UNIT - 3 NOTES

UNIT – 3: Node.js and Express.js

I. Introduction to Node.js

1. **What is Node.js?**
    - Node.js is an open-source, cross-platform JavaScript runtime environment.
    - It allows executing JavaScript code *outside* of a web browser, primarily on the server-side.
    - It utilizes Google's high-performance V8 JavaScript engine (the same engine used in Chrome).
2. **What are the Key Features of Node.js?**
    - **Asynchronous & Event-Driven:** Node.js uses a non-blocking, event-driven I/O model. Instead of waiting for I/O operations (like reading files or network requests) to complete, it continues processing other tasks and uses callbacks or promises to handle results when ready. This makes it highly efficient for handling concurrent connections.
    - **Single-Threaded (with Event Loop):** Node.js operates on a single main thread managed by an event loop. While it's single-threaded, its non-blocking nature allows it to handle many connections simultaneously without the overhead of traditional multi-threading.
    - **Non-Blocking I/O:** I/O operations don't block the main thread. When an I/O task starts, Node.js registers a callback and moves on. The callback is executed via the event loop once the I/O operation finishes.
    - **npm (Node Package Manager):** Comes bundled with npm, the world's largest ecosystem of open-source libraries (packages or modules). npm simplifies adding external functionality and managing project dependencies.
    - **Cross-Platform:** Runs consistently on Windows, macOS, and Linux.
    - **Uses JavaScript:** Enables developers to use JavaScript across the entire stack (frontend and backend).
3. **Why Use Node.js for Backend Development?**
    - **Efficiency and Performance:** Excellent for I/O-bound tasks and real-time applications (chat, streaming, APIs) due to its non-blocking nature. Handles many concurrent connections with low resource usage.
    - **Full-Stack JavaScript:** Allows teams to use a single language (JavaScript) for both client-side and server-side development, facilitating code sharing and faster development cycles.
    - **Scalability:** Designed for building scalable applications. It's easy to scale horizontally (adding more server instances) due to its lightweight architecture.
    - **Large Community and Ecosystem:** A vast collection of libraries (via npm) and a vibrant community provide extensive support and pre-built solutions.

4. **How does Node.js compare to Traditional Server-Side Technologies?**
   - **Language:** Uses JavaScript, unlike traditional Java, C#, PHP, etc.
   - **Concurrency Model:** Primarily uses a single-threaded, event-driven, non-blocking I/O model, contrasting with the multi-threaded or blocking models often used by traditional platforms.
   - **Performance:** Often exhibits higher performance for I/O-intensive and real-time scenarios.
   - **Development Paradigm:** Encourages asynchronous programming patterns (callbacks, Promises, async/await).

II. Setting Up Node.js

5. **How to Install Node.js and npm?**
   1. Visit the official Node.js website.
   2. Download the LTS (Long-Term Support) version recommended for most users due to its stability.
   3. Run the installer and follow the on-screen instructions. Ensure `npm` is included in the installation.
6. **How to Verify the Installation?**
   Open your terminal or command prompt and run:
   ```
   node -v
   npm -v
   ```
   These commands should display the installed versions of Node.js and npm, respectively.
7. **How to Set Up a Basic Node.js Project?**
   1. Create a new directory for your project and navigate into it:
      ```
      mkdir my-node-app
      cd my-node-app
      ```
   2. Initialize the project using npm. This creates a `package.json` file to track dependencies and project metadata:
      ```
      npm init -y
      ```
      (The `-y` flag accepts default settings).
8. **How to Install Project Dependencies?**
   Use npm to install packages needed for your project. They will be listed in `package.json`.
   ```
   npm install <package-name>
   ```
   Example: Installing the Express framework:
   ```
   npm install express
   ```
9. **How to Run a Simple Node.js Script?**
   1. Create a JavaScript file (e.g., `app.js`).

2. Add some Node.js code:

   ```
   // app.js
   console.log("Hello, Node.js!");
   ```

3. Execute the script from your terminal:

   ```
   node app.js
   ```

   Output: `Hello, Node.js!`

III. Node.js Core Concepts

10. **What are Node.js Modules?**

   - Modules are reusable blocks of code contained in separate files. They help organize code and promote reusability.
   - **Types of Modules:**
     1. **Core Modules:** Built-in modules provided with Node.js (e.g., `http`, `fs`, `path`, `os`). No installation needed, just `require` them.
     2. **Local/Custom Modules:** Modules you create within your project. Export functionality using `module.exports` or `exports` and import using `require('./path/to/module')`.
     3. **Third-Party Modules:** Modules installed from the npm registry using `npm install`. Imported using `require('package-name')`.

11. **How to Use Core Modules? (Examples)**

   - `http` **Module (Creating a simple 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, () => {
         console.log('Server running at http://localhost:3000');
     });
     ```

   - `fs` **(File System) Module:** (See File System section below)
   - `path` **Module (Working with file paths):**

     ```
     const path = require('path');
     const filePath = '/users/admin/file.txt';

     console.log(path.basename(filePath)); // Output: file.txt
     ```

```
console.log(path.dirname(filePath));  // Output: /users/admin
console.log(path.extname(filePath));  // Output: .txt
```

- `os` **Module (Getting Operating System info):**

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

console.log(os.platform());  // e.g., 'win32', 'linux', 'darwin'
console.log(os.arch());      // e.g., 'x64'
console.log(os.freemem());   // Free system memory in bytes
```

12. **How to Create and Use Custom Modules?**
    1. **Create the module file (e.g., `math.js`):**

```
// math.js
const add = (a, b) => a + b;
const subtract = (a, b) => a - b;

// Export functions to make them available outside
module.exports = { add, subtract };

// Alternatively (exporting one by one):
// exports.add = (a, b) => a + b;
// exports.sub = (a, b) => a - b;
```

    2. **Import and use the module in another file (e.g., `app.js`):**

```
// app.js
const math = require('./math'); // Use relative path

console.log(math.add(5, 3)); // Output: 8
console.log(math.subtract(10, 4)); // Output: 6
```

13. **How to Use Third-Party Modules (npm)?**
    1. Install the package:

```
npm install lodash
```

    2. Require and use it in your code:

```
const _ = require('lodash');

const numbers = [1, 2, 3, 4, 5];
console.log(_.shuffle(numbers)); // Outputs a shuffled version of the array
```

14. **How to Use ES6 Modules (`import`/`export`)?**
    1. Enable ES6 module support in your `package.json`:

```
{
  "name": "my-es6-app",
  "version": "1.0.0",
  "type": "module", // Add this line
  "main": "app.mjs" // Optional: use .mjs extension or keep .js
}
```

2. **Create the module (e.g., `math.mjs`):**

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

3. **Import and use (e.g., `app.mjs`):**

```
// app.mjs
import { add, subtract } from './math.mjs';

console.log(add(5, 3)); // Output: 8
console.log(subtract(10, 4)); // Output: 6
```

15. **How does Node.js handle Asynchronous Operations?**
    Node.js primarily uses these patterns for managing asynchronous code:

    - **Callbacks:** Passing a function as an argument to another function, which is called when the async operation completes. Can lead to "callback hell" if nested deeply.

```
const fs = require('fs');
fs.readFile('file.txt', 'utf8', (err, data) => { // Callback function
    if (err) {
        console.error("Error reading file:", err);
        return;
    }
    console.log("File content:", data);
});
console.log("Reading file..."); // This logs first
```

    - **Promises:** Objects representing the eventual completion (or failure) of an asynchronous operation. Allow chaining (`.then()`) and better error handling (`.catch()`).

