Introduction to Node.js

~ An Overview

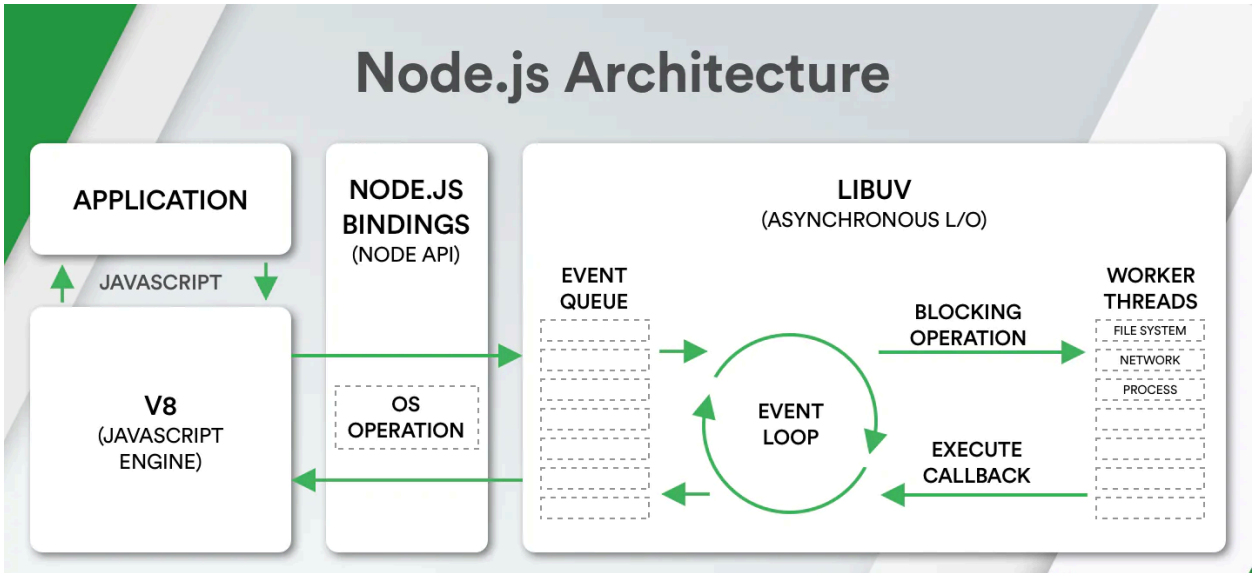
[Source: How much JavaScript do you need to know to use Node.js?]
Github reference: https://github.com/MendDevs/Node.js

What is Node.js?

- It is Open-source, cross-platform JavaScript runtime.
- It is built on Google Chrome's V8 engine, allowing JS to run outside the browser.
- Ideal for building fast and scalable applications.

Architecture & Performance:

- Single-threaded with non-blocking asynchronous I/O (Input/ Output).
- Handles thousands of concurrent connections without creating new threads.
- Avoids CPU waste by resuming operations once I/O responses return.
- Reduces thread management complexity and potential bugs.



Developer Advantages:

- Allows frontend developers to write backend code using JavaScript.
- Supports modern ECMAScript features-you control the version via Node.
- Experimental features can be enabled via runtime flags.

Uses:

- Both frontend and backend
- It excels in performance, simplicity, and modern JavaScript support.

JavaScript prerequisite for Node.js

What is recommended to learn before diving deep with Node.js?	**For Asynchronous Programming:**
Lexical Structure	Asynchronous Programming and Callbacks
Expressions	Timers

Data Types	Promises
Classes	Async and Await
Variables	Closures
Functions	The Event Loop
this operator	
Arrow Functions	
Loops	
Scopes	
Arrays	
Template Literals	
Strict Mode	
ECMAScript 2015 (ES6) and beyond	
Asynchronous JavaScript	

Difference between Node.js and the Browser

1. Same Language, Different Environments

- Both use JavaScript, but development experience is radically different.
- Node.js allows full-stack development using a **single language** (JavaScript for both frontend and backend).
-

2. Ecosystem Differences:

- **Browser:** Focus on DOM manipulation and Web APIs (e.g. document, window, cookies).
- Node.js : Lacks browser-specific APIs but offers powerful modules (eg. filesystem, networking).

3. Environment Control:

- Node.js: You control the runtime version - can use the latest ECMAScript features freely.
- Browser: Dependent on users' browser versions - may need to transpile modern JS and tools like Babel.

