code>	Joins all given path segments.	<code>path.join('a', 'b', 'c')</code>
<code>path.resolve(...paths)</code>	Resolves to an absolute path.	<code>path.resolve('folder', 'file.txt')</code>
<code>path.normalize(path)</code>	Normalizes a path (remove <code>..</code> , <code>.</code>)	<code>path.normalize('a/./b/c/../../d')</code>
<code>path.basename(path, ext)</code>	Returns the last portion of a path.	<code>path.basename('/a/b/c/file.txt')</code>
<code>path.dirname(path)</code>	Returns the directory name of a path.	<code>path.dirname('/a/b/c/file.txt')</code>
<code>path.extname(path)</code>	Returns the extension of a file.	<code>path.extname('file.txt')</code>
<code>path.parse(path)</code>	Returns an object with path parts.	<code>path.parse('/a/b/c/file.txt')</code>
<code>path.format(pathObject)</code>	Creates a path string from an object.	<code>path.format({dir: '/a/b', base: 'file.txt'})</code>
<code>path.isAbsolute(path)</code>	Checks if the path is absolute.	<code>path.isAbsolute('/a/b')</code>
<code>path.relative(from, to)</code>	Returns the relative path from 'from' to 'to'.	<code>path.relative('/a/b', '/a/b/c/d')</code>
<code>path.sep</code>	Returns the platform specific separator.	<code>'/'</code> on Linux/macOS, <code>'\\'</code> on Windows
<code>path.delimiter</code>	Returns the delimiter used in PATH env.	<code>':'</code> on Linux/macOS, <code> ';' </code> on Windows

12.14.3 Example: path.join() & path.resolve()

Using path.join()

```
const path = require('path');
const p = path.join('folder', 'sub', 'file.txt');
console.log(p);
// Output: folders\sub\file.txt (Windows)
//         folder/sub/file.txt (Linux/macOS)
```

Using path.resolve()

```
const path = require('path');
const p = path.resolve('folder', 'sub', 'file.txt');
console.log(p);
// Output: C:\Users\user\folder\sub\file.txt
//         /home/user/folder/sub/file.txt
```

12.14.4 Examples of Other path Methods

1. basename()

```
const path = require('path');
console.log(path.basename('/a/b/c/file.txt'));
// Output: file.txt

console.log(path.basename('/a/b/c/file.txt', '.txt'));
// Output: file
```



2. dirname()

```
const path = require('path');
console.log(path.dirname('/a/b/c/file.txt'));
// Output: /a/b/c
```



3. extname()

```
const path = require('path');
console.log(path.extname('index.html'));
// Output: .html
```



4. parse()

```
const path = require('path');
const obj = path.parse('/a/b/c/file.txt');
console.log(obj);
/* Output:
{
  root: '/',
  dir: '/a/b/c',
  base: 'file.txt',
  ext: '.txt',
  name: 'file'
} */
```



5. format()

```
const path = require('path');
const p = path.format({
  dir: '/a/b/c',
  base: 'file.txt'
});
console.log(p);
// Output: /a/b/c/file.txt
```



6. isAbsolute()

```
const path = require('path');
console.log(path.isAbsolute('/a/b'));
// Output: true
console.log(path.isAbsolute('a/b'));
// Output: false
```



7. relative()

```
const path = require('path');
const rel = path.relative('/a/b', '/a/b/c/d/file.txt');
console.log(rel);
// Output: c/d/file.txt (Windows)
//         c/d/file.txt (Linux/macOS)
```

`/a/b` → `c/d/file.txt`

8. normalize()

```
const path = require('path');
console.log(path.normalize('a/./b/c/../../d'));
// Output: a/b/d (Linux/macOS)
//         a\b\d (Windows)
```



12.14.5 Key Points

- ✓ path module helps in manipulating file and directory paths.
- ✓ It ensures cross-platform compatibility.
- ✓ Use **path.join()** to join multiple path segments.
- ✓ Use **path.resolve()** to get absolute path.
- ✓ Methods like **basename()**, **dirname()**, **extname()**, **parse()** are very useful.
- ✓ Always prefer **path** methods over manual string operations for paths.