```
const fs = require('fs').promises; // Use the promise-based fs module

fs.readFile('file.txt', 'utf8')
    .then(data => {
        console.log("File content:", data);
    })
    .catch(err => {
        console.error("Error reading file:", err);
    });
console.log("Reading file..."); // This logs first
```

- **Async/Await:** Syntactic sugar built on top of Promises, making asynchronous code look and behave more like synchronous code, improving readability. Requires functions to be marked `async`.

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

async function readFileContent() {
    console.log("Reading file..."); // This logs first
    try {
        const data = await fs.readFile('file.txt', 'utf8'); // Pauses here
until promise resolves
        console.log("File content:", data);
    } catch (err) {
        console.error("Error reading file:", err);
    }
}
readFileContent();
```

16. **What is the Event Loop?**
    - The Event Loop is the core mechanism that enables Node.js's non-blocking, asynchronous behavior, even though JavaScript itself is single-threaded.
    - It constantly checks for and processes events from various queues (timers, I/O, etc.).
    - **How it works (simplified):**
        1. Main JavaScript code executes.
        2. When an async operation (like `fs.readFile` or `setTimeout`) is called, Node.js hands it off to the underlying system (OS or thread pool via libuv). It registers a callback function to be executed upon completion.
        3. The main thread continues executing other synchronous code without waiting.
        4. When the async operation completes, its callback is placed into the appropriate event queue (e.g., timer queue, I/O queue).
        5. The Event Loop picks up callbacks from these queues (in a specific order of phases) and executes them on the main thread.
    - **Phases (Simplified Order):** Timers -> Pending Callbacks (I/O) -> Idle/Prepare -> Poll (New I/O events) -> Check (`setImmediate`) -> Close Callbacks. `process.nextTick()` callbacks run between phases.

    - This allows Node.js to handle thousands of concurrent operations efficiently without needing many threads.

17. **What are Events and the `EventEmitter`?**
    - Many core Node.js objects (like HTTP servers, streams) emit events to signal state changes or occurrences.
    - The `events` module provides the `EventEmitter` class, which is the foundation for event-driven architecture in Node.js.
    - **Key Methods:**
        - `on(eventName, listener)`: Registers a listener function for an event.
        - `emit(eventName, [...args])`: Triggers an event, calling all registered listeners with optional arguments.
        - `once(eventName, listener)`: Registers a listener that runs only once for the first occurrence of the event.
        - `off(eventName, listener)` or `removeListener()`: Removes a specific listener.
        - `removeAllListeners([eventName])`: Removes all listeners for a specific event, or all listeners if no eventName is given.
        - `listenerCount(eventName)`: Returns the number of listeners for an event.
    - **Example:**

```javascript
const EventEmitter = require('events');
const myEmitter = new EventEmitter();

// Listener function
const greetHandler = (name) => {
    console.log(`Hello, ${name}!`);
};

// Register listener
myEmitter.on('greet', greetHandler);
myEmitter.once('special', () => console.log('Special event happened once!'));

// Emit events
myEmitter.emit('greet', 'Alice');   // Output: Hello, Alice!
myEmitter.emit('greet', 'Bob');     // Output: Hello, Bob!
myEmitter.emit('special');        // Output: Special event happened once!
myEmitter.emit('special');        // No output

// Remove listener
myEmitter.off('greet', greetHandler);
myEmitter.emit('greet', 'Charlie'); // No output
```

18. **How to Use the File System (`fs`) Module?**

   - Provides functions for interacting with the file system (reading, writing, updating files and directories).
   - Available in both synchronous (blocking) and asynchronous (non-blocking - recommended) forms. Asynchronous forms usually take a callback or return a Promise (`fs.promises`).
   - **Reading a File (Async with Promise):**

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

     async function readFile() {
         try {
             const data = await fs.readFile('example.txt', 'utf8');
             console.log("File content:", data);
         } catch (err) {
             console.error("Error reading file:", err);
         }
     }
     readFile();
     ```

   - **Writing a File (Async with Promise):**

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

     async function writeFile() {
         try {
             await fs.writeFile('output.txt', 'Hello from Node.js!', 'utf8');
             console.log("File written successfully!");
         } catch (err) {
             console.error("Error writing file:", err);
         }
     }
     writeFile();
     ```

19. **What are Node.js Streams?**

   - Streams are objects for efficiently handling reading or writing data sequentially, especially large amounts of data (like files or network traffic), without loading everything into memory at once.
   - They are instances of `EventEmitter`.
   - **Types:**
     - **Readable:** Streams from which data can be read (e.g., `fs.createReadStream`).
     - **Writable:** Streams to which data can be written (e.g., `fs.createWriteStream`).
     - **Duplex:** Streams that are both Readable and Writable (e.g., a TCP socket).

- **Transform:** Duplex streams that modify data as it passes through (e.g., compression with `zlib`).
- Common use case: Piping data from a readable stream to a writable stream (`readableStream.pipe(writableStream)`).

20. **What are Node.js Buffers?**
    - Buffers handle binary data directly. They represent a fixed-size chunk of memory allocated outside the V8 JavaScript heap.
    - Often used when dealing with TCP streams, file system operations, or other scenarios involving raw data.
    - JavaScript historically wasn't designed for binary data, so Buffers fill this gap in Node.js.

21. **How to Interact with the Command Line?**
    - **Running Scripts:** `node your_script.js`
    - **REPL (Read-Eval-Print Loop):** Interactive Node.js shell. Start by typing `node` in the terminal. Execute JavaScript code line by line. Exit with `.exit` or `Ctrl+D`.
    - **Command-Line Arguments (`process.argv`):** An array containing command-line arguments passed to the script. `process.argv[0]` is the Node.js executable path, `process.argv[1]` is the script path, subsequent elements are the actual arguments.

```
// args.js
console.log(process.argv);
const args = process.argv.slice(2); // Get user-provided args
console.log("Arguments:", args);
// Run: node args.js arg1 arg2
```

    - **Reading User Input (`readline` module):** For interactive command-line applications.

```
const readline = require('readline').createInterface({
    input: process.stdin,
    output: process.stdout,
});

readline.question('What is your name? ', (name) => {
    console.log(`Hello, ${name}!`);
    readline.close();
});
```

    - **Executing Shell Commands (`child_process` module):** Run external commands.

```
const { exec } = require('child_process');

exec('ls -l', (error, stdout, stderr) => { // Use 'dir' on Windows
    if (error) {
        console.error(`exec error: ${error}`);
        return;
    }
    console.log(`stdout:\n${stdout}`);
    if (stderr) console.error(`stderr: ${stderr}`);
});
```

- **Building CLI Tools:** Libraries like `yargs` or `commander` simplify parsing complex command-line arguments and building sophisticated CLI applications.

  ```
  npm install yargs
  ```

  ```
  // cli.js
  const yargs = require('yargs/yargs');
  const { hideBin } = require('yargs/helpers');

  const argv = yargs(hideBin(process.argv))
      .command('greet <name>', 'Greet someone', (yargs) => {
          yargs.positional('name', { describe: 'Name to greet', type: 'string'
  });
      }, (argv) => {
          console.log(`Hello, ${argv.name}!`);
      })
      .demandCommand(1, 'You need at least one command')
      .help()
      .argv;
  // Run: node cli.js greet Alice
  ```

22. **How to Use the `console` Module for Debugging?**

    - Provides simple debugging and logging capabilities, similar to the browser console.
    - **Common Methods:**
      - `console.log()`: Standard output.
      - `console.error()`: Output to stderr, typically for errors.
      - `console.warn()`: Output to stderr, typically for warnings.
      - `console.table(data)`: Displays tabular data (arrays or objects) in a formatted table.
      - `console.time(label)` / `console.timeEnd(label)`: Start and stop a timer to measure execution duration.
      - `console.count(label)`: Counts how many times `count()` has been called with a specific label.
      - `console.group(label)` / `console.groupEnd()`: Creates an indented logging group.
      - `console.assert(assertion, message)`: Logs the message and stack trace if `assertion` is false.

