1. Node.js vs Traditional Server-Side Technologies (PHP, Java)

- **Node.js**:
  - Uses JavaScript on the server side.
  - Event-driven, non-blocking I/O (good for real-time apps).
  - Single-threaded but highly scalable via asynchronous operations.
- **PHP**:
  - Traditional synchronous, multi-threaded model.
  - Widely used for web development (e.g., WordPress).
  - Each request spawns a new process/thread, less efficient under heavy load.
- **Java (Servlets/Spring)**:
  - Multi-threaded, strongly typed, enterprise-grade.
  - Powerful but heavier than Node.js.
- **Key Difference**: Node.js is lightweight, asynchronous, and ideal for I/O-heavy apps; PHP and Java rely more on multi-threading and blocking I/O.

---

2. Role of npm (Node Package Manager)

- Manages external libraries and dependencies for Node.js.
- Provides a registry of reusable packages.
- Helps automate scripts (build, test, deploy).
- **Common Commands**:
  - `npm init` → create `package.json`.
  - `npm install <pkg>` → install dependency.
  - `npm uninstall <pkg>` → remove dependency.
  - `npm update` → update packages.
  - `npm run <script>` → run custom scripts.

---

3. JavaScript Module System

- **CommonJS (CJS)**: Used in Node.js.
  - `require()` and `module.exports`.
  - Loads modules synchronously.
- **ES Modules (ESM)**: Standardized in modern JS.
  - `import` and `export`.
  - Supports asynchronous loading, works in browsers.
- **Significance**: Keeps code modular, organized, and reusable.

---

4. APIs and Fetch API

- **API (Application Programming Interface)**: A way for apps to communicate.
- **JavaScript interaction**: Uses HTTP requests to talk to APIs (GET, POST, PUT, DELETE).
- **Fetch API**:

```
fetch('https://api.example.com/data')
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

  - Promise-based, easier than older `XMLHttpRequest`.

---

5. Functional Programming in JavaScript

- **Principles**:
  - *Immutability*: Avoid changing state; use new objects instead.
  - *Pure Functions*: Same input → same output, no side effects.
  - *First-Class Functions*: Functions can be passed around like values.
  - *Higher-Order Functions*: Functions that take/return other functions.
- Improves readability, testability, and reliability.

6. CommonJS vs ES Modules in Node.js

- **CommonJS**:
  - Syntax: `const x = require('x')`.
  - Executes modules at runtime.
- **ESM**:
  - Syntax: `import x from 'x'`.
  - Static analysis, better optimization.
- **Difference**: CJS loads synchronously; ESM supports async imports and is the future standard.

---

7. File System Module in Node.js

- Provides access to the computer's file system.
- **Synchronous (`fs.readFileSync`)**: Blocks execution until finished.
- **Asynchronous (`fs.readFile`)**: Non-blocking, uses callbacks/promises.
- Importance: Enables reading/writing files, logging, config storage.

---

8. Handling HTTP in Node.js

- Node's `http` module allows server creation.
- **Lifecycle**:
  1. Client sends request.
  2. Node parses request.
  3. Server executes logic.
  4. Response is sent back.
- Example:

```
 const http = require('http');
http.createServer((req, res) => {
  res.write('Hello World');
  res.end();
}).listen(3000);
```

---

9. Express.js vs HTTP Module

- **HTTP Module**: Low-level, requires manual handling of routes, headers, parsing.
- **Express.js**:
  - Middleware-based.
  - Easier routing.
  - Built-in utilities (body parsing, error handling).
  - Faster development.

---

10. RESTful API Principles

- **Principles**:
  - Stateless (each request independent).
  - Client-server separation.
  - Uniform interface (standard HTTP verbs).
  - Resource-based URLs (`/users/1`).
- Importance: Standardizes API design, making apps scalable and maintainable.

---

11. Asynchronous Programming in Node.js

- **Why Important**: Node.js is single-threaded, so async prevents blocking.
- **Callbacks**: Functions passed to handle results/errors.
- **Promises**: Cleaner way to handle async, avoid callback hell.
- **Async/Await**: Syntactic sugar for promises, makes async look synchronous.

---

12. SQL vs NoSQL Databases

- **SQL**: Relational, structured schema, ACID transactions. (e.g., MySQL, PostgreSQL).
- **NoSQL**: Non-relational, schema-less, flexible, scalable. (e.g., MongoDB).
- **Connecting in Node.js**:
  - SQL: `mysql2`, `pg`.
  - NoSQL: `mongoose` (MongoDB).

---

13. Error Handling in Node.js

- Important for stability and debugging.
- **Techniques**:
  - Try/Catch (for sync and async/await).
  - `.catch()` for promises.
  - Error-first callbacks (`function(err, data)`).
  - Global error handlers (`process.on('uncaughtException')`).

---

14. Testing in Node.js

- Ensures code works correctly, prevents regressions.
- **Popular Frameworks**:
  - Mocha → flexible testing.
  - Jest → all-in-one, widely used.
  - Chai → assertion library.
  - Supertest → HTTP endpoint testing.

---

15. Middleware in Express.js

- Middleware = functions that process requests before response.
- **Roles**:
  - Logging, authentication, validation, error handling.
  - Example:

```
app.use((req, res, next) => {
  console.log(req.method, req.url);
  next();
});
```

---

16. Authentication & Security in Web Apps

- **Importance**: Prevent unauthorized access, protect data.
- **Strategies in Node.js**:
  - Session-based auth (cookies).
  - Token-based auth (JWT).
  - OAuth (third-party login, e.g., Google).
  - Password hashing with `bcrypt`.
  - HTTPS and input validation for security.