Payal Mam Theory Assignment 1

Q.1. Nodejs : Introduction, features, execution architecture

Ans.1.

Introduction to Node.js

Node.js is an open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside of a web browser. It allows developers to use JavaScript to write server-side code, thus enabling the creation of dynamic web applications with one language for both client and server-side scripting.

Key Features of Node.js

1. Asynchronous and Event-Driven:

- Asynchronous: Node.js uses non-blocking I/O operations, allowing multiple operations to be handled concurrently. This improves the efficiency and performance of applications, especially under high-load conditions.

- Event-Driven: Node.js is based on an event-driven architecture, where an event loop processes and handles events. This model makes it well-suited for real-time applications such as chat apps and gaming servers.

2. Single-Threaded but Scalable:

- Node.js uses a single-threaded event loop to handle all asynchronous operations. While this might seem like a limitation, it is actually a strength when combined with non-blocking I/O. The single thread handles multiple concurrent connections without the overhead of context switching.

3. High Performance:

- Built on Google's V8 JavaScript engine, Node.js compiles JavaScript directly into native machine code, resulting in faster execution and improved performance.

4. NPM (Node Package Manager):

- Node.js includes NPM, which is the largest ecosystem of open-source libraries in the world. Developers can easily find and use various packages to add functionality to their applications, accelerating development and reducing redundancy.

5. Cross-Platform:

- Node.js can be run on multiple platforms such as Windows, macOS, Linux, and Unix, making it versatile and widely usable.

6. Microservices and APIs:

- Node.js is an excellent choice for building microservices and APIs due to its lightweight nature and ability to handle many simultaneous connections efficiently.

Execution Architecture of Node.js

Node.js utilizes the following key architectural components:

1. Event Loop:

- The core of Node.js's execution model is the event loop. The event loop is a single-threaded loop that handles all asynchronous callbacks in Node.js. It continuously polls for new events and executes their corresponding callback functions.

2. Single Thread:

- Despite being single-threaded, Node.js can manage multiple I/O operations concurrently using the event loop. This is different from traditional multi-threaded server models where each client request spawns a new thread.

3. Non-Blocking I/O:

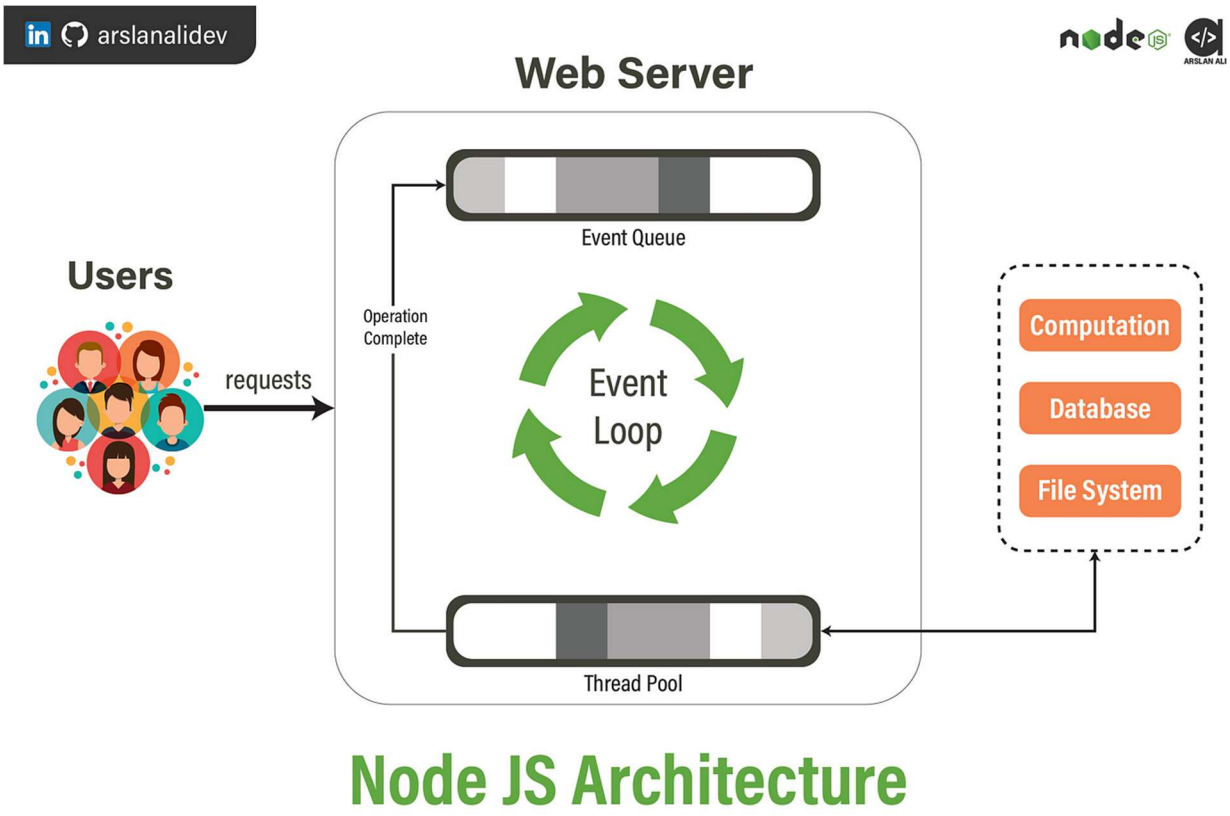
- Node.js operations are non-blocking, meaning that I/O operations (such as reading from disk or network requests) do not block the execution of other code. This is achieved through callback functions, promises, or async/await syntax.