- **Example:**

  ```
  console.log("Starting script...");
  console.warn("This is a warning.");
  const users = [{ name: 'Alice', age: 30 }, { name: 'Bob', age: 25 }];
  console.table(users);
  console.time("Loop");
  for (let i = 0; i < 1000; i++) {}
  console.timeEnd("Loop"); // Outputs: Loop: X.XXXms
  console.assert(users.length > 5, "Assertion failed: Not enough users!"); // Will log assertion failure
  ```

23. **How does Single-Threaded Node.js Handle Concurrency?**
    - Node.js achieves concurrency *without* traditional multi-threading for user code via the **Event Loop** and **Non-blocking I/O**.
    - For I/O operations (network, disk), Node.js delegates the work to the system's kernel or utilizes a **Thread Pool** (managed by the `libuv` library).
    - The thread pool handles computationally intensive tasks or blocking I/O operations in the background, preventing them from blocking the main event loop thread.
    - When the background task completes, the result and its associated callback are pushed onto the event queue for the Event Loop to process.
24. **What are Worker Threads?**
    - For CPU-intensive tasks (not I/O), which *would* block the single main thread, Node.js provides `worker_threads`.
    - These allow running JavaScript code in parallel threads, separate from the main event loop.
    - They communicate with the main thread using message passing.
    - Useful for tasks like complex calculations, image processing, etc.
25. **What is Clustering?**
    - The `cluster` module allows creating multiple Node.js processes (forks) that can share the same server port.
    - This enables a Node.js application to take advantage of multi-core systems by running multiple instances of the app, typically one per CPU core.
    - A master process manages the worker processes. Incoming connections are distributed among the workers (e.g., round-robin).
    - Primarily used for scaling network applications (like web servers) across multiple CPU cores.
26. **What is Forking (`child_process.fork`)?**
    - `child_process.fork()` is a special case of spawning child processes specifically for running new Node.js instances.

- It establishes an IPC (Inter-Process Communication) channel between the parent and child process, allowing them to send messages back and forth.
- Used by the `cluster` module internally.

IV. Introduction to Express.js

27. **What is Express.js?**
    - Express.js is a minimal, flexible, and widely-used web application framework for Node.js.
    - It provides a robust set of features for building web and mobile applications, particularly APIs.
    - It simplifies tasks like routing, middleware integration, request/response handling, etc.
28. **Why Use Express.js?**
    - Simplifies building web servers and APIs in Node.js.
    - Provides helpful utilities for routing and HTTP requests/responses.
    - Extensible through a large ecosystem of middleware.
    - Unopinionated, giving developers flexibility in how they structure their applications.
29. **How to Set Up a Basic Express Application?**
    1. Ensure you have a Node.js project initialized (`npm init -y`).
    2. Install Express:

       ```
       npm install express
       ```

    3. Create your main application file (e.g., `server.js`):

       ```
       // server.js
       const express = require('express');
       const app = express(); // Create an Express application instance
       const port = 3000;

       // Define a simple route for the homepage (GET request to '/')
       app.get('/', (req, res) => {
         res.send('Hello World from Express!');
       });

       // Start the server and listen on the specified port
       app.listen(port, () => {
         console.log(`Express server listening at http://localhost:${port}`);
       });
       ```

    4. Run the server:

       ```
       node server.js
       ```

    5. Open `http://localhost:3000` in your browser. You should see "Hello World from Express!".

V. Express.js Core Concepts

30. **What is Routing in Express?**
    - Routing refers to how an application's endpoints (URIs) respond to client requests (HTTP methods like GET, POST, etc.).
    - You define routes using methods on the `app` object (or an `express.Router` instance) that correspond to HTTP methods.
    - Syntax: `app.METHOD(PATH, HANDLER)`
        - `METHOD`: Lowercase HTTP method (e.g., `get`, `post`, `put`, `delete`).
        - `PATH`: The URL path on the server (e.g., `/`, `/users`, `/products/:id`).
        - `HANDLER`: A function (or multiple functions) executed when the route is matched. It receives the request (`req`) and response (`res`) objects, and optionally the `next` function.
31. **How to Define Basic Routes?**

```
// GET request to the homepage
app.get('/', (req, res) => {
  res.send('Homepage');
});

// POST request to /users
app.post('/users', (req, res) => {
  // Logic to create a new user...
  res.send('User created');
});

// GET request to /about
app.get('/about', (req, res) => {
  res.send('About Page');
});
```

32. **How to Handle Route Parameters (`req.params`)?**
    - Route parameters capture dynamic values from the URL path segments. Defined using a colon (`:`).
    - Accessed via the `req.params` object.

```
// Route with a user ID parameter
app.get('/users/:userId', (req, res) => {
  const userId = req.params.userId;
  res.send(`Fetching user profile for ID: ${userId}`);
});
// Request URL: /users/123 -> userId will be "123"
```

```javascript
// Multiple parameters
app.get('/products/:category/:productId', (req, res) => {
  const category = req.params.category;
  const productId = req.params.productId;
  res.send(`Product ID ${productId} in category ${category}`);
});
// Request URL: /products/electronics/456
```

33. **How to Handle Query Parameters (`req.query`)?**
    - Query parameters are key-value pairs appended to the URL after a `?`, used for optional data like sorting or filtering.
    - Accessed via the `req.query` object.

```javascript
// Route to search items
app.get('/search', (req, res) => {
  const keyword = req.query.keyword; // Access 'keyword' query param
  const limit = req.query.limit || 10; // Access 'limit' or default to 10
  res.send(`Searching for "${keyword}" with limit ${limit}`);
});
// Request URL: /search?keyword=express&limit=5
```

34. **How to Use `express.Router` for Modular Routes?**
    - `express.Router` is a mini Express application, useful for grouping related routes into separate files, making the application more organized.

    1. **Create a router file (e.g., `routes/userRoutes.js`):**

    ```javascript
    // routes/userRoutes.js
    const express = require('express');
    const router = express.Router();

    // Middleware specific to this router (optional)
    router.use((req, res, next) => {
      console.log('Time:', Date.now());
      next();
    });

    // Define routes relative to the mount point
    router.get('/', (req, res) => {
      res.send('List of users');
    });

    router.get('/:userId', (req, res) => {
    ```

```
  res.send(`Details for user ${req.params.userId}`);
});

module.exports = router; // Export the router
```

2. **Mount the router in your main app file (`server.js`):**

```
// server.js
const express = require('express');
const app = express();
const userRoutes = require('./routes/userRoutes'); // Import the router

// Mount the user routes under the '/users' path
app.use('/users', userRoutes);

// Other app routes...
app.get('/', (req, res) => res.send('Homepage'));

app.listen(3000, () => console.log('Server running...'));
```

   - Now, requests to `/users` will be handled by `userRoutes.js`, e.g., `GET /users` maps to `/` in `userRoutes.js`, and `GET /users/123` maps to `/:userId`.

35. **How to Use Route Chaining (`app.route()`)?**

   - A way to group different HTTP method handlers for the same route path, avoiding repetition.

```
app.route('/items')
  .get((req, res) => {
    res.send('Get a list of items');
  })
  .post((req, res) => {
    res.send('Add a new item');
  });

app.route('/items/:itemId')
  .get((req, res) => {
    res.send(`Get item ${req.params.itemId}`);
  })
  .put((req, res) => {
    res.send(`Update item ${req.params.itemId}`);
  })
  .delete((req, res) => {
    res.send(`Delete item ${req.params.itemId}`);
  });
```

36. **What is Middleware in Express?**

- Middleware functions are functions that have access to the request object (`req`), the response object (`res`), and the `next` function in the application's request-response cycle.
- They can:
  - Execute any code.
  - Make changes to the request and response objects.
  - End the request-response cycle (by sending a response).
  - Call the next middleware function in the stack using `next()`. If `next()` is not called, the request will hang.
- Used for tasks like logging, authentication, data validation, data parsing, error handling, etc.