Key Takeaway:

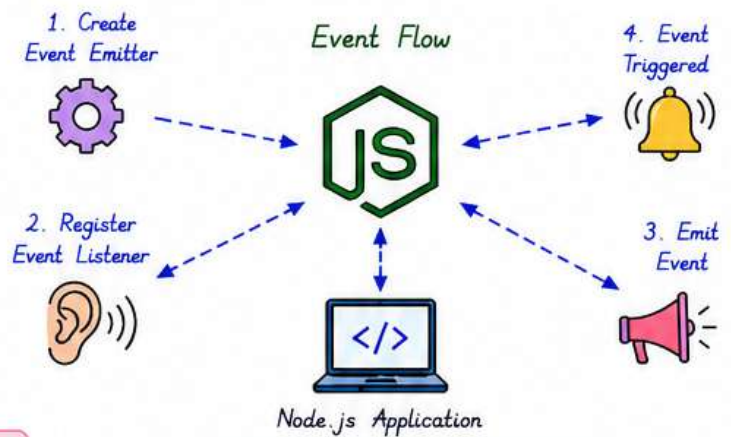


The **path** module in Node.js makes working with file and directory paths **easy** and **reliable**. It avoids **platform-specific issues** and improves code **portability**.

12.15 Events

12.15.1 What are Events?

- * Events are actions or occurrences that happen in a Node.js application.
- * The **events** module allows us to create, emit and listen for events.
- * It follows the **EventEmitter** class.
- * It supports **event-driven, asynchronous** programming.
- * Used in many Node.js core modules like **http, fs, stream, net**, etc.



12.15.2 Common EventEmitter Methods

Method	Description	Example
<code>on(event, listener)</code>	Registers a listener for the specified event.	<code>emitter.on('data', callback)</code>
<code>emit(event, ...args)</code>	Emits an event, invoking all attached listeners.	<code>emitter.emit('data', arg1, arg2)</code>
<code>once(event, listener)</code>	Registers a listener that is invoked only once.	<code>emitter.once('end', callback)</code>
<code>off(event, listener)</code>	Removes a specific listener.	<code>emitter.off('data', callback)</code>
<code>removeAllListeners(event)</code>	Removes all listeners for the specified event or all events.	<code>emitter.removeAllListeners('data')</code>
<code>listeners(event)</code>	Returns an array of listeners for the event.	<code>emitter.listeners('data')</code>
<code>eventNames()</code>	Returns an array of all event names.	<code>emitter.eventNames()</code>

12.15.3 Example: Basic Event

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

// Register event listener
emitter.on('greet', (name) => {
  console.log('Hello, ' + name + '! 🗣️');
});

// Emit event
emitter.emit('greet', 'Alice');
emitter.emit('greet', 'Bob');
```

Output:
Hello, Alice! 🗣️
Hello, Bob! 🗣️

12.15.4 Example: once()

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

emitter.once('login', () => {
  console.log('User logged in (this will run once)');
});

emitter.emit('login'); // First time
emitter.emit('login'); // Second time (will not run)
```

Output:
User logged in (this will run once)

12.15.5 Example: Removing Listeners

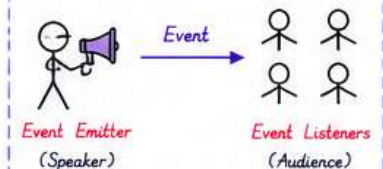
```
const EventEmitter = require('events');
const emitter = new EventEmitter();

function handler() {
  console.log('Data received');
}

emitter.on('data', handler);
emitter.emit('data'); // Data received
emitter.off('data', handler); // Remove listener
emitter.emit('data'); // (No output)
```

Output:
Data received

Real-life Analogy



The speaker announces an event, and all the listeners (audience) react to it.

12.15.6 Common Events in Node.js Core Modules

Module	Common Events	Description	Example Use
http	request, response, error	Triggered on HTTP request, response or error.	Create web servers. 🌐
fs	open, close, read, write, error	Triggered on file system operations.	Read/Write files. 📁
stream	data, end, error, close	Triggered on stream data flow.	Handle streaming data. 🌊
net	connection, data, error, close	Triggered on network events.	Network communication. 🌐
process	exit, uncaughtException, warning	Triggered on process lifecycle events.	Handle process level events. ⚙️

12.15.7 Key Points

- ✓ Events are the heart of Node.js event-driven architecture.
- ✓ Use **EventEmitter** to create and manage events.
- ✓ Use **on()** to listen, **emit()** to trigger events.
- ✓ **once()** is useful when the listener should run only once.
- ✓ Always remove listeners when no longer needed.
- ✓ Many core modules in Node.js use the Events API.
- ✓ Events help in writing scalable and responsive applications.