4. Libuv:

- Node.js uses the libuv library to implement the event loop and handle asynchronous operations. Libuv provides a consistent API across different operating systems, abstracting away platform-specific details.

5. V8 JavaScript Engine:

- Node.js uses the V8 engine (developed by Google for Chrome) to execute JavaScript code. V8 compiles JavaScript directly to machine code, making execution fast and efficient.



Q.2. Note on modules with example. Explain ECM, CJS with examples

Ans.2.

Modules in Node.js

Node.js uses modules to encapsulate and reuse code across different parts of an application. Modules are essentially JavaScript files that export variables, functions, classes, or objects to be used in other files. Node.js has a built-in module system that supports two types of modules: CommonJS (CJS) and ECMAScript (ESM).

CommonJS (CJS) Modules

CommonJS is the default module system in Node.js. In CommonJS, modules are loaded synchronously, and they are defined using `module.exports` or `exports`.

Example of CommonJS Module

math.js (CommonJS Module):

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

function subtract(a, b) {
  return a - b;
}

module.exports = {
  add,
  subtract
};
```

app.js (Using the CommonJS Module):

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

const sum = math.add(5, 3);
const difference = math.subtract(5, 3);
```

```
console.log(`Sum: ${sum}`);

console.log(`Difference: ${difference}`);
```

ECMAScript Modules (ESM)

ECMAScript Modules (ESM) are the standardized module system introduced in ECMAScript 2015 (ES6). ESM is designed to be used both in browsers and in Node.js, with an emphasis on asynchronous loading. ESM uses `import` and `export` statements.

Example of ECMAScript Module

math.mjs (ECMAScript Module):

```
// math.mjs

export function add(a, b) {
  return a + b;
}

export function subtract(a, b) {
  return a - b;
}
```

app.mjs (Using the ECMAScript Module):