37. **How to Use Middleware?**
    - **Application-Level Middleware:** Applied to all requests using `app.use(middlewareFunction)` or `app.METHOD('*', middlewareFunction)`. Must be defined before the routes it should affect.

```javascript
// Simple logger middleware
const logger = (req, res, next) => {
  console.log(`${req.method} ${req.originalUrl} - ${new
Date().toISOString()}`);
  next(); // Pass control to the next middleware/route handler
};
app.use(logger); // Apply logger to all requests
```

    - **Router-Level Middleware:** Applied using `router.use()` or `router.METHOD()`. Works like application-level but is bound to an instance of `express.Router`.
    - **Route-Specific Middleware:** Passed as arguments before the route handler function.

```javascript
const checkAuth = (req, res, next) => {
  // Example: Check if user is authenticated
  if (req.session && req.session.user) {
    next(); // User is authenticated, proceed
  } else {
    res.status(401).send('Unauthorized'); // End cycle with error
  }
};

app.get('/dashboard', checkAuth, (req, res) => { // checkAuth runs before the
route handler
  res.send('Welcome to your dashboard!');
});
```

    - **Error-Handling Middleware:** Special type defined with four arguments (`err`, `req`, `res`, `next`). Must be defined *last*, after all other `app.use()` and routes.

```
app.use((err, req, res, next) => {
  console.error(err.stack); // Log the error
  res.status(500).send('Something broke!');
});
```

38. **What are Common Built-in Express Middleware?**

    - `express.json()`: Parses incoming requests with JSON payloads (Content-Type: application/json). Makes the parsed data available on `req.body`.
    - `express.urlencoded({ extended: true })`: Parses incoming requests with URL-encoded payloads (Content-Type: application/x-www-form-urlencoded), typically from HTML forms. Makes data available on `req.body`. `extended: true` allows for rich objects and arrays.
    - `express.static('public')`: Serves static files (like HTML, CSS, images, client-side JS) from a specified directory (e.g., `public`).

39. **What are Common Third-Party Middleware Examples?**

    - `cors`: Enables Cross-Origin Resource Sharing (allowing requests from different domains, e.g., a frontend on a different port). `npm install cors`.
    - `morgan`: HTTP request logger. Provides predefined logging formats. `npm install morgan`.
    - `cookie-parser`: Parses `Cookie` header and populates `req.cookies` with an object keyed by cookie names. Also handles signed cookies. `npm install cookie-parser`.
    - `multer`: Handles `multipart/form-data` requests, primarily used for file uploads. `npm install multer`.
    - `helmet`: Helps secure Express apps by setting various HTTP headers. `npm install helmet`.

40. **How to Handle Requests (Using the `req` Object)?**
    The `req` object represents the incoming HTTP request and has properties containing information about it:

    - `req.params`: Object containing route parameters (from path segments like `/users/:id`).
    - `req.query`: Object containing query string parameters (from URL like `?key=value`).
    - `req.body`: Object containing the parsed request body (requires middleware like `express.json()` or `express.urlencoded()`).
    - `req.headers`: Object containing request headers.
    - `req.method`: HTTP request method (e.g., 'GET', 'POST').
    - `req.url` / `req.originalUrl`: Request URL path.
    - `req.ip`: Requesting client's IP address.
    - `req.cookies`: Object containing cookies (requires `cookie-parser` middleware).
    - `req.file` / `req.files`: Information about uploaded files (requires `multer` middleware).

41. **How to Send Responses (Using the `res` Object)?**
    The `res` object represents the HTTP response that the Express app sends back. Key methods:

- `res.send([body])`: Sends the HTTP response. Can send strings, Buffer, objects (automatically stringified as JSON), or arrays. Sets `Content-Type` automatically based on content.
- `res.json([body])`: Sends a JSON response. Explicitly sets `Content-Type` to `application/json`.
- `res.status(code)`: Sets the HTTP status code (e.g., `res.status(200)`, `res.status(404)`). Chainable with other response methods (e.g., `res.status(404).send('Not Found')`).
- `res.sendStatus(code)`: Sets the status code and sends the corresponding status message as the body (e.g., `res.sendStatus(404)` sends "Not Found").
- `res.redirect([status,] path)`: Redirects the client to a different URL. Status defaults to 302 (Found).
- `res.render(view [, locals] [, callback])`: Renders a view template (requires a configured template engine) and sends the resulting HTML. `locals` is an object with data for the template.
- `res.sendFile(path [, options] [, callback])`: Sends a file at the given path. Sets `Content-Type` based on file extension. Requires absolute path or specifying `root` option.
- `res.set(field [, value])` or `res.header(field [, value])`: Sets response headers. Can pass an object to set multiple headers.
- `res.cookie(name, value [, options])`: Sets a cookie. Requires `cookie-parser`.
- `res.clearCookie(name [, options])`: Clears a cookie. Requires `cookie-parser`.
- `res.end()`: Ends the response process without sending data. Useful in specific cases like streaming.

42. **How to Use Template Engines (View Engines)?**

- Template engines allow embedding dynamic data into HTML templates before sending them to the client.
- **Steps:**
  1. Install the engine: `npm install ejs` (or `pug`, `hbs`, etc.)
  2. Configure Express to use it (usually in `app.js` or `server.js`):

```
// Tell Express where to find template files (default is 'views' directory)
app.set('views', path.join(__dirname, 'views'));
// Set the view engine
app.set('view engine', 'ejs'); // Use EJS
```

  3. Create template files in the `views` directory (e.g., `views/profile.ejs`).
  4. Use `res.render()` in your route handlers to render the template with data:

```
app.get('/profile/:username', (req, res) => {
  const userData = {
    username: req.params.username,
    email: 'user@example.com',
    posts: ['Post 1', 'Post 2']
  };
  // Renders views/profile.ejs and passes userData to it
  res.render('profile', userData);
});
```

- **Template Syntax Examples:**
  - **EJS (`.ejs`):** Uses `<%= ... %>` for outputting escaped data, `<%- ... %>` for unescaped HTML, `<% ... %>` for control flow (if, loops). Looks very much like HTML with embedded JS.

```
<h1>Welcome, <%= username %>!</h1>
<p>Email: <%= email %></p>
<h2>Posts:</h2>
<ul>
  <% posts.forEach(post => { %>
    <li><%- post %></li> <%# Assuming posts can contain HTML %>
  <% }); %>
</ul>
```

  - **Pug (`.pug`):** Uses indentation and CSS-selector-like syntax. Minimalist.

```
h1 Welcome, #{username}!
p Email: #{email}
h2 Posts:
ul
  each post in posts
    li= post // Escapes HTML by default
```

  - **Handlebars (`.hbs`):** Uses `{{ variable }}` for output. Logic-less by default but supports helpers.

```
<h1>Welcome, {{ username }}!</h1>
<p>Email: {{ email }}</p>
<h2>Posts:</h2>
<ul>
  {{#each posts}}
    <li>{{this}}</li>
  {{/each}}
</ul>
```