Key Takeaway:



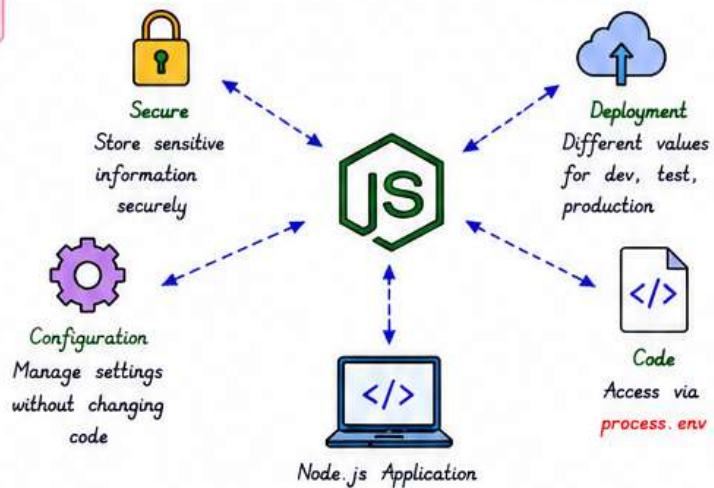
The Events module in Node.js allows your application to react to things that happen. It makes your code **asynchronous, modular** and **easy to extend**.



12.16 Environment Variables

12.16.1 What are Environment Variables?

- * Environment variables are **key-value pairs** used to store configuration outside the code.
- * They are useful for managing **sensitive information** (like API keys, passwords), configuration settings, and environment specific data.
- * Node.js provides access to environment variables via the **process.env** object.
- * They help keep code **secure, flexible** and **easy to deploy**.



12.16.2 How Environment Variables Work

Step	Description
1. Set	Define environment variables in your system, .env file or hosting platform.
2. Access	Access variables in code using process.env .
3. Use	Use the values in your application logic.
4. Secure	Keep sensitive data out of codebase.

12.16.3 Accessing Environment Variables in Node.js

```
// Access environment variables
console.log(process.env.PORT);           // 3000 (example)
console.log(process.env.DB_HOST);        // localhost (example)

const apiKey = process.env.API_KEY;
console.log('API Key:', apiKey);         // your_api_key
```

12.16.4 Using .env File

- * **.env** file is used to store environment variables in **key-value** format.
- * It should be placed in the root directory of your project.
- * Do not commit **.env** file to Git. Add it to **.gitignore**.

.env file (example)

```
PORT=3000
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=secret123
API_KEY=your_api_key_here
```

12.16.5 Example: Using dotenv Package

```
// npm install dotenv
require('dotenv').config(); // Load .env file

console.log('Port:', process.env.PORT);
console.log('DB Host:', process.env.DB_HOST);
console.log('API Key:', process.env.API_KEY);
// Now you can use these variables in your code
```

12.16.6 Ways to Set Environment Variables

Method	Example	Description
1. System Wide	Set in OS environment variables.	Available for all applications on the system.
2. Command Line	PORT=3000 node app.js	Set temporarily for a single session.
3. In Code	process.env.PORT = '3000';	Set inside the code (not recommended for sensitive data).
4. .env File	PORT=3000	Set in .env file and load using dotenv .

12.16.7 Example Usage in Application

```
require('dotenv').config();

const port = process.env.PORT || 5000; // default port
const dbHost = process.env.DB_HOST;

console.log('Server running on port ${port}');
console.log('Database host: ${dbHost}');
```

Output (example):

```
Server running on port 3000
Database host: localhost
```

12.16.8 Key Points

- ✓ Environment variables store configuration outside the code.
- ✓ Access them in Node.js using **process.env**.
- ✓ Use **.env** file and **dotenv** for local development.
- ✓ Never hardcode sensitive information in your code.
- ✓ Environment variables make applications secure and flexible.



