Internals of Node.js | V8 Engine

What is Node.js?

Node.js is an open-source, cross-platform JavaScript runtime environment that allows you to run JavaScript outside the browser. It is built on Chrome's V8 engine and is designed for building scalable network applications.

What is a runtime environment?

A runtime environment is a software layer that provides the necessary resources (such as memory management, I/O operations, and execution context) for a program to run. In the case of Node.js, it provides an environment where JavaScript code can execute outside the browser, using system-level features like file handling, networking, and databases.

Components of a Browser JavaScript Runtime

1. **JavaScript Engine** – Executes JavaScript code (e.g., V8 for Chrome, SpiderMonkey for Firefox, JavaScriptCore for Safari).
2. **Web APIs** – Provides features like `DOM Manipulation`, `Fetch API`, `setTimeout`, `console.log`, and more.
3. **Event Loop** – Handles asynchronous operations and ensures smooth execution of non-blocking tasks.
4. **Callback Queue** – Stores asynchronous tasks (like timers and event listeners) to be processed by the event loop.
5. **Microtask Queue** – Handles promises and other high-priority asynchronous operations before moving to the callback queue.
6. **Rendering Engine** – Updates the UI based on changes in the DOM and CSS.
7. **Heap & Stack** – Manages memory allocation and function execution.

Who Created Node.js?

Node.js was created by **Ryan Dahl** in **2009**. He developed it to improve the way JavaScript handled asynchronous operations, particularly for building scalable, non-blocking server applications.

Capabilities Node.js Added to JavaScript

Before Node.js, JavaScript was mainly used in browsers. Node.js extended JavaScript’s capabilities by allowing it to run outside the browser with features like:

1. **File System Access** – Read, write, and manipulate files (`fs` module).
2. **Networking** – Build web servers, handle HTTP, TCP, UDP, and WebSocket connections (`http`, `net`, and `dgram` modules).
3. **Process Management** – Execute system commands, spawn child processes (`child_process` module).
4. **Asynchronous & Non-blocking I/O** – Handles I/O operations efficiently using event-driven architecture.
5. **Package Management (npm)** – Access a huge ecosystem of libraries and tools via the Node Package Manager (npm).
6. **Cross-platform Compatibility** – Runs on Windows, macOS, and Linux.

Common Myths & Misconceptions About Node.js

1. **"Node.js is a framework"** – ❌ Wrong!

   ✅ Node.js is a **runtime environment**, not a framework. Frameworks like Express.js or NestJS are built on top of Node.js.

2. **"Node.js is only for backend development"** – ❌ Wrong!

   ✅ While Node.js is widely used for backend development, it can also be used for **CLI tools, desktop apps (Electron), and even IoT applications**.

3. **"Node.js is single-threaded, so it's not scalable"** – ❌ Wrong!

   ✅ Node.js uses **event-driven, non-blocking architecture**, allowing it to handle thousands of concurrent connections efficiently. It also supports worker threads for multi-threading when needed.

4. **"Node.js is only good for small projects"** – ❌ Wrong!

✅ Many large companies (Netflix, PayPal, LinkedIn) use Node.js for high-scale applications.

5. **"Node.js is just JavaScript running on the server"** – ❌ Partially True!

   ✅ Node.js extends JavaScript with **built-in modules (fs, http, os, etc.), process handling, and system-level operations** that browsers do not provide.

6. **"Node.js replaces databases"** – ❌ Wrong!

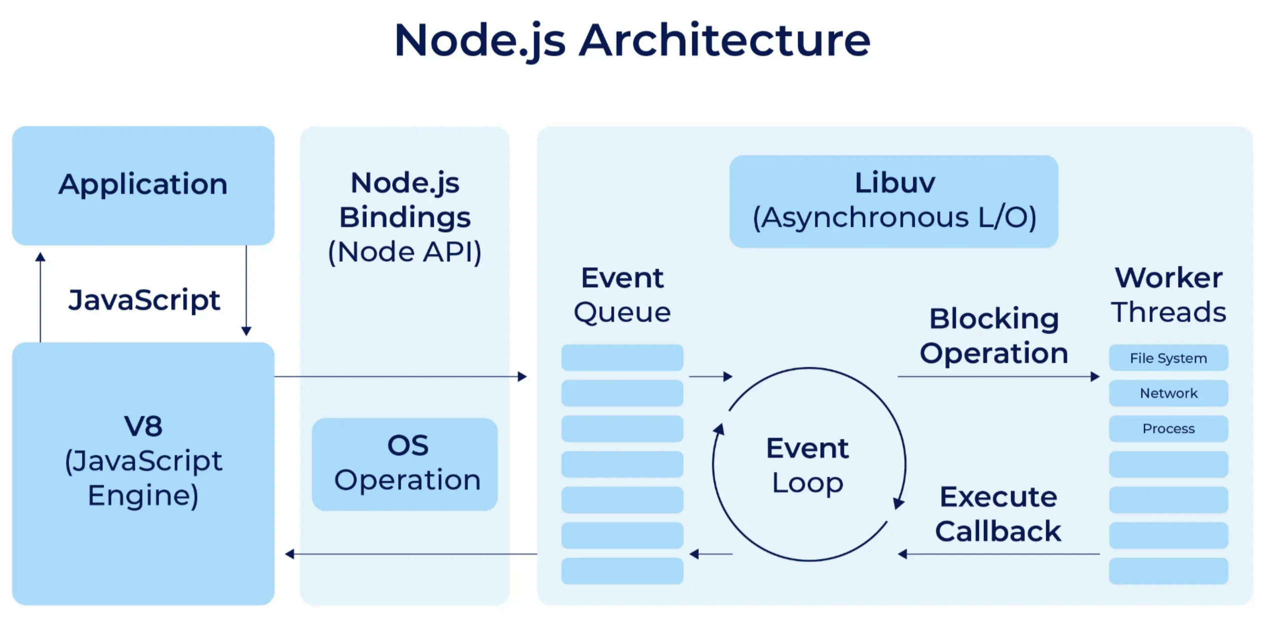
   ✅ Node.js is not a database; it is used to interact with databases like **MongoDB, MySQL, PostgreSQL, and Redis**.