43. **How to Handle File Uploads (using `multer`)?**
    1. Install multer: `npm install multer`
    2. Require and configure multer in your route file or main app file:

```
const multer = require('multer');
const path = require('path');

// Configure storage (disk or memory)
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads/'); // Directory to save uploaded files
  },
  filename: function (req, file, cb) {
    // Create a unique filename (e.g., fieldname-timestamp.ext)
    cb(null, file.fieldname + '-' + Date.now() +
path.extname(file.originalname));
  }
});

// Configure file filter (optional: limit file types)
const fileFilter = (req, file, cb) => {
  if (file.mimetype === 'image/jpeg' || file.mimetype === 'image/png') {
    cb(null, true); // Accept file
  } else {
    cb(new Error('Invalid file type, only JPEG and PNG is allowed!'), false);
// Reject file
  }
};

// Initialize multer with options
const upload = multer({
  storage: storage,
  limits: {
    fileSize: 1024 * 1024 * 5 // 5MB file size limit
  },
  fileFilter: fileFilter
});

// Alternatively, for memory storage (useful for cloud uploads):
// const upload = multer({ storage: multer.memoryStorage() });
```

3. Use multer middleware in your route handler:

   - `upload.single(fieldname)`: For a single file upload. Access file via `req.file`.
   - `upload.array(fieldname, maxCount)`: For multiple files under the same fieldname. Access files via `req.files`.
   - `upload.fields([{ name: fieldname1, maxCount: 1 }, { name: fieldname2, maxCount: 5 }])`: For mixed fields. Access files via `req.files`.

```
// Route for single file upload
app.post('/upload-profile-pic', upload.single('profilePic'), (req, res) => {
  if (!req.file) {
    return res.status(400).send('No file uploaded.');
  }
  console.log('Uploaded file:', req.file);
  res.send(`File uploaded successfully: ${req.file.path}`);
});

// Route for multiple file uploads
app.post('/upload-gallery', upload.array('galleryImages', 10), (req, res) =>
{
  if (!req.files || req.files.length === 0) {
    return res.status(400).send('No files uploaded.');
  }
  console.log('Uploaded files:', req.files);
  res.send(`${req.files.length} files uploaded successfully.`);
});
```

4. Ensure the HTML form sending the request has `enctype="multipart/form-data"`.

44. **How to Work with Cookies (using `cookie-parser`)?**

1. Install: `npm install cookie-parser`
2. Use the middleware (provide a secret for signed cookies):

```
const cookieParser = require('cookie-parser');
// Use cookie-parser middleware (provide a secret for signed cookies)
app.use(cookieParser('your secret key here'));
```

3. **Setting Cookies (`res.cookie`):**

```
app.get('/set-cookie', (req, res) => {
  // Simple cookie
  res.cookie('username', 'john_doe');

  // Cookie with options (expires in 1 hour, httpOnly)
  res.cookie('session_id', 'abc123xyz', {
    maxAge: 3600000, // 1 hour in milliseconds
    httpOnly: true, // Cannot be accessed by client-side JS
    // secure: true, // Only send over HTTPS
    // signed: true // Sign the cookie value
  });

  res.send('Cookies have been set!');
});
```

4. **Reading Cookies (`req.cookies`, `req.signedCookies`):**

   ```
   app.get('/read-cookies', (req, res) => {
     const username = req.cookies.username; // Read regular cookie
     const sessionId = req.signedCookies.session_id; // Read signed cookie

     res.send(`Username: ${username}, Session ID: ${sessionId}`);
   });
   ```

5. **Deleting Cookies (`res.clearCookie`):**

   ```
   app.get('/clear-cookie', (req, res) => {
     res.clearCookie('username');
     res.clearCookie('session_id'); // Make sure options match if needed
     res.send('Cookies cleared!');
   });
   ```

45. **How to Use `express-generator` for Scaffolding?**

   - `express-generator` is a command-line tool to quickly create a basic Express application structure (skeleton).

   1. Install it globally: `npm install -g express-generator`
   2. Generate a new project:

      ```
      express my-app --view=ejs # Create 'my-app' using EJS view engine
      # Or: express my-app --view=pug
      # Or: express my-app (defaults to Jade/Pug or none)
      ```

   3. Navigate into the project and install dependencies:

      ```
      cd my-app
      npm install
      ```

   4. Start the server (usually):

      ```
      npm start
      # Or sometimes: DEBUG=my-app:* npm start (for debug logs)
      ```

   - This creates a standard layout with folders for `routes`, `views`, `public` (static assets), a `bin/www` start script, and the main `app.js` file.

46. **What is a Common Project Structure (MVC-like)?**
   While Express is unopinionated, a common pattern inspired by Model-View-Controller (MVC) is often used for organization:

   ```
   /my-project
   |
   ├── /config/          # Configuration files (database, keys, etc.)
   ```

```
├── /controllers/     # Logic for handling requests, interacts with models
│   ├── userController.js
│   └── productController.js
├── /models/          # Data definitions, database interactions (e.g., Mongoose
schemas)
│   ├── User.js
│   └── Product.js
├── /routes/          # Route definitions, map paths to controllers
│   ├── userRoutes.js
│   └── productRoutes.js
├── /views/           # Template files (EJS, Pug, HBS)
├── /public/          # Static assets (CSS, JS, images)
├── /middleware/      # Custom middleware functions
├── /utils/           # Utility functions
├── app.js            # Main Express app setup, middleware registration,
mounting routes
├── server.js         # Server initialization (http.createServer, app.listen) -
Often combined with app.js
├── package.json
└── .env              # Environment variables (use dotenv package)
```

Separating the server creation (`server.js`) from the application logic (`app.js`) can be beneficial for testing, as you can import and test `app.js` without actually starting a listening server.

VI. Node.js Database Integration

47. **How to Connect to Databases?**

    - Node.js connects to databases using specific driver libraries (npm packages) for each database type.
    - Common choices:
        - **MongoDB (NoSQL):** `mongoose` (ODM - recommended), `mongodb` (native driver)
        - **PostgreSQL (SQL):** `pg`
        - **MySQL (SQL):** `mysql2` (faster, supports Promises), `mysql`
        - **Firebase (NoSQL Cloud):** `firebase-admin` (server-side SDK)
        - **Redis (In-Memory Key-Value):** `redis`

48. **Example: Connecting to MongoDB using Mongoose**
    1. Install Mongoose: `npm install mongoose`
    2. Connect and define a Schema/Model:

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

// Connect to MongoDB (replace with your connection string)
```

```
mongoose.connect('mongodb://localhost:27017/myDatabase', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
})
  .then(() => console.log('MongoDB connected successfully.'))
  .catch(err => console.error('MongoDB connection error:', err));

// Define a schema (structure of the document)
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: Number,
  createdAt: { type: Date, default: Date.now }
});

// Create a model from the schema
const User = mongoose.model('User', userSchema); // Model name 'User'
corresponds to 'users' collection

module.exports = User; // Export model for use elsewhere
```

3. **Perform CRUD Operations (in controllers or route handlers):**

```
const User = require('./models/User'); // Import the model

// CREATE (Example in an Express route)
app.post('/users', async (req, res) => {
  try {
    const newUser = new User(req.body); // Assumes req.body matches schema
    const savedUser = await newUser.save();
    res.status(201).json(savedUser);
  } catch (error) {
    res.status(400).json({ message: error.message });
  }
});

// READ (Find all users)
app.get('/users', async (req, res) => {
  try {
    const users = await User.find(); // Find all documents
    res.json(users);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

// READ (Find one user by ID)
app.get('/users/:id', async (req, res) => {
  try {
    const user = await User.findById(req.params.id);
    if (!user) return res.status(404).send('User not found');
    res.json(user);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

