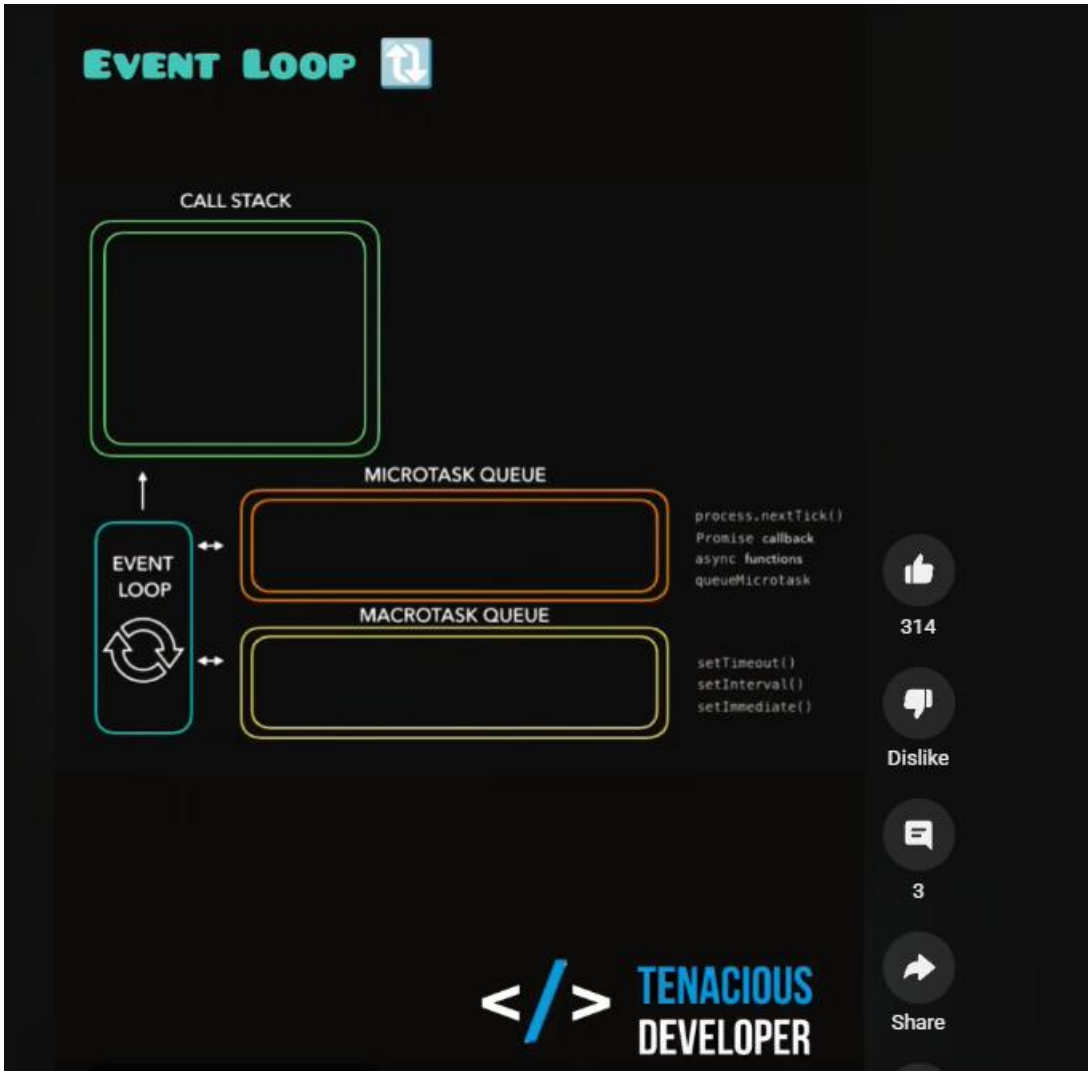

11. What is NodeJS and how it works?

NodeJS is a JavaScript runtime environment that allows developers to run JavaScript code outside of a web browser. It uses the V8 JavaScript engine, which is the same engine used by Google Chrome, to execute JavaScript code on the server-side.