7. **"Node.js does not support multi-threading at all"** – ❌ Wrong!

   ✅ While Node.js is single-threaded by default, it provides **Worker Threads and the Cluster module** to utilize multi-core processors.

Components of Node.js

1. **V8 Engine** – Executes JavaScript code by converting it into machine code. It is the same engine used in Google Chrome.
2. **libuv** – Handles asynchronous operations like file system access, networking, and timers using an event-driven architecture.
3. **Event Loop** – The core mechanism that makes Node.js non-blocking. It continuously checks and executes callbacks from the callback queue.
4. **Node.js APIs** – Built-in modules like `fs` (File System), `http`, `crypto`, `path`, and `os` that provide system-level functionality.
5. **Event Emitter** – A pattern used to handle and listen for events (`events` module).
6. **Streams** – Handles large amounts of data efficiently using readable, writable, duplex, and transform streams.
7. **Buffers** – Helps in handling binary data, especially useful for reading/writing files and working with TCP streams.
8. **Process & Child Processes** – Manages the Node.js process (`process` object) and allows creating subprocesses for multi-threading (`child_process` module).
9. **NPM (Node Package Manager)** – The default package manager for installing and managing dependencies in Node.js projects.
10. **C++ Bindings** – Some Node.js core modules use C++ to interact with system resources for better performance.



V8 Engine – The Heart of Node.js

What is V8?

V8 is an open-source **JavaScript engine** developed by Google, written in **C++**, and used in Google Chrome and Node.js. It is responsible for **compiling and executing JavaScript code** efficiently by converting it into **machine code** instead of interpreting it.

Key Features of V8

1. **Just-In-Time (JIT) Compilation** – Uses **two compilers (Ignition & TurboFan)** to convert JavaScript into optimized machine code at runtime.
2. **Garbage Collection (GC)** – Automatically frees up memory using a **generational garbage collector (Orinoco)** to optimize performance.
3. **Hidden Classes & Inline Caching** – Improves property access speed by dynamically creating hidden classes for objects.
4. **Memory Management** – Uses heap memory and stack memory efficiently to handle variable storage.
5. **WebAssembly Support** – Can execute WebAssembly code for high-performance applications.

How V8 Works in Node.js?

- **JavaScript Code → (V8) → Machine Code**
- Node.js **uses V8 to run JavaScript outside the browser** and extends it with system-level features like **file system access, networking, and process management**.
- Unlike browsers, **Node.js does not include a DOM or Web APIs**, since it is designed for backend development.

Components of the V8 Engine

The **V8 Engine** consists of multiple components that work together to **parse, interpret, compile, and optimize JavaScript code**. Here's a breakdown of its key components:

---

1. Parser (Syntax Analyzer)

- Converts JavaScript code into an **Abstract Syntax Tree (AST)**.
- Ensures the syntax is correct and identifies keywords, variables, and expressions.
- Uses **recursive descent parsing** to process JavaScript code.

Example:

```
let x = 10 + 20;
```

The parser breaks this into:

- **Tokens** (`let`, `x`, `=`, `10`, `+`, `20`)
- **AST** representing the structure of the expression.

2. Ignition (Interpreter)

- **Interprets the AST and generates bytecode** (an intermediate form of code).
- Executes JavaScript in a quick but less optimized way.
- Helps in faster startup time, as it avoids immediate compilation.
- More Resources:
  - Ignition: An Interpreter for V8 [BlinkOn]
  - BlinkOn 6 Day 1 Talk 2: Ignition - an interpreter for V8

3. TurboFan (Optimizing Compiler)

- Converts **bytecode into highly optimized machine code**.
- Performs optimizations like **inline caching, dead code elimination, and hidden class optimizations**.
- Runs in the background while Ignition executes bytecode.

4. Garbage Collector (Memory Manager)

- V8 uses **Orinoco (a generational garbage collector)** to free unused memory.
- Divides memory into **New Space (young objects)** and **Old Space (long-lived objects)**.
- Uses **incremental and concurrent garbage collection** to avoid performance bottlenecks.

- More Resources:
    - Trash talk: the Orinoco garbage collector

5. Profiler & Deoptimizer

- Monitors code execution and identifies frequently used functions for optimization.
- If assumptions made by **TurboFan** turn out to be incorrect, it **de-optimizes** the code and falls back to Ignition.

Execution Flow in V8

1. **Parsing** → Converts JavaScript into AST.
2. **Ignition** → Interprets AST & generates bytecode.
3. **TurboFan** → Compiles bytecode into optimized machine code.
4. **Garbage Collector** → Cleans up memory.