// UPDATE (Find by ID and update)
app.put('/users/:id', async (req, res) => {
  try {
    const updatedUser = await User.findByIdAndUpdate(
      req.params.id,
      req.body,
      { new: true, runValidators: true } // Options: return updated doc, run
schema validations
    );
    if (!updatedUser) return res.status(404).send('User not found');
    res.json(updatedUser);
  } catch (error) {
    res.status(400).json({ message: error.message });
  }
});

// DELETE (Find by ID and delete)
app.delete('/users/:id', async (req, res) => {
  try {
    const deletedUser = await User.findByIdAndDelete(req.params.id);
    if (!deletedUser) return res.status(404).send('User not found');
    res.status(204).send(); // No content to send back
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});
```

49. **Example: Connecting to PostgreSQL using `pg`**
    1. Install: `npm install pg`
    2. Connect and Query:

```javascript
const { Pool } = require('pg'); // Or Client for single connection

const pool = new Pool({
  user: 'db_user',
  host: 'localhost',
  database: 'my_database',
  password: 'db_password',
  port: 5432,
});

// Example Query (in an async function or route)
async function getUsers() {
  try {
    const result = await pool.query('SELECT * FROM users');
    console.log(result.rows); // Array of user objects
    return result.rows;
  } catch (err) {
    console.error('Error executing query', err.stack);
    throw err;
  }
}

// Example Insert
async function addUser(name, email) {
  try {
    const result = await pool.query(
      'INSERT INTO users(name, email) VALUES($1, $2) RETURNING *',
      [name, email]
    );
    console.log('Inserted user:', result.rows[0]);
    return result.rows[0];
  } catch (err) {
     console.error('Error executing insert', err.stack);
     throw err;
  }
}
```

VII. Frequently Asked Questions (FAQ) / Advanced Topics

50. **What are First-Class Functions in JavaScript?**
    Functions in JavaScript are "first-class citizens," meaning they can be treated like any other variable:

    - Assigned to variables.

- Passed as arguments to other functions (Callbacks).
- Returned from other functions (Higher-Order Functions like `map`, `filter`, `reduce`).

51. **What is Callback Hell and How to Avoid It?**

    Callback hell (or the "pyramid of doom") refers to deeply nested callback functions, making code hard to read, debug, and maintain.

    ```
    asyncA(resultA => {
      asyncB(resultA, resultB => {
        asyncC(resultB, resultC => {
          // ...and so on...
        });
      });
    });
    ```

    Avoidance Strategies:

    1. **Promises:** Chain `.then()` calls for sequential operations.
    2. **Async/Await:** Write asynchronous code that looks synchronous. (Most recommended)
    3. **Named Functions:** Define callbacks as separate named functions instead of inline anonymous functions.
    4. **Modularization:** Break down complex asynchronous logic into smaller, reusable modules/functions.

52. **Why use Promises over Callbacks?**

    - **Readability:** Avoids deep nesting (callback hell). Chaining `.then()` is often clearer.
    - **Error Handling:** Centralized error handling using `.catch()` for an entire promise chain. Callbacks require error checking in each step (`if (err) ...`).
    - **Composition:** Easier to combine multiple asynchronous operations (e.g., `Promise.all()`, `Promise.race()`).
    - **Return Values:** Promises *return* a value (the promise object itself), making them easier to pass around and manage.

53. **What is the difference between `process.nextTick()` and `setImmediate()`?**

    Both schedule callbacks to run asynchronously, but at different phases of the event loop:

    - **`process.nextTick(callback)`:** Schedules the callback to run *immediately* after the current operation completes, *before* the event loop proceeds to the next phase (like I/O or timers). `nextTick` callbacks have higher priority and run before any I/O events or timers. Use sparingly, as too many can starve the I/O loop.
    - **`setImmediate(callback)`:** Schedules the callback to run in the "check" phase of the event loop, which occurs *after* the "poll" (I/O) phase and before the "close callbacks" phase. It effectively runs after I/O events for the current loop iteration. Generally safer for deferring execution after I/O.

54. **What are Node.js Exit Codes?**
    Exit codes are numbers returned by a process when it terminates, indicating success or the type of error. Common codes:

    - `0`: Success.
    - `1`: Uncaught Fatal Exception (unhandled error).
    - `5`: Fatal Error (V8 internal error).
    - `7`: Internal Exception Handler Run-Time Failure.
    - `9`: Invalid Argument.
    - `12`: Invalid Debug Argument.
    - Others exist for specific internal errors.

55. **What are Stubs in Testing?**

    - Stubs are functions or objects used in testing to replace real implementations of dependencies.
    - They simulate behavior (e.g., returning specific data, throwing errors) without actually executing the original code (like making real DB calls or HTTP requests).
    - Useful for:
        - Isolating the code under test.
        - Making tests faster and more predictable.
        - Controlling different scenarios (success, failure).
    - Libraries like `Sinon.JS` are commonly used to create stubs, spies, and mocks in Node.js testing.

56. **What is the Reactor Pattern?**

    - An architectural pattern for handling concurrent service requests delivered to a server.
    - Node.js's event loop model is an implementation of the Reactor pattern.
    - **Components:**
        - **Reactor:** Waits for I/O events (requests).
        - **Dispatcher/Demultiplexer:** Handles event notification and dispatches them.
        - **Handlers/Request Handlers:** Specific functions that process the actual events (callbacks).
    - It allows handling multiple I/O operations concurrently using a single thread by reacting to events as they occur.

57. **How to measure the performance of async operations?**

    - Node.js has a built-in `perf_hooks` module based on the standard Web Performance APIs.
    - **Key tools:**
        - `performance.now()`: High-resolution timestamp.
        - `performance.mark(name)`: Creates a named timestamp marker.

        - `performance.measure(name, startMark, endMark)`: Measures the duration between two marks.
        - `PerformanceObserver`: Efficiently observes and collects performance entries (marks, measures).

```
const { PerformanceObserver, performance } = require('perf_hooks');

const obs = new PerformanceObserver((items) => {
  const entries = items.getEntries();
  entries.forEach(entry => {
    console.log(`${entry.name}: ${entry.duration}ms`);
  });
  // performance.clearMarks(); // Clear marks after observing if needed
});
obs.observe({ entryTypes: ['measure'], buffered: true });

async function someAsyncTask() {
  performance.mark('task-start');
  // Simulate async work
  await new Promise(resolve => setTimeout(resolve, 100));
  performance.mark('task-end');
  performance.measure('Async Task Duration', 'task-start', 'task-end');
}