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

const sum = add(5, 3);

const difference = subtract(5, 3);

console.log(`Sum: ${sum}`);

console.log(`Difference: ${difference}`);
```

Q.3. Note on package with example.

Ans.3.

Packages in Node.js

A package in Node.js is a collection of related code files, often organized into modules, that are bundled together and distributed through the npm (Node Package Manager) registry. Packages can include JavaScript code, JSON files, configuration files, and other assets. They can be libraries, frameworks, tools, or complete applications.

Structure of a Node.js Package

A typical Node.js package contains the following components:

1. package.json:

   - This is the main configuration file for a Node.js package. It contains metadata about the package, such as its name, version, description, author, dependencies, and scripts.

2. Module Files:

   - These are the JavaScript files that contain the code for the package. They can be organized into subdirectories.

3. README.md:

   - This file contains documentation for the package, explaining how to install, use, and contribute to it.

4. License:

   - This file specifies the license under which the package is distributed.

Example of a Node.js Package

Step 1: Initialize the Package

First, create a new directory for the package and navigate into it:

mkdir my-math-package

cd my-math-package

Initialize the package with npm:

npm init -y

This command will generate a `package.json` file with default values.

Step 2: Create Module Files

Create a `math.js` file that exports arithmetic functions:

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

function subtract(a, b) {
  return a - b;
}

module.exports = {
  add,
  subtract
};
```

Step 3: Update `package.json`

Update the `package.json` file to include relevant information about the package. For example:

```
{
  "name": "my-math-package",
  "version": "1.0.0",
  "description": "A simple math package",
  "main": "math.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": ["math", "arithmetic", "addition", "subtraction"],
  "author": "Your Name",
  "license": "MIT"
}
```

Q.4. Use of package.json and package-lock.json .

Ans.4.

Both `package.json` and `package-lock.json` are essential files in a Node.js project, serving different but complementary purposes in managing dependencies and project metadata.

package.json

The `package.json` file is the heart of a Node.js project. It contains metadata about the project and its dependencies. Here are its primary uses:

1. Metadata:

   - Defines the project's name, version, description, author, license, and other relevant information.

2. Dependencies:

- Lists the project's dependencies (both runtime and development) and their versions. These dependencies are libraries or packages that the project requires to function properly.

3. Scripts:

- Defines custom scripts that can be run using npm. Common scripts include `start`, `test`, `build`, and `lint`.

4. Entry Point:

- Specifies the main file of the project using the `main` field. This file is the entry point of the application.

5. Keywords:

- Provides an array of keywords related to the project, which helps in making the project discoverable when published on npm.

package-lock.json

The `package-lock.json` file is automatically generated when you install dependencies using npm. It provides a detailed and precise record of the entire dependency tree, ensuring consistent installations across different environments.

1. Exact Versions:

- Records the exact versions of all installed dependencies, including nested dependencies. This ensures that the same versions are installed every time the project is set up, preventing the "works on my machine" problem.

2. Performance:

- Improves the performance of subsequent installations by allowing npm to skip repeated metadata resolutions.

3. Integrity:

- Contains integrity hashes to verify that the downloaded packages have not been tampered with.

4. Consistency:

- Ensures that all team members and deployment environments use the same dependency versions, which helps avoid issues related to differing dependency versions.

Q.5. Nodejs packages.

Ans.5.

Node.js packages are collections of files and modules that add specific functionality to your Node.js applications. These packages are typically distributed through npm (Node Package Manager), which is the default package manager for Node.js. Here are some essential Node.js packages commonly used in development.

Commonly used packages and its examples are:

Express :

- npm install express
- const express = require('express');

Mongoose :

- npm install mongoose
- const mongo = require('mongoose');

Q.6. npm introduction and commands with its use.

Ans.6.

npm (Node Package Manager) is the default package manager for Node.js. It allows developers to install, share, and manage dependencies (packages) for their Node.js projects. npm also provides tools to manage versioning, dependency conflicts, and scripts for project automation.

