Assignment : 1

Name : Patel Khushi S.

Roll no. : 47

Sem : MSc (IT) - 7th

Subject : Application Development using Full stack.

Q.1 Node is : Introduction, features, execution architecture.

→ Node is is an open-source, Cross-Platform, server-side Javascript runtime environment that allows developers to build scalable and efficient network applications. It was first introduced in 2009 by Ryan Dahl and has since gained immense popularity in the web development community. Node is enables developers to run Javascript code outside of the browser, making it suitable for building server-side applications and real-time web application.

⇒ Fectures of Node.js :-

- Asynchronous and Non-blocking Ilo :-

Node is built on the principle of non-blocking, event-driven architecture It employs on event loop that allows asynchronous processing of Ilo operation This means that Node is applications can handle multiple requests concurrently without getting blocked, making it highly efficient and suitable for handling a large number of concurrent connection.

- Single-threaded and Event-Driven :-

Node is runs on a single-threaded event

loop, which helps in handling concurrent requests efficiently, while traditional server-side technologies create separate threads for each request, Node.js uses a single thread to handle multiple requests. When a request is received, it triggers an event and the event loop manages the asynchronous I/O operation.

→ Fast Execution :-

Node.js is built on Google V8 JavaScript engine, which compiles JavaScript code into native machine code. This allows Node.js applications to execute with high speed and efficiency. V8 engine is also used in Google Chrome browser, which ensures a consistent and high-performance execution environment for JavaScript.

→ NPM (Node Package Manager) :-

Node.js comes with a powerful package manager called NPM, which allows developers to install, publish and manage packages and dependencies easily. NPM hosts a vast repository of open-source package that can be used to extend the functionality of Node.js applications. This extensive ecosystem of libraries and modules significantly accelerates the development process.

-> Extensible and Modular :

Node.js encourages a modular approach to development, allowing developers to break down applications into smaller, reusable modules. This modularity fosters code reusability, maintainability and scalability, making it easier to manage large codebase.

=> Execution Architecture of Node.js :

-> Event Loop :

The event loop is the core of Node.js's asynchronous and non-blocking architecture. It continuously runs and waits for events to occur. When an event is detected, it triggers the corresponding callback function, and the event loop moves on to the next event. This loop allows Node.js to handle multiple requests concurrently without blocking the execution of other code.

-> Callbacks :

Callbacks are a fundamental part of Node.js development. When asynchronous operations complete, they trigger callback functions, allowing developers to handle the results or errors accordingly. Callbacks are passed as arguments to asynchronous

functions, making it possible to perform action after specific operation complete.

→ Event Emitters :-

Event Emitters are objects in Node.js that can emit named events and attach listeners to those events. They form the basis of many core Node.js modules and are widely used for handling various asynchronous operations.

→ Non-blocking I/O :-

Node.js utilizes non-blocking I/O operations, enabling it to handle multiple request simultaneously without waiting for one operations to complete before moving to the next. This approach makes Node.js highly efficient, especially in applications that require frequent I/O operations.

→ Threads and Concurrency :-

Though Node.js runs on a single thread, it employs a thread pool for executing ~~the~~ certain I/O operations like file system access. This allows Node.js to take advantage of multi-core processors while still maintaining its single-threaded, event-driven nature.

→ Modules and NPM :-

Node is encourages the use of modules, which are independent units of functionality that can be reused across applications. These modules are managed using NPM, which simplifies dependency management and makes it easy to integrate third-party libraries.

Q.7 ~~Nod~~ Note on modules with example :-

→ Modules are fundamental concept in Node is that allows developers to organize code into reusable and maintainable units. In Node is, a module is essentially a separate file containing JavaScript code that can be imported and used in other files. This modularity is a key factor in making Node is applications manageable and scalable. documents, we will explore the concept of modules in Node is

⇒ Create Modules in Node is :-

To create a module in Node is, we simply define the desired functionality within a separate file and export in for use in other parts of the application.