someAsyncTask();
```

    - `async_hooks` can be combined with `perf_hooks` for more detailed tracking tied to asynchronous contexts.

58. **What is WASI (WebAssembly System Interface)?**
    - WASI is an API specification designed to allow WebAssembly code running outside the browser (like in Node.js) to interact with the underlying operating system in a standardized and secure way.
    - It provides POSIX-like functions for accessing resources like the file system, networking, clocks, etc.
    - Node.js provides experimental WASI support via the `WASI` class, enabling developers to run compiled WASM modules that need system interaction.
59. **How do you manage packages in your Node.js project?**
    - Packages (libraries or modules) are managed using a package manager, most commonly **npm** (Node Package Manager) or **yarn**.
    - **`package.json`**: This file, located in the root of your project, is crucial. It records:
        - Project metadata (name, version, description, author).

        - Dependencies (`dependencies`): Packages required for the application to run in production.
        - Development Dependencies (`devDependencies`): Packages only needed during development (e.g., testing libraries, build tools).
        - Scripts (`scripts`): Custom commands you can run using `npm run <script-name>` (e.g., `npm start`, `npm test`).
    - **`package-lock.json` (npm) or `yarn.lock` (yarn):** These files automatically generated/updated when you install packages. They lock down the *exact* versions of all installed dependencies (including dependencies of dependencies). This ensures that anyone else setting up the project gets the exact same versions, leading to consistent builds and avoiding the "works on my machine" problem.
    - **Common Commands:**
        - `npm install <package-name>`: Installs a package and adds it to `dependencies`.
        - `npm install <package-name> --save-dev` (or `-D`): Installs a package and adds it to `devDependencies`.
        - `npm install`: Installs all dependencies listed in `package.json` (using versions specified in `package-lock.json`).
        - `npm uninstall <package-name>`: Removes a package.
        - `npm update`: Updates packages to the latest allowed versions according to `package.json` constraints and updates `package-lock.json`.
        - `npm list`: Lists installed packages.

60. **How is Node.js better than other frameworks most popularly used?**
    - **Asynchronous Performance:** Node.js excels at handling many concurrent I/O-bound operations (like network requests, database queries) efficiently due to its non-blocking, event-driven architecture. This can lead to higher throughput and lower latency compared to traditional blocking frameworks for certain types of applications (e.g., real-time apps, APIs).
    - **JavaScript Ecosystem:** Leverages the vast npm ecosystem, providing readily available libraries for almost any task.
    - **Full-Stack JavaScript:** Enables using JavaScript on both the frontend and backend, potentially streamlining development, sharing code, and reducing context switching for developers.
    - **Fast Development Cycle:** The single language and large library support can often lead to faster development and iteration.
    - **V8 Engine Performance:** Runs on Google's highly optimized V8 engine, which is constantly improving.
    - **Simplicity (for certain tasks):** The event-driven model can simplify handling concurrency compared to manual thread management in some other frameworks, although asynchronous programming requires a different mindset.

61. **What are some commonly used timing features of Node.js?**

   - **`setTimeout(callback, delay, [...args])`** / **`clearTimeout(timeoutId)`**: Executes the `callback` function once after the specified `delay` (in milliseconds). Returns a `timeoutId` which can be used with `clearTimeout` to cancel the execution.
   - **`setInterval(callback, delay, [...args])`** / **`clearInterval(intervalId)`**: Executes the `callback` function repeatedly every `delay` milliseconds. Returns an `intervalId` used with `clearInterval` to stop the repetition.
   - **`setImmediate(callback, [...args])`** / **`clearImmediate(immediateId)`**: Schedules the `callback` to execute in the "check" phase of the *next* event loop iteration (essentially, after I/O callbacks for the current iteration).
   - **`process.nextTick(callback, [...args])`**: Schedules the `callback` to execute at the very end of the current operation, *before* the event loop continues to the next phase. Has higher priority than `setImmediate`.

62. **Why is Node.js single-threaded?**
   - Node.js was designed with the explicit goal of exploring asynchronous I/O processing on a single thread.
   - The creators believed this model could be more efficient and scalable for I/O-bound applications compared to traditional thread-per-request models, which can incur significant overhead from thread creation, context switching, and memory usage.
   - By keeping the main execution path single-threaded and offloading I/O to the system/thread pool, Node.js avoids the complexities of managing threads directly in user code for concurrent I/O.

63. **How do you create a simple server in Node.js that returns Hello World?**
   Using the built-in `http` module:

```
const http = require('http'); // Import the http module

// Create the server
const server = http.createServer((request, response) => {
  // Set the response HTTP header with HTTP status and Content type
  response.writeHead(200, {'Content-Type': 'text/plain'});

  // Send the response body "Hello World"
  response.end('Hello World\n');
});

// Server listens on port 3000
server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

64. **How many types of API functions are there in Node.js?**
    Node.js APIs can broadly be categorized by their execution behavior:

    - **Asynchronous, Non-blocking Functions:** These functions typically handle I/O operations (file system, network, database). They initiate the operation and return immediately, allowing the event loop to continue. They use callbacks, Promises, or async/await to handle the result later.
    - **Synchronous, Blocking Functions:** These functions execute and block the event loop until they complete. They are often used for operations that directly affect the current process state or for simplicity in scripts. Many core modules offer synchronous alternatives (e.g., `fs.readFileSync`), but they should be used cautiously in server applications as they can block all other concurrent requests.

65. **What is REPL?**

    - REPL stands for **Read**, **Eval**uate, **Print**, **Loop**.
    - It's an interactive programming environment or shell that takes single user inputs (Read), executes them (Evaluate), returns the result to the user (Print), and then waits for the next input (Loop).
    - The Node.js REPL is accessed by simply typing `node` in your terminal without any filename. It allows you to quickly test JavaScript code and explore Node.js features.

66. **What is the purpose of module.exports?**

    - `module.exports` is a special object in the Node.js module system.
    - Its primary purpose is to define what code (functions, objects, variables) from a specific module (file) will be exposed and made available for use when that module is imported (`require`d) by another module.
    - By default, `module.exports` is an empty object. You assign or add properties to it to export functionality.
    - Example:

      ```
      // utils.js
      const PI = 3.14159;
      function calculateCircumference(radius) {
        return 2 * PI * radius;
      }
      // Export the function and constant
      module.exports = {
        calculateCircumference: calculateCircumference,
        PI: PI
      };

      // app.js
      const utils = require('./utils');
      ```

```
console.log(utils.PI); // 3.14159
console.log(utils.calculateCircumference(10)); // ~62.83
```

- `exports` is a shorthand alias for `module.exports`. You can attach properties directly (`exports.myFunction = ...`), but reassigning `exports` itself (`exports = ...`) will break the connection, while reassigning `module.exports` (`module.exports = ...`) works as expected.

67. **What tools can be used to assure consistent code style?**
    - **Linters:** Analyze code for potential errors and stylistic issues based on configurable rules.
      - **ESLint:** The most popular linter for JavaScript. Highly configurable with many plugins and predefined style guides (like Airbnb, Standard, Google).
    - **Formatters:** Automatically reformat code to conform to a specific style.
      - **Prettier:** An opinionated code formatter that enforces a consistent style by parsing your code and re-printing it. Often used alongside ESLint.
    - **Integration:** These tools can be integrated into code editors (like VS Code) to provide real-time feedback and automatic formatting on save. They can also be added to Git hooks (e.g., using `husky` and `lint-staged`) to ensure code meets standards before being committed.

68. **If Node.js is single threaded then how does it handle concurrency?**
    This is a crucial point often misunderstood. Node.js *itself* (your JavaScript code execution) runs on a single main thread managed by the Event Loop. However, it achieves concurrency for I/O operations through:
    1. **Non-Blocking I/O:** Node.js delegates I/O operations (like reading files, network requests, database queries) to the operating system's kernel, which can handle many such operations concurrently. Node.js doesn't wait; it registers a callback and moves on.
    2. **Libuv Library & Thread Pool:** For operations that cannot be handled directly by the OS's non-blocking mechanisms or are CPU-intensive, Node.js uses the `libuv` library. `libuv` manages a **thread pool** (a small number of background threads). Blocking or intensive tasks are offloaded to these threads.
    3. **Event Loop:** When an OS operation or a thread pool task completes, `libuv` places the corresponding callback into the event queue. The single-threaded Event Loop picks up these callbacks and executes them *sequentially* on the main thread.

    So, while your *JavaScript code* runs on one thread, the *underlying I/O and some computations* happen concurrently in the background (OS or thread pool), allowing the main thread to remain responsive and handle many connections efficiently.

69. **How does Node.js overcome the problem of blocking of I/O operations?**
    Node.js overcomes I/O blocking primarily through its **asynchronous, non-blocking I/O model** facilitated by the **Event Loop** and **libuv**.
    - Instead of waiting for an I/O operation (like reading from disk) to finish, Node.js initiates the operation and immediately returns control to the event loop.
    - It provides a callback function (or uses Promises/async-await) that will be executed *only when* the I/O operation is complete.