- Installation: npm is installed automatically with Node.js. You can verify the installation using the following commands:

```
node -v
npm -v
```

- `npm init -y`: Initializes a new Node.js project with default settings, skipping the interactive prompts.

```
npm init -y
```

- `npm install <package-name>`: Installs a package locally in the `node_modules` directory and adds it to the `dependencies` in `package.json`.

```
npm install express
```

-`npm install`: Installs all dependencies listed in the `package.json` file.

```
npm install
```

- `npm install <package-name> --save-dev`: Installs a package as a development dependency and adds it to `devDependencies` in `package.json`.

```
npm install jest --save-dev
```

- `npm uninstall <package-name>`: Uninstalls a package and removes it from the `dependencies` in `package.json`.

```
npm uninstall express
```

- `npm uninstall -g <package-name>`: Uninstalls a globally installed package.

```
npm uninstall -g nodemon
```

- `npm update <package-name>`: Updates a package to the latest version that matches the version range specified in `package.json`.

```
npm update express
```

- `npm list`: Lists all installed packages and their dependencies.

```
npm list
```

- `npm list -g`: Lists all globally installed packages.

```
npm list -g
```

- `npm view <package-name>`: Displays information about a package in the npm registry.

  npm view express

- `npm info <package-name>`: Another command to view information about a package.

  npm info express

- `npm publish`: Publishes the package to the npm registry. You need to have a user account on npm.

  npm publish

- `npm unpublish`: Removes a package from the npm registry.

  npm unpublish <package-name>

Q.7 & 8. Describe use and working of following Nodejs packages. Important properties and methods and relevant programs.

Ans.7 & 8.

`url`

Use: The `url` module provides utilities for URL resolution and parsing.

Important Properties and Methods:

- `url.parse(urlString)`: Parses a URL string and returns a URL object.
- `url.format(urlObject)`: Returns a formatted URL string.
- `url.resolve(from, to)`: Resolves a target URL relative to a base URL.

Example:

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

const myURL = url.parse('https://example.com:8080/path/name?foo=bar#baz');
```

console.log(myURL.hostname); // 'example.com'

console.log(myURL.port); // '8080'

console.log(myURL.query); // 'foo=bar'

`process`

Use: The `process` object provides information and control over the current Node.js process. It can be accessed from anywhere.

Important Properties and Methods:

- `process.env`: Returns an object containing the user environment.

- `process.argv`: An array containing the command-line arguments.

- `process.exit([code])`: Ends the process with the specified code.

- `process.cwd()`: Returns the current working directory.

Example:

console.log(process.env.NODE_ENV); // Prints 'development', 'production', etc.

console.log(process.argv); // Command-line arguments

process.exit(0); // Exits the process

`pm2` (external package)

Use: `pm2` is a process manager for Node.js applications. It allows you to keep applications alive forever, reload them without downtime, and facilitate common system admin tasks.

Important Properties and Methods:

- `pm2.start(script, options)`: Start a new process.

- `pm2.stop(process)`: Stop a process.

- `pm2.restart(process)`: Restart a process.

Example:

Install pm2 globally

npm install -g pm2

Start an app

pm2 start app.js

List all processes

pm2 list

Stop an app

pm2 stop app

Restart an app

pm2 restart app

`readline`

Use: The `readline` module provides an interface for reading data from a readable stream (like `process.stdin`).

Important Properties and Methods:

- `readline.createInterface(options)`: Creates a new `readline.Interface` instance.

- `interface.question(query, callback)`: Displays a query and waits for user input.

- `interface.close()`: Closes the interface.

Example:

const readline = require('readline');

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

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

`fs`

Use: The `fs` module provides an API for interacting with the file system.

Important Properties and Methods:
- `fs.readFile(path, options, callback)`: Reads the contents of a file.
- `fs.writeFile(path, data, options, callback)`: Writes data to a file.
- `fs.appendFile(path, data, options, callback)`: Appends data to a file.
- `fs.unlink(path, callback)`: Deletes a file.

Example:

```
const fs = require('fs');
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
fs.writeFile('example.txt', 'Hello, world!', (err) => {
  if (err) throw err;
```

```
  console.log('File written successfully');
});
```

`events`

Use: The `events` module provides the EventEmitter class for implementing event-driven programming.

Important Properties and Methods:
- `EventEmitter.on(event, listener)`: Registers an event listener.
- `EventEmitter.emit(event, [...args])`: Emits an event.

Example:

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

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('An event occurred!');
});
myEmitter.emit('event');
```

`console`

Use: The `console` module provides a simple debugging console.