-> Create a new file :-

Start by creating a new file with a descriptive name that reflects the purpose of the module for example, let's create a file called "mathutils.js" that will contain some basic math functions

-> Define the functionality :-

Within the "mathutils.js" files, define the functions you want to make available as part of the modale for example :-

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

function multiply (a,b)
{
   return a*b;
}

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

- Export the module :-

To make the function available outside. In this example, we are exporting the 'add' and 'multiply' functions.

=> Benefits of using modules :-

-> Code organization and Reusability :-

Modules allow developers to organize code into logical units, making it easier to manage and maintain larger projects. By breaking down functionality into modules, code reusability is promoted, reducing the need for redundant code.

-> Encapsulation :-

Modules provide a level of encapsulation, as the internal details of a module are hidden from the outside scope. Only the functions and variables explicitly exported through 'module.exports' are accessible externally. This helps in preventing unintended modifications to module internals.

-> maintainability and collaboration :-

The use of modules facilitates collabration among developers working on the same project. Different team members can work on separate module indepenently, and as long as the module's interface remains consistent, changes in one module are less likely to effect others.

Q3 Note on Package with example.

-> In Node.js a Package refers to a collection of reusable code modules, along with meta data ~~etc~~ about those modules, published and distributed on the Node Package Manager (NPM) registry. Package play a vital role in the Node.js ecosystem, as they allow developers to share and reuse code easily, saving time and effort when building applications.

-> NPM and Package Management:

NPM is the default Package manager for Node.js and it comes bundled with the Node.js installation. NPM Provides a vast collection of open-source Packages that are available for use in Node.js Project. With NPM, developers can install Packages, manage their dependencies, and Publish their own Package for other to use.

-> Using Package in Node.js:

To utilize Package in a Node.js Proied, follow these steps:

- Initializing a New Project:

First, create a new folder for your Project and open a terminal inside that directory. Run

the following command to initialize the Project and create a 'package.json' file.

npm init

- ~~Installing Dependencies~~ :-
- Initialize package and Publish :-

In the terminal, navigate to the "greet-utils" folder and run "npm init' to initialize the package. Then, run 'npm Publish' to Publish the package to NPM.

- Use the Published Package :-

In another Node.js Project, you can now use the "greet-utils' package: with following command.

Const greet = require ('greet-utils');

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

=> In node.js Projects, 'Package.json' and 'Package-lock.json' are essential files used for package management and dependency tracking They play a crucial role in ensuring consistency in the Project's dependencies, which is vital for Collaboration and deployment.

=> Packag.json :-

The 'Package.json' file is a metadat file in

Json format that provides essential information about a Node.js project and its dependencies. It serves as the entry point for package management and acts as a manifest for the project.

- When someone wants to use your project, they can check the 'package.json' file to understand its name, version, dependencies, and other relevant details.

⇒ Package-lock.json :

The 'Package-lock.json' file was introduced in npm version 5 to address issues related to dependency resolution and version conflicts. It provides a deterministic representation of the project's dependency tree, ensuring that every installation of the project results in the same set to dependencis with the same version.

- The 'Package-lock.json file contains as exact, nested representation of the project's dependencies. It includes the package names, versions, and the specific versions, and the spesific versions of their dependencies, along with other metadata.

- Automatic Generation :-
The 'package-lock.json' file is

automatically generated by npm whenever you perform certain operations, such as installing new packages or updating existing ones using the 'npm install' command.

- Commit to version control :
  It is essential to commit both 'package.json' and 'package-lock.json' to version control to ensure that the project's dependencies are consistent across team members and deployments.

Q.5 Node.js Packages :-

→ Node.js packages are a fundamental aspect of the Node.js ecosystem, enabling developers to extend the functionality of their applications and leverage the power of open-source code.

⇒ The significance of package in Node.js :-

- Code Reusability : Packages allow developers to reuse existing code written by others. This promotes a modular approach to development, when developers can focus on building specific features without reinventing the wheel.

