Node.js Core Modules – From Basics to Advanced

We will deeply cover:

1. FS (File System) Module
2. Crypto Module
3. HTTP Module
4. Buffer
5. Events Module

1️⃣ FS (File System) Module

☸️ Definition

The **fs module** allows Node.js to **interact with the file system**. It lets you **create, read, update, delete, and watch files and directories**.

Node.js is often used for backend servers, CLIs, and automation — all of which need file access.

🟩 Why do we need FS module?

Without `fs`, Node.js would:

- Not be able to read configuration files
- Not store logs
- Not upload/download files
- Not serve static files

Almost **every backend project** uses the FS module in some way.

🛑 Features of FS Module

- Works with **files and folders**
- Supports **synchronous & asynchronous** operations
- Supports **streams** (for large files)
- Can **watch files** for changes
- Low-level OS file handling

🔁 Sync vs Async vs Promises

Type	When to use
Synchronous	Scripts, startup tasks

Type	When to use
Asynchronous (callbacks)	Traditional Node apps
Promises (`fs/promises`)	Modern async/await apps

☣️Common FS Methods (Core Knowledge)

1. `fs.readFile()`

Reads entire file into memory.

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

fs.readFile('data.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

✅Use when file size is small.

---

2. `fs.readFileSync()`

Blocking version of `readFile`.

```
const data = fs.readFileSync('data.txt', 'utf8');
```

⚠️Avoid in servers (blocks event loop).

---

3. `fs.writeFile()`

Creates or overwrites a file.

```
fs.writeFile('file.txt', 'Hello Node', err => {
  if (err) throw err;
});
```

---

4. `fs.appendFile()`

Adds content to an existing file.

```
fs.appendFile('logs.txt', 'New log\n', () => {});
```

---

5. `fs.unlink()`

Deletes a file.

```
fs.unlink('file.txt', () => {});
```

---

6. `fs.mkdir()` / `fs.mkdirSync()`

Creates directories.

```
fs.mkdir('uploads', { recursive: true }, () => {});
```

---

7. `fs.readdir()`

Reads directory contents.

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

---

8. `fs.stat()`

Gets file metadata.

```
fs.stat('file.txt', (err, stats) => {
  console.log(stats.isFile());
});
```

---

9. Streams (`fs.createReadStream`, `fs.createWriteStream`)

Used for **large files**.

```
const read = fs.createReadStream('big.mp4');
const write = fs.createWriteStream('copy.mp4');
read.pipe(write);
```

☑️Very important for performance.

⭕Real Projects Using FS

- File upload systems
- Log management
- Static file servers
- Backup systems
- CLI tools

2️⃣Crypto Module

☸️Definition

The **crypto module** provides **cryptographic functionality** such as:

- Hashing
- Encryption
- Decryption
- Secure random values
- Digital signatures

🟩Why Crypto is Important?

Used for:

- Password hashing
- Secure authentication
- Data encryption
- Tokens (JWT, API keys)

📵Hashing vs Encryption (VERY IMPORTANT)

Hashing	Encryption
One-way	Two-way
Cannot be reversed	Can be decrypted
Used for passwords	Used for sensitive data

♓**Passwords should NEVER be encrypted. They must be hashed.**

⚫Password Hashing – REAL WORLD & INTERVIEW READY

❌Wrong Way (Beginner Mistake)

```
crypto.createHash('sha256').update(password).digest('hex');
```

❌Problems: - No salt - Vulnerable to rainbow-table attacks

✅Correct Way – Using Salt + Key Stretching

Node.js uses **PBKDF2** internally (same concept as bcrypt).

📵Register / Signup (Hash Password)

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

function hashPassword(password) {
  const salt = crypto.randomBytes(16).toString('hex');
  const hash = crypto.pbkdf2Sync(password, salt, 100000, 64,
'sha512').toString('hex');

  return { salt, hash };
}
```

♒Store in DB:

```
{
  "salt": "ab12...",
  "hash": "9f8c..."
}
```

🔀Login / Password Verification

```
function verifyPassword(password, salt, storedHash) {
  const hash = crypto.pbkdf2Sync(password, salt, 100000, 64,
'sha512').toString('hex');
  return hash === storedHash;
}
```

🆎Interview Answer (Perfect)

> We never encrypt passwords. We hash them using a salt and key stretching algorithm like PBKDF2 or bcrypt, then compare hashes during login.

📶Encryption & Decryption (Sensitive Data)

Used for: - API secrets - Tokens - Private data

🔐Encryption Example (AES-256)

```
const algorithm = 'aes-256-cbc';
const key = crypto.randomBytes(32); // store securely
const iv = crypto.randomBytes(16);

function encrypt(text) {
  const cipher = crypto.createCipheriv(algorithm, key, iv);
  let encrypted = cipher.update(text, 'utf8', 'hex');
  encrypted += cipher.final('hex');
  return encrypted;
}
```

🔀Decryption Example

```
function decrypt(encrypted) {
  const decipher = crypto.createDecipheriv(algorithm, key, iv);
  let decrypted = decipher.update(encrypted, 'hex', 'utf8');
  decrypted += decipher.final('utf8');
  return decrypted;
}
```

⚫Security Keys – BEST PRACTICES

❌Never do this

```
const key = 'my-secret-key';
```

✅Correct Way (Environment Variables)

```
const key = Buffer.from(process.env.ENCRYPTION_KEY, 'hex');
```

- Stored in `.env`
- Not committed to Git

📵Token Generation