- While the I/O operation happens in the background (handled by the OS or libuv's thread pool), the single main Node.js thread is free to handle other incoming requests or execute other code.
- This prevents a slow I/O operation from halting the entire application, allowing it to remain responsive and handle high concurrency.

70. **How can we use async await in node.js?**
`async/await` is modern syntax built on top of Promises to make asynchronous code easier to write and read, making it look more like synchronous code.
    1. **`async` Keyword:** Place `async` before a function definition to indicate that it will perform asynchronous operations and will implicitly return a Promise.
    2. **`await` Keyword:** Use `await` *inside* an `async` function before calling a function that returns a Promise. It pauses the execution of the `async` function *only* until that Promise settles (either resolves with a value or rejects with an error).
    3. **Error Handling:** Use standard `try...catch` blocks within the `async` function to handle rejected Promises (errors).

```javascript
const fs = require('fs').promises; // Using the promise-based fs module

// Define an async function
async function readAndLogFile(filePath) {
  console.log(`Attempting to read: ${filePath}`);
  try {
    // Pause execution until fs.readFile resolves
    const data = await fs.readFile(filePath, 'utf8');
    // Execution resumes here after the file is read
    console.log('File content:', data);
    return data; // The promise returned by readAndLogFile resolves with data
  } catch (error) {
    // Catch errors if the promise rejects (e.g., file not found)
    console.error('Error reading file:', error);
    // The promise returned by readAndLogFile rejects with the error
    throw error; // Re-throw if needed, or handle appropriately
  }
}

// Call the async function
async function main() {
  await readAndLogFile('myFile.txt');
  console.log('Finished reading file.');
}

main();
```

71. **What is an Event Emitter in Node.js?**

- An `EventEmitter` is a class provided by Node.js's built-in `events` module.
- It's the foundation for objects that need to emit named events to signal that something has occurred (e.g., an HTTP server receiving a request, a stream receiving data, a timer finishing).
- Other parts of the application can "listen" for these events using methods like `.on()` or `.once()` and provide callback functions (listeners) to be executed when the event is emitted.
- This facilitates an event-driven architecture, allowing different parts of an application to react to occurrences without being tightly coupled.
- **Example:**

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

class MyEmitter extends EventEmitter {} // Inherit from EventEmitter
const myEmitter = new MyEmitter();

// Register a listener for the 'data' event
myEmitter.on('data', (chunk) => {
  console.log(`Received data chunk: ${chunk}`);
});

// Simulate emitting the 'data' event
console.log("Emitting event...");
myEmitter.emit('data', 'Sample Data 1');
myEmitter.emit('data', 'Sample Data 2');
console.log("Finished emitting.");
// Output:
// Emitting event...
// Received data chunk: Sample Data 1
// Received data chunk: Sample Data 2
// Finished emitting.
```

72. **How can Node.js performance be enhanced through clustering?**

- By default, a Node.js application runs as a single process on a single CPU core, regardless of how many cores the server has.
- The `cluster` module allows you to create multiple child processes (workers) of your Node.js application, typically one worker per available CPU core.
- These worker processes all run the same application code but operate independently.
- A special **master** process is responsible for creating and managing the workers.
- Crucially, all worker processes can **share the same server port** (e.g., port 80 or 3000). The master process listens on the port and distributes incoming connections among the available worker processes (often using a round-robin strategy).
- **Benefits:**

        - **CPU Utilization:** Fully utilizes multi-core CPUs, significantly increasing the application's capacity to handle concurrent requests.
        - **Increased Throughput:** More requests can be processed in parallel across the workers.
        - **Improved Resilience:** If one worker process crashes, the master process can detect it and spawn a new one, while other workers continue handling requests (though application state might be lost in the crashed worker).
    - **Implementation:** Involves checking if the current process is the master or a worker (`cluster.isMaster` or `cluster.isPrimary` / `cluster.isWorker`) and either forking workers (master) or starting the application server (worker).
73. **What is a thread pool and which library handles it in Node.js?**
    - A **thread pool** is a collection of pre-initialized background threads managed by a library or system.
    - In Node.js, the thread pool is primarily used to handle **synchronous, blocking, or CPU-intensive operations** that cannot be efficiently handled by the operating system's asynchronous, non-blocking APIs.
    - Examples include certain file system operations (`fs` synchronous methods, sometimes async ones on certain platforms), DNS lookups (`dns.lookup`), some cryptography (`crypto`) functions, and compression (`zlib`).
    - When Node.js needs to perform one of these tasks, instead of blocking the main event loop thread, it delegates the task to one of the threads in the pool.
    - The library responsible for managing the thread pool (and much of Node.js's async I/O abstraction) is **libuv**. `libuv` is a multi-platform C library focused on asynchronous I/O.
74. **How are worker threads different from clusters?**
    Both are used for parallelism in Node.js, but serve different primary purposes:
    - **Cluster Module:**
        - **Unit:** Multiple **Processes**. Each worker is a separate Node.js process with its own memory space and V8 instance.
        - **Primary Use: Scaling network applications** (like HTTP servers) across multiple CPU cores to handle more concurrent **I/O-bound** requests by sharing a port.
        - **Communication:** Uses Inter-Process Communication (IPC) which is generally slower than intra-process communication.
        - **Memory:** No direct memory sharing between workers (each has its own heap).
        - **Overhead:** Higher memory footprint and startup time due to multiple processes.
    - **Worker Threads (`worker_threads` module):**
        - **Unit:** Multiple **Threads** within a *single* Node.js process.
        - **Primary Use:** Performing **CPU-intensive** tasks in parallel without blocking the main event loop thread within that single process. Good for calculations, data processing, etc.

- **Communication:** Uses message passing via `MessageChannel` and can share memory directly using structures like `SharedArrayBuffer` and `Atomics` (requires careful handling).
- **Memory:** Can share memory, leading to lower overall memory consumption compared to clusters for certain tasks.
- **Overhead:** Lower startup time and memory overhead compared to spawning full processes.

Analogy: Think of `cluster` as opening multiple separate copies of your restaurant (processes) to serve more customers (requests) simultaneously. Think of `worker_threads` as hiring multiple specialized chefs (threads) within your single restaurant (process) to handle complex cooking tasks (CPU-bound work) faster, while the main waiter (event loop) keeps taking orders.

75. **How to measure the duration of async operations?**
(This is largely the same as question 57, focusing specifically on duration)
Use the `perf_hooks` module:
    1. **Mark Start:** `performance.mark('operation-start');` just before initiating the async operation.
    2. **Mark End:** `performance.mark('operation-end');` immediately after the async operation completes (e.g., inside the `.then()` block of a Promise, after an `await`, or inside the callback).
    3. **Measure:** `performance.measure('Operation Duration', 'operation-start', 'operation-end');` to create a performance entry representing the duration.
    4. **Observe:** Use a `PerformanceObserver` to listen for 'measure' entries and log or process their `.duration` property.

```
const { PerformanceObserver, performance } = require('perf_hooks');

const obs = new PerformanceObserver((list) => {
  const entry = list.getEntries()[0];
  console.log(`${entry.name}: ${entry.duration.toFixed(2)}ms`);
});
obs.observe({ entryTypes: ['measure'] });

function simulateAsyncWork(callback) {
  performance.mark('work-start');
  setTimeout(() => {
    performance.mark('work-end');
    performance.measure('Simulated Work Duration', 'work-start', 'work-end');
    callback();
  }, 150); // Simulate 150ms work
}

simulateAsyncWork(() => {
  console.log('Async work finished.');
  // Disconnect observer if no longer needed
  // obs.disconnect();
});
```