- Community collaboration : Node.js has a thriving community that actively contributes to the creation and maintenance

of packages. This community-driven approach results in a rich and diverse ecosystem of packages, ensuring that developers have access to a wide rage of solution for their needs.

- Dependency Management :- Using Packages through Package managers like Npm or Yarn allows for efficient dependency management.

⇒ Types of Node.js Packages :-

- Core Package :- Core Packages refer to the built-in modules provided by Node.js itself. These modules cover essential functionalities, such as working with file systems, networking, streams and more. it's not require seperate installation and are avilable out of the box where using node.js.
ex: fs, http, Path

- Third-Party Package :- It's also known as NPM Package, are created and maintained by developers outside of Node.js core team. These Packages are published on the Npm registry.
ex: express, lodash, mogoose

Q.6 npm introduction and commands with its use.

→ NPM is powerful and widely used package manager for Node.js applications. It is the default package manager that comes bundled with Node.js installations, making it an integral part of Node.js ecosystem. NPM facilites the installation, management, and sharing of Node.js Packages, allowing developers to ~~es~~ easily integrate third-party libraries, manage project dependencies, and streamline the development process.

⇒ Core Functionalities of NPM :-

- Package Installation :-
  One of the primary functionalities of NPM is installing Node.js packages. By using NPM's 'install' command, NPM will automatically download the specified package and its dependencies, and store them in the 'node-modules' directory.

- Dependency Management :-
  NPM simplifies the management of project dependencies. When you install a package, NPM automatically adds it to the 'dependencies' section of the 'package.json' file, along with its version number.

- Version management :-
  NPM supports semantic versioning, which helps maintain backward compatibility and ensures that updates to packages don't introduce breaking changes.

⇒ Commonly used NPM commands

- 'npm init' :-
  The 'npm init' command initializes a new Node.js project and generates a 'package.json' file interactively. It prompts the user to provide information such as project name, version, description, entry point, and more.
  ex npm init

- 'npm install' :-
  This command is used to install packages listed in the 'dependencies' section of the 'package.json' file. It will download the specified packages and their dependencies store them in the 'node_modules' directory.
  ex npm install

- 'npm install <package-name>' :-
  To install a specific package, we can use this command.
  ex npm install express

- 'npm uninstall <package-names>' :-
  To remove a package from your

project, use the 'npm uninstall' command.
ex npm unistall lodash

- `npm update' :
The 'npm update' command will update packages to their latest versions, based on the version constraints specified in the 'package.json' file.
ex npm update.

- `npm search <package-name>' :
To search for packages available on the NPM registry, you can use this command.
ex npm search moment.

Q.7. Describe use and working of following Nodejs Packages, Important properties and methods and relevant programs.

⇒ url :

-> Description: The 'url' module provides utilities for parsing and formatting URL strings. It allows developers to work with URLs by providing methods to parse query strings, extract URL components and manipulate URL parameters.

-> Important properties and methods:

- 'url.parse(urlString[, parseQueryString[, slashesDenoteHost]])' :- Parses a URL string and returns an object with its components

- 'url.format(urlObject)':
  Formats a URL objects into a URL string.

- 'url.resolve(from, to)':
  Resolves a relative URL 'to' relative to the base URL 'from'.

-> Relevant Program:

```
Const url = require('url');
Const urlString = 'https://www.example.com/search?
q=nodejs&page=1';
const parsedUrl = url.parse(urlString, true);
console.log('Parsed URL:', parsedUrl);

Const formattedUrl = url.format(parsedUrl);
console.log('formatted URL:', formattedUrl);

Const resolvedUrl = url.resolve('https://www.example
com/', '/about');
Console.log('Resolved URL:', resolvedUrl);
```

=> Process :

-> Description : The 'process' module provides access to information about the current Node.js process. It allows developers to interact with the underlying process and provides various events

and methods to control the Node.js application's behavior.

⇒ Important Properties and methods :

- 'Process.argv' :
  An array containing the command-line arguments passed to the Node.js process.

- 'Process.env' :
  An object containing the user environment variables.