```
crypto.randomBytes(32).toString('hex');
```

Used for: - Password reset links - Email verification - API tokens

---

Crypto Interview Summary

If asked:

Q: How do you store passwords securely?

Hash with salt using PBKDF2 or bcrypt.

Q: Why not encrypt passwords?

Encryption is reversible, hashing is not.

Q: Where do you store encryption keys?

Environment variables or secret managers.

3️⃣HTTP Module

☸️Definition

The **http module** allows Node.js to **create web servers** and handle HTTP requests/responses.

🌐Why HTTP Module?

This is the **foundation of Express.js**.

Creating a Server

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

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World');
});

server.listen(3000);
```

♓Request Object (`req`)

- `req.url`
- `req.method`
- `req.headers`

🌠Response Object (`res`)

- `res.write()`
- `res.end()`
- `res.setHeader()`
- `res.statusCode`

⭕Real Use Cases

- REST APIs
- Microservices
- Backend servers

4️⃣Buffer

☸️Definition

A **Buffer** is a temporary storage for **binary data**.

Node.js runs outside the browser, so it handles **raw binary data** directly.

🟩Why Buffer is Needed?

- Files
- Network streams
- Images, videos

- TCP data

JavaScript alone cannot handle binary efficiently.

🔢 Creating Buffer

```
const buf = Buffer.from('Hello');
```

🔄 Convert Buffer

```
buf.toString();
buf.toJSON();
```

Real Usage

- Streams
- File handling
- Crypto
- HTTP bodies

5️⃣ Events Module

☸️ Definition

The **events module** allows Node.js to handle **event-driven programming**.

Node.js itself is built on events.

Core Concept

- Emit events
- Listen to events

✡️ Example

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

const emitter = new EventEmitter();

emitter.on('login', (user) => {
```

```
  console.log(user, 'logged in');
});

emitter.emit('login', 'Aman');
```

🚫Real-World Usage

- Logging systems
- Notifications
- Chat systems
- Background jobs

🈸Interview-Critical Concepts (Added)

6️⃣Polyfills

☸️Definition

A **polyfill** is code that provides **modern JavaScript features in older environments** that do not natively support them.

In simple words:

> **Polyfill = fallback implementation**

🟩Why Polyfills Exist

JavaScript evolves fast, but:

- Old browsers
- Older Node.js versions
- Embedded systems

may not support new features.

Polyfills allow **backward compatibility**.

How Polyfills Work

If a feature does not exist, we **define it ourselves**.

Example: `Array.prototype.includes`

```
if (!Array.prototype.includes) {
  Array.prototype.includes = function (value) {
    return this.indexOf(value) !== -1;
  };
}
```

🧳Real Usage

- Frontend: Babel + core-js
- Node.js: Runtime compatibility

⚠️Interview Tip

Polyfills:

- Do NOT transpile syntax
- Only patch missing APIs

7️⃣Temporal Dead Zone (TDZ)

🛞Definition

The **Temporal Dead Zone** is the time between:

- Variable creation (hoisting)
- Variable initialization

During this time, accessing the variable causes an error.

Applies To

- `let`
- `const`

❌Not `var`

🧳Example

```
console.log(a); // ❌ ReferenceError
let a = 10;
```

Explanation:

- `a` is hoisted
- But not initialized

⚠️Why TDZ Exists

- Prevents bugs
- Enforces clean code
- Makes `let/const` safer

⚠️Interview Tip

TDZ exists until **execution reaches the declaration**.

⚠️Deep Dive Enhancements

Buffer – Interview-Level Explanation

Why Do We Need Buffer?

JavaScript strings work with **characters**, not raw bytes.

But Node.js deals with:

- Files
- Network packets
- Streams
- Images / videos

All of these are **binary data**.

♓Buffer bridges **JavaScript ↔ Binary world**.

Real-World Buffer Scenario

☢File Upload

- File arrives as chunks (Buffer)
- Node processes chunks
- Saves to disk

```
req.on('data', chunk => {
  console.log(chunk); // Buffer
});
```

Why Convert Buffer?

```
Buffer.from('Hello');        // String → Buffer
buffer.toString('utf8');    // Buffer → String
```

Because:

- Humans read strings
- Machines process bytes

Interview Answer (Short)

> We use Buffer because Node.js handles binary data directly, and JavaScript alone cannot efficiently manage raw bytes.

☣️FS – Advanced Clarity

mkdir vs mkdirSync

mkdir	mkdirSync
Async	Blocking
Non-blocking	Blocks event loop
Preferred in servers	OK for scripts

```
fs.mkdir('logs', () => {});
fs.mkdirSync('logs');
```

---

readFile vs readFileSync

readFile	readFileSync
Async	Sync
Event loop safe	Blocks execution
Best for servers	CLI tools

🔠Streams – Real Scenarios

When to Use Streams

❌`readFile` for large files (memory heavy)

✅Streams for:

- Videos
- Logs
- Large JSON
- File copy

---

createReadStream & createWriteStream

```
const rs = fs.createReadStream('big.txt');
const ws = fs.createWriteStream('copy.txt');
rs.pipe(ws);
```

Real Scenarios

1. **Video streaming server**
2. **Log processing**
3. **File upload/download**
4. **Data pipelines**

Interview Tip

> Streams reduce memory usage by processing data in chunks.

Final Interview Summary

If asked:

"Why Node.js is fast?"

- Event loop
- Non-blocking I/O
- Streams & Buffers

"Why Buffer?"

- Binary data handling

"Why async FS?"

- Avoid blocking event loop

"What is TDZ?"

- Safer variable access

"What is polyfill?"

- Backward compatibility