- NodeJS works on an event-driven, non-blocking I/O model, enabling efficient and scalable server-side applications.
- It uses asynchronous programming to handle multiple requests simultaneously, making it ideal for real-time web apps, APIs, and microservices.
- With a vast library ecosystem, NodeJS extends its capabilities for various use cases.

What is a V8 Engine?

The V8 engine is an open-source JavaScript engine developed by Google. It is written in C++ and is primarily designed to execute JavaScript code. V8 is used in multiple applications, especially in Google Chrome, but later utilized to create Node.js for server-side coding.

V8 compiles JavaScript directly to machine code rather than using an interpreter. This compilation process allows for the efficient execution of JavaScript, which is key to the performance benefits in both web browsers (like Chrome) and Node.js applications.

Role of the V8 Engine in Node.js

Node.js is a JavaScript runtime built on Chrome's V8 engine, which means that the engine itself is responsible for executing JavaScript code in a Node.js application. While Node.js provides the environment and APIs for running JavaScript server-side, the actual execution of the code is handled by the V8 engine.

When you write a Node.js application, the code you write is ultimately executed by the V8 engine. It converts JavaScript code into machine code that the system's processor can execute. This helps in fast performance for server-side applications, making Node.js a popular choice for building scalable network applications.

Key Features of Node.js

- **Single-threaded & Event-driven** → means it performs the single task at a time but uses event loop to handle many requests at once.
- **Handle Non-blocking I/O** → Asynchronous operations are handled efficiently with the help of worker pool and threads.
- **Cross-platform** → means we can run them on Windows, Linux, macOS.
- **NPM (Node Package Manager)** → rich ecosystem of libraries and modules.
- **Scalable** → suitable for real-time applications like chat apps, streaming services.
- **Fast Execution** → powered by Google's V8 JavaScript engine.

Answer (Interview Style):

1. Definition of NPX:
NPX stands for **Node Package Execute**. It is a tool that comes with NPM (version 5.2+). It is used to **directly execute Node.js packages** without permanently installing them.

2. Difference between NPM and NPX (Crisp Points):

- **NPM (Node Package Manager):**
  - Used to **install, update, and manage packages.**
  - Example:

    ```bash
    npm install react
    ```

    This downloads React into your project's `node_modules`.
- **NPX (Node Package Execute):**
  - Used to **execute packages directly** without installing them globally.
  - Example:

    ```bash
    npx create-react-app myApp
    ```

    This runs the `create-react-app` package temporarily, creates a project, and then removes it — no global installation needed.

3. Analogy (Easy to remember):
👉 Think of **NPM** like buying a tool and keeping it in your toolbox.
👉 Think of **NPX** like borrowing a tool, using it once, and returning it — no need to store it.

✅ Short closing line:
"In short, **NPM is for managing packages**, while **NPX is for running them directly.**"

NPM (Node Package Manager)

- **Definition:**
  NPM stands for **Node Package Manager.** It is the default package manager for Node.js and helps developers manage libraries, tools, and dependencies.
- **Explanation:**
  Instead of writing everything from scratch, we can install ready-made packages and reuse them with the help of NPM.
- **Key Points:**
  - Comes bundled with Node.js installation.
  - Provides command-line tool (`npm install` , `npm init` , etc.) to manage packages.
- **Example:**
  To use MongoDB in a Node.js project, I can simply run:

```bash
npm install mongoose
```

1. What are Modules in JavaScript?

- A **module** is a file that contains code (functions, objects, variables) which can be reused in other files.
- We just need to **import it** where required.

Benefits:

- **Reusability** → write once, use anywhere.
- **Maintainability** → code is organized in smaller pieces.
- **Avoiding global pollution** → each module has its own scope.

⚡ **Note:**
The code inside a module is wrapped in a hidden function by the module system, so variables/functions don't leak into the global scope.