- 'Process.exit([code])' :
  Exits the current process with an optional exit code.

- 'Process.on(event, callback)' :
  Sets up event listeners for various process events (e.g., 'exit', 'uncaughtException');

⇒ Relevant Program :

```
Console.log('Command line argument:', Process.argv);
Console.log('user Environment variables:', Process.env);

Process.on('exit', (code) => {
  Console.log('Exiting with code: ${code}');
});

Process.on('uncaughtException', (error) => {
  Console.error('uncaughtException:', error); });
```

⇒ Pm2 :- (external package)

→ Description :- 'Pm2' is an external package that provides process management for Node.js applications. It is a production process manager that allows developers to monitor and manage Node.js processes, enabling automatic restarts on failures and providing features for scaling applications.

→ Important properties and method :-

- 'Pm2 start <app.js> :-
  starts a Node.js application using 'Pm2'.

- Pm2 list :-
  Listing of all running processes managed by 'pm2'.

- Pm2 stop <app_name|id> :-
  stops a specific application process.

- Pm2 delete <app_name|id> :-
  Deletes a specific application from 'Pm2'.

→ Relevant Program :-

- npm install -g pm2 :- install Pm2 globally
- Pm2 start app.js :- start a Node.js application
- Pm2 list :- Monitor running processes.
- Pm2 stop app.js / Pm2 delete app.js :- stop or delete an application.

⇒ readline :-

-> Description :- The 'readline' module provides an interface for reading input from readable stream and writing output to writable streams. It is useful for building interactive command-line applications.

-> Important properties and methods :-

- readline.createInterface(options) :-
  creates a new readline instance.

- rl.question(query, callback) :-
  Asks a question and waits for user input.

- rl.close() :-
  Closes the readline interface.

-> Relevant program :-

```
Const readline = require('readline');
Const rl = readline.createInterface({
    input: Process.stdin,
    output: Process.stdout,
});
rl.question('What is your age?', (age) => {
    console.log('Hello, ${age}');
    rl.close();
});
```

⇒ fs :-

→ Description :- The 'fs' module provides the system-related functionalities to read, write, and manipulate files and directories. It allows developers to perform synchronous and asynchronous file operations.

→ Important properties and methods :-

- fs.readFile (path[, options], callback) :-
  Readfile asynchronously.

- fs.readFilesync (path[, options], callback) :-
  Reads a file synchronously.

- fs.writeFile (file, data [, options], callback) :-
  writes data to a file asynchronously.

- fs.readdir (Path[, options], callback) :-
  Reads a directory asynchronously.

→ Relevant Program :-

```
Const  fs = require ('fs');
fs.readFile ('example.txt', 'utf-8', (err, data) => {
    if (err) throw err;
    Console.log ('File content:', data);
});
fs.writeFilesync ('example.txt', 'utf-8' Hellow', 'utf-8')
Console.log ('file written Done!');
```

=> events :-

-> Description :- The 'events' module provides a simple and efficient event-driven programming paradigm for building custom events and handling event emitters.

-> Important properties and methods :-

- event.EventEmitter :-
  The class that represents an event emitter.

- event.on(event, listener) :-
  Adds a listener for a specific event.

- event.emitter(event[, ...args]) :-
  Emits an event with optional arguments.

-> Relevant program :-

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();

myEmitter.on('greet', (name) => {
    console.log('Hello', ${name}!');
});

myEmitter.emit('greet', 'Khushi');
```

Date: YOUVA

⇒ Console :-

→ Description :- The 'console' module provides methods for printing data to the console and debugging Node.js application.

→ Important Methods :-

- Console.log() :-
  Prints a message to the console with a newli

- Console.error() :-
  Prints a error message to the console.

- Console.warn() :-
  Prints a warning message to the console.

→ Relevant Program :-

```
Console.log('Hello, Node.js!');

Const x=5;
Const Y=10;
Console.log('The sum is:', x+y);