Key Takeaway:



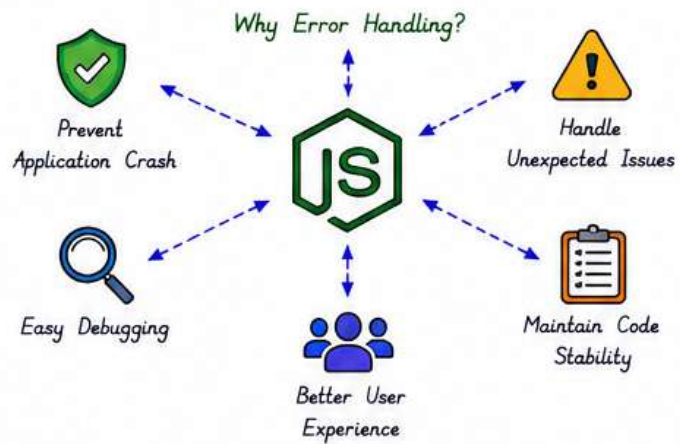
Environment variables help you manage configuration and sensitive data **securely**. They make your Node.js applications more **maintainable, portable** and **production-ready**.



12.17 Error Handling

12.17.1 What is Error Handling?

- * Errors can occur during the execution of a program because of invalid input, failed operations, network issues, etc.
- * Error handling allows us to **gracefully** handle these errors without crashing the application.
- * Node.js provides mechanisms to **catch errors**, **handle** them and take **appropriate actions**.
- * It improves the **reliability** and **user experience** of our applications.



12.17.2 Types of Errors in Node.js

Type	Description	Example
Syntax Errors	Errors in the code syntax.	Missing brackets, commas, etc.
Runtime Errors	Errors that occur during program execution.	ReferenceError, TypeError, RangeError, etc.
System Errors	Errors returned by the OS (file not found, permissions, etc.)	ENOENT, EACCES, etc.
Logical Errors	Errors in the logic of the program (not detected by Node.js).	Wrong calculation, invalid conditions, etc.
Asynchronous Errors	Errors in async operations like callbacks, promises, events.	File read failure, DB connection failure, etc.

12.17.3 Example: Try...Catch

```
try {
  const data = JSON.parse('{ "name": "John" }');
  console.log(data.name);
  console.log(data.age); // This will cause an error
} catch (err) {
  console.log('Error Caught: ' + err.message);
} finally {
  console.log('This block always executes');
```

Output:

```
John
Error Caught: Cannot read properties of undefined (reading 'age')
This block always executes
```

12.17.4 Example: Callback Error Handling

```
const fs = require('fs');
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.log('Error reading file:', err.message);
    return;
  }
  console.log('File content:', data);
});
```

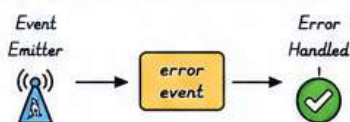
Always check the first argument (err). If it is not null, an error occurred.

12.17.5 Example: Promise Error Handling

```
const fs = require('fs').promises;
fs.readFile('example.txt', 'utf8')
  .then(data => {
    console.log('File content:', data);
  })
  .catch(err => {
    console.log('Error:', err.message);
  });
```

Use `.catch()` to handle rejected promises.

12.17.6 Handling Errors in EventEmitter



- * If an 'error' event is emitted and no listener is attached, Node.js will crash the application.
- * Always handle 'error' events.

```
const EventEmitter = require('events');
const emitter = new EventEmitter();
emitter.on('error', (err) => {
  console.log('Error Event:', err.message);
});
emitter.emit('error', new Error('Something went wrong!'));
```

Output:

```
Error Event:
Something went wrong!
```

12.17.7 Common Built-in Error Objects

Error Name	Description
SyntaxError	Thrown when there is a syntax mistake in the code.
ReferenceError	Thrown when a variable that is not defined is used.
TypeError	Thrown when a value is not of the expected type.
RangeError	Thrown when a value is not in the valid range.
Error	Base class for all errors.

12.17.8 Example: Custom Error

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = 'ValidationError';
  }
}
throw new ValidationError('Invalid input provided');
```

We can create custom errors by extending the Error class.

12.17.9 Key Points

- ✓ Always anticipate errors and handle them properly.
- ✓ Use `try...catch` for synchronous code.
- ✓ Use callbacks / `.catch()` / `async-await` with `try...catch` for asynchronous code.
- ✓ Handle 'error' events in EventEmitter.
- ✓ Provide meaningful error messages to users.
- ✓ Good error handling makes applications robust and reliable.

Key Takeaway:



Error handling is essential in Node.js to gracefully manage failures, prevent crashes and build **reliable**, **maintainable** applications.