---

2. CommonJS (CJS)

Working

- **Default module system in Node.js.**
- Uses `require()` to import and `module.exports` / `exports` to export.
- Modules are loaded **synchronously (blocking)** → execution waits until the module is fully loaded.
- Each module runs in its **own scope**, preventing variable conflicts.
- **Not natively supported in browsers.**

👉 **Synchronous = one thing at a time**
Like reading a book → page 1 → page 2 → page 3.

Examples

Default Export

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

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

Named Exports

```js
// math.js
exports.sub = function (a, b) { return a - b; };
exports.mul = function (a, b) { return a * b; };

// app.js
const math = require('./math');
console.log(math.sub(10, 4)); // 6
console.log(math.mul(3, 4));  // 12
```

Multiple Exports

```js
// calculator.js
function add(a, b) { return a + b; }
function subtract(a, b) { return a - b; }
function multiply(a, b) { return a * b; }

const PI = 3.14159;
const version = "2.0.0";

module.exports = { add, subtract, multiply, PI, version };
```

```js
// app.js
const calculator = require('./calculator');
console.log(calculator.add(2, 3)); // 5
console.log(calculator.PI);        // 3.14159

// Destructuring
const { add, PI } = require('./calculator');
console.log(add(10, 20)); // 30
console.log(PI);
```

3. ES6 Modules (ECMAScript Modules / ESM)

Working

- Introduced in ES6 (2015) as the official standard module system.
- Uses `import` and `export`.
- Works in both Node.js (with `"type": "module"` in `package.json` or `.mjs` files) and browsers.
- Imports/exports are static (resolved at compile time) → enables tree shaking (removing unused code).
- Always runs in strict mode.
- Asynchronous (non-blocking) → modules can be loaded in parallel.

Examples

Named Exports

```js
// math.js
export function add(a, b) { return a + b; }
export function sub(a, b) { return a - b; }

// app.js
import { add, sub } from './math.js';
console.log(add(5, 3)); // 8
console.log(sub(10, 4)); // 6
```

Default Export

```js
// math.js
export default function multiply(a, b) {
  return a * b;
}

// app.js
import multiply from './math.js';
console.log(multiply(3, 4)); // 12
```

Import Everything

```js
// app.js
import * as math from './math.js';
console.log(math.add(2, 3)); // 5
console.log(math.sub(7, 4)); // 3
```

4. Is ES6 async?

✅ Yes — ES6 modules are **asynchronous by default**.

- `import` does **not block** execution.
- The JS engine can load multiple modules in parallel.

👉 Example:

```js
import { readFile } from 'fs/promises';
// Loaded asynchronously behind the scenes
```

Copy code

In contrast:

- **CommonJS require() is synchronous** → execution waits until the file is loaded.

---

5. Why prefer ES6 modules over CommonJS?

Advantages of ES6

- **Async loading** → faster for web apps.
- **Cross-platform** → works in both browsers + Node.js.
- **Cleaner syntax** → `import/export` is easier and static.
- **Optimized** → supports tree-shaking.
- **Future standard** → preferred by React, Angular, Vue, Next.js, etc.

---

6. Major Changes in ES6 (2015 → now)

Besides modules, ES6 added many powerful features:

- `let` & `const` → block-scoped variables.
- **Arrow functions** → shorter syntax.
- **Classes** → cleaner OOP.
- **Template literals** → `` `Hello ${name}` ``.
- **Destructuring** → extract values from arrays/objects easily.
- **Default + Rest + Spread operators.**
- **Promises** → better async handling.
- **async/await** → cleaner async code.
- **Modules (import/export)** → standardized modular system.

---

7. Can we use ES6 features in CommonJS?

👉 Yes ✅

- Features like `let/const`, arrow functions, classes, promises, async/await, etc. are **part of JavaScript itself.**
- The only difference is the **module system:**
  - CommonJS → `require/module.exports`.
  - ES6 → `import/export`.