Console.error('This is error msg');
Console.warn('This is waring msg');
```

⇒ buffer :-

→ Description :- The 'buffer' module provides a way to handle binary data in Node.js. It is

useful for handling streams of data, such as reading and writing files or handling network data.

→ Important properties and methods:-

- Buffer :
  The class that represents a buffer.

- Buffer.from(data) :-
  creates a new buffer from the specified

- Buffer.alloc(size) :
  Creates a new buffer of the specified size.

→ Relevant Program :

```
Const buf1 = Buffer.from('Hello');
Const buf2 = Buffer.from([0x48, 0x65, 0x6c, 0x6c, 0x6f]);
Const buf3 = Buffer.alloc(5);

Console.log(buf1.toString()); // Hello
Console.log(buf2.toString()); // Hello
Console.log(buf3.toString()); // \u0000\u0000\
                              u0000\u0000\u0000
```

⇒ queryString :

→ Description : The 'queryString' module provides method for parsing and formatting URL query strings. It is useful for working with the query parameters of a URL.

→ Important Properties and methods :-

- querystring.parse(str[, sep[, eq[, option]]]) :-
  Parses query string and returns an obied.

- querystring.stringify(obi[, sep[, eq[, options]]]) :-
  converts and object to a query string.

→ Relevant program :-

```
const querystring = require('querystring');
const query string = 'q=nodejs&lang=en';
const parsedQuery = querystring.parse(queryString)
console.log(parsedQuery);

const obi = {name: 'Khushi', age: 20};
const queryStringified = querystring.stringify(obi)

console.log(queryStringified);
```

⇒ http :-

→ Description :- The 'http' module provides functionality to create HTTP servers and make HTTP requests. It is the foundation for building web servers and clients in Node.j

→ Important Properties and methods :-

- http.createserver([options][, requestListener]) :-
  creates an HTTP server.

- server.listen ([port][,hostname][,backlog][,callback])
  start the HTTP server to listen for incoming requests.

-> Relevant Program :

```
Const https =require ('http');

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

Const Port = 3000;
Const hostname = '127.0.0.1';

Server.listen (Port, hostname, () => {
  Console.log ('server listen 3000 Port');
});
```

=> V8 :

-> Description : The V8 module provides access to the V8 Javascript engine's API. It allows developers to access V8 engine information and statistics.

-> Important properties and methods :

- V8.getHeapStatistics () :
  return statistic about the V8 heap memory usage.

- v8.getHeapSpaceStatistics() :
  return statistics about the v8 heap space memory usage.

-> Relevant program :

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

Const heapStats = v8.getHeapStatistics();
Console.log('Heap statistics:', heapStats);

Const heapSpaceStats = v8.getHeapSpaceStatistics();
Console.log('Heap Space statistics:' heapSpaceStats);
```

=> os :

-> Description : The 'os' module provides operating system-related utilities. It allow developers to retrieve information about the operating system, such as hostname, network interfaces, and CPU architecture.

-> Important properties and methods :

- os.hostname() :
  Returns the hostname of the os.

- os.platform() :
  Returns the os platform.

- os.cpus() :
  Returns an array of object containing informa about CPUs.

-> Relevant Program :-

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

Console.log('Hostname:' os.hostname());
Console.log('Platform:' os.platform());
Console.log('CPUs:', os.cpus());
```

=> zlib :-

-> Description : The 'zlib' module provides compression and decompression functionalities. It is useful for working with compressed data, such as gzip and deflate.

-> Important properties and methods :

- zlib.gzip(input [, options], callback) :-
  Compresses data using gzip.

- zlib.gunzip(buffer [, options], callback) :-
  Decompresses data compressed using gzip

- zlib.inflate(buffer [, options], callback) :-
  Decompresses data compressed using deflate

-> Relevant program :-

```
const zlib = require('zlib');
const input = 'Hello';
const compressed = zlib.gzipSync(input);
```

```
console.log('compressed:', compressed);

const decompressed = zlib.gunzipSync(compressed)
                        .toString();
console.log('Decompressed:', decompressed);
```