Important Properties and Methods:
- `console.log(message, [...args])`: Prints to stdout.
- `console.error(message, [...args])`: Prints to stderr.

- `console.warn(message, [...args])`: Prints a warning.

Example:

```
console.log('Hello, world!');
console.error('An error occurred!');
console.warn('This is a warning!');
```

`buffer`

Use: The `buffer` module provides a way to handle binary data.

Important Properties and Methods:

- `Buffer.alloc(size)`: Creates a new buffer of the specified size.
- `Buffer.from(string, encoding)`: Creates a new buffer from a string.

Example:

```
const buffer = Buffer.alloc(10);
console.log(buffer);

const bufferFromString = Buffer.from('Hello, world!', 'utf8');
console.log(bufferFromString.toString());
```

`querystring`

Use: The `querystring` module provides utilities for parsing and formatting URL query strings.

Important Properties and Methods:

- `querystring.parse(str)`: Parses a query string into an object.
- `querystring.stringify(obj)`: Converts an object into a query string.

Example:

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

const parsed = querystring.parse('foo=bar&abc=xyz&abc=123');
console.log(parsed);

const stringified = querystring.stringify({ foo: 'bar', abc: ['xyz', '123'] });
console.log(stringified);
```

`http`

Use: The `http` module provides the functionality for creating HTTP servers and making HTTP requests.

Important Properties and Methods:
- `http.createServer(callback)`: Creates a new HTTP server.
- `http.request(options, callback)`: Makes an HTTP request.

Example:

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

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, world!\n');
});

server.listen(3000, '127.0.0.1', () => {
  console.log('Server running at http://127.0.0.1:3000/');
```

});

`v8`

Use: The `v8` module exposes APIs for interacting with the V8 JavaScript engine.

Important Properties and Methods:

- `v8.getHeapStatistics()`: Returns an object with heap statistics.
- `v8.getHeapSpaceStatistics()`: Returns an array with heap space statistics.

Example:

```
const v8 = require('v8');

console.log(v8.getHeapStatistics());
```

`os`

Use: The `os` module provides operating system-related utility methods and properties.

Important Properties and Methods:

- `os.platform()`: Returns the operating system platform.
- `os.cpus()`: Returns an array of objects containing information about each CPU/core.
- `os.totalmem()`: Returns the total amount of system memory.

Example

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

console.log('Platform:', os.platform());
console.log('CPUs:', os.cpus());
console.log('Total Memory:', os.totalmem());
```

`pm2`

Use: `pm2` is a production process manager for Node.js applications, allowing you to keep applications running, restart them without downtime, and manage various aspects of app performance.

Important Properties and Methods:

- `pm2.start(script, options)`: Starts a new process.
- `pm2.stop(process)`: Stops a process.
- `pm2.restart(process)`: Restarts a process.

Example:

```
# Install pm2 globally
npm install -g pm2

# Start an app
pm2 start app.js

# List all processes
pm2 list

# Stop an app
pm2 stop app

# Restart an app
pm2 restart app
```

`zlib`

Use: The `zlib` module provides compression and decompression functionalities.

Important Properties and Methods:

- `zlib.gzip(buffer, callback)`: Compresses a buffer using Gzip.
- `zlib.gunzip(buffer, callback)`: Decompresses a Gzip buffer.
- `zlib.deflate(buffer, callback)`: Compresses a buffer using Deflate.
- `zlib.inflate(buffer, callback)`: Decompresses a Deflate buffer.

Example:

```
const zlib = require('zlib');
const input = 'Hello, world!';

zlib.gzip(input, (err, buffer) => {
  if (!err) {
    console.log('Compressed:', buffer);
    zlib.gunzip(buffer, (err, output) => {
      if (!err) {
        console.log('Decompressed:', output.toString());
      }
    });
  }
});
```