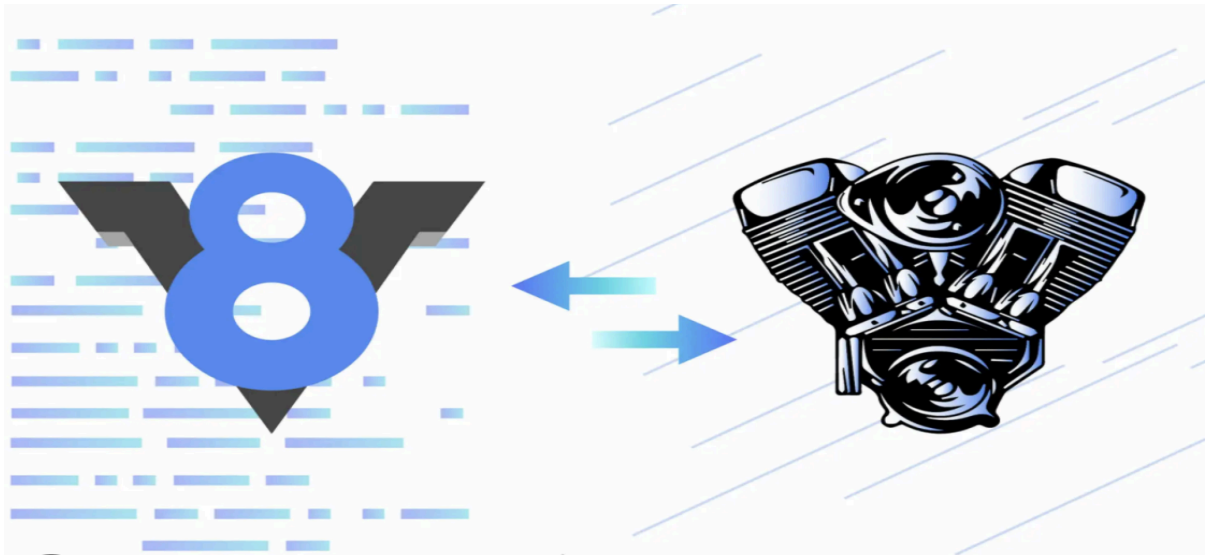
4. Module Systems:

- Node.js : Supports both CommonJS (require ()) and ES Modules (import).
- Browser: Supports only ES Modules (import), and adoption is still evolving.

5. Developer Advantage

- Using JavaScript across both client and server sides simplifies the learning curve and increases productivity for full-stack developers.

The V8 JavaScript Engine:



What is V8?

- JavaScript engine developed by Google, used in Google Chrome and Node.js
- Parses and executes JavaScript code; it does not include the DOM (Document Object Model) or Web APIs - those come from the browser.
- V8 is browser-independent, enabling its use in Node.js and other platforms.

Role in Node.js and Beyond:

- Chosen as the engine for Node.js in 2009.
- Powers not just servers but desktop apps too (eg. via Electron).
- A major reason behind the rise of JavaScript on the server-side

Other JavaScript Engines

- Firefox -> SpiderMonkey
- Safari - JavaScriptCore (aka Nitro)
- Microsoft Edge -> Originally Chakra, now V8 (via Chromium)
- All conform to the ECMAScript (ECMA -262_ standard

Probability & Development :

- V8 is written in C++, runs on MAC, Windows, Linux, etc
- Continuously evolving for performance improvements.
- Part of an ongoing performance race among JS engines, benefiting both developers and users.

Compilation vs Interpretation in JavaScript (with V8) [AI generated]

Traditional Interpretation (Pre-V8 Era)	V8 and Just-In-Time (JIT) Compilation
JavaScript was originally **interpreted**, meaning the code was read and executed **line-by-line**, directly and immediately by the browser. While this made JavaScript simple and flexible, it came at the cost of **performance**—especially for **large or complex applications**.	V8 introduced a game-changing approach: **Just-In-Time (JIT) Compilation**. Instead of purely interpreting JavaScript, V8 **translates JavaScript into optimized machine code** *while the program is running*. This is done in stages:

Each time code ran, it had to be **parsed and executed from scratch**, with no reuse or optimization.	• **Initially** interprets the code for quick startup. • Then **analyzes performance-critical parts** (hot paths). • **Compiles those parts into machine code** for faster execution. • Continuously **optimizes or de-optimizes** code based on actual usage (adaptive optimization).

Why is this important?

In modern web apps, JavaScript codebases often have:

1. Thousands of lines of code
2. Intensive computations

Apps that **run for minutes or hours** (e.g., Google Maps, online editors)

- In such cases, interpretation would be **too slow and inefficient**.
- JIT compilation ensures: Faster execution, Lower CPU usage and Better scalability for real-world applications

Real-World Benefits

- Pages load **faster**.
- Applications run **more smoothly**.
- Users get a **desktop-like experience in the browser**.
- Developers don't need to manually optimize performance at the machine level—V8 handles it intelligently.

Resources:
For more info [https://nodejs.org/en/learn/getting-started/fetch]
Geek4Geek: https://images.app.goo.gl/41wd13VepitQPdV99
Image V8: https://images.app.goo.gl/9fQHYujYDHHLkW4H9