Name :- Vanaiya Khushi Hirenkumar

Semester :- 7

Division : A

Roll no :- 77

Subject :- Application Development using Full Stack.

Assignment :- Theory Assignment 01

Q Nodejs :- Introduction, features, execution architecture.

→

Node.js 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.js enables developers to run Javascript code outside of the browser making it suitable for building server-side applications and real-time web application.

★ Features of Node.js.

1) Asynchronous and Non-blocking I/O :-

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

2) Single-threaded and Event-Driven :-

Node.js 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 events, and the event loop manages the asynchronous I/O operation.

3) 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.

4) NPM (Node Package Manager) :-

Node.js comes with a powerful package manager called NPM, which allows developers to install, publish and manages 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.

5) 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 codebases.

★ Execution Architecture of Node.js.

1) 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.

2) 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.

3) 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 operation:

4) Non-blocking I/O :-

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

5) Threads and Concurrency :-

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

6) Modules and NPM :-

Node.js 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 Note on ⊛ modules with example.

→

Modules are fundamental concept in Node.js that allows developers to organize code into reusable and maintainable units. In Node.js, 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.js applications manageable and scalable. In this document, we will explore the concept of modules in Node.js

★ Create Modules in Node.js.

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

★ Define the functionality

1 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.

2 Define the functionality:- Within the "mathUtils.js" files, define the functions you want to make available as part of the module. For example:-

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

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

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

1. Export the Module :- To make the function available outside the module, we use the 'module.exports' object. In this example, we are exporting the 'add' and 'multiply' functions.

✱ Benefits of Using Modules.

1) 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 functionali into modules, code reusability is promoted, reducing the need for redundant code.

2) Encapsulation :-

Modules allow developers to organize code into logical units, making it easier to manage

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.

3) Maintainability and Collaboration :-

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

Q Note on Package with example ?

→

In Node.js a package refers to a collection of reusable code modules, along with meta data about those modules, published and distributed on the Node Package Manager (NPM) registry. Packages 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. [illegible]

★ NPM and Package Management :-

NPM is the default packages 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 packages for other to use.

★ Using Packages in Node.js :-

To utilize packages in a Node.js project, follow these steps:

1) 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 ~~object~~ project and create a 'package.json' file:-

```
npm init.
```

2) Installing Packages :-

Once ~~you~~ we have a 'package.json' file, you can start installing packages using the 'npm install' command followed by the package name. For example,:-

```
npm install lodash
```

3) using Packages in your Code :-

After installing a package, we can start using its functionality in your Node.js code. To do this, require the package in your file and use the provided functions or classes.
For example :-

```
const _ = require('lodash');
const numbers = [1, 2, 3, 4, 5];
const sum = _.sum(numbers);
console.log('Sum:', sum);
```

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

→

'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.

★ Package.json :-

The 'package.json' file is a metadata 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.

★ 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 of dependencies with the same versions

★ Use of package.json :-

To create a 'package.json' file for our Node.js project, run the following command.

- npm init

This command will prompt you to enter various details about your project, and once completed, it will generate a 'package.json' file.

To install the dependencies listed in 'package.json'

- npm install

This will download and install all the packages specified in the 'dependencies' section. The installed packages will be stored in a 'node_modules' folder.

To add a new dependency to your project, we can use the following command :-

° npm install <package-name>

This will add the package to the 'dependencies' section in the 'package.json' file and install it in the 'node_modules' folder.

★ Use of Package-lock.json.

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.

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. This guarantees that everyone working on the project is using the same set of dependencies with the same versions.

★ Difference between package.json and package-lock.json.

1) Purpose :- The 'package.json' file is a manifest containing metadata about the project and its dependencies, while the 'package-lock.json' file ensures deterministic dependency resolution by specifying the exact versions of all packages.

2) Manual vs Automatic Editing :- Developers typically manually edit 'package.json' to add or

update dependencies, while 'package-lock.json' is automatically generated by npm and should not be manually modified.

3) Commit to Version Control :- Both files should be committed to version control, but it is particularly crucial to commit 'package-lock.json' to ensure consistent dependencies across all environments.

4) Usage in production :- When deploying a project in production, 'package-lock.json' is used to install the exact specified versions of dependencies, guaranteeing consistency between development and production environments.

Q Short note on 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 Packages in Node.js

1) Code Reusability :-

Packages allow developers to reuse existing code written by others. This promotes a modular approach to development, where developers can focus on building specific features without reinventing the wheel. By leveraging well-maintained

and thoroughly tested packages, developers can save time and effort, leading to faster development cycles.

2) Enhanced Functionality:-

The Node.js package ecosystem offer a vast array of modules covering various functionalities, such as database integration, authentication, encryption and much more. By using packages, developers can quickly enhance their application.

3) 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 range of solutions for their needs.

4) Dependency Management:-

Using packages through package managers like NPM (Node Package Manager) or Yarn allows for efficient dependency management. These package managers handle the installation, versioning, and resolution of package dependencies, ensuring that the project stays consistent across different environments and collaborators.

★ Types of Node.js packages.

1) Core Packages :- Core packages refers to the built-in modules provides by Node.js itself.
examples :- 'fs', 'http', 'https', & 'path'

2) Third-Party Packages :- Third-Party packages, also known as NPM packages, are created and maintained by developers outside of the Node.js core team.
Examples :- 'express', 'lodash', & 'mongoose'

★ NPM (Node Package Manager) :-

NPM is the default package manager for Node.js, and it comes bundled with the Node.js installation. It provides a command-line interface for installing, managing and publishing packages.

- 'npm init'
- 'npm install <package-name>'
- 'npm install'
- 'npm update <package-name>'
- 'npm uninstall <package-name>'

Q → npm introduction and commands with its use.

NPM is a 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 the Node.js ecosystem. NPM facilitates the installation, management and sharing of Node.js packages, allowing developers to easily integrate third-party libraries, manage project dependencies, and streamline the development process.

★ Core Functionalities of NPM.

1. Package Installation :-

One of the primary functionalities of NPM is installing Node.js packages. By using NPM's 'install' command, developers can quickly add packages to their projects. NPM will automatically download the specified package and its dependencies, and store them in the 'node_modules' directory.

2. 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.

3) Version Management :-

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

★ NPM Commands

1) 'npm init' :- The 'npm init' command initializes a new Node.js project and generates a 'package.json' file interactively.

Syntax :- npm init

2) 'npm install' :- The 'npm install' command is used to install packages listed in the 'dependencies' section of the 'package.json' file.

Syntax :- npm install

3) 'npm install <package-name>' :- To install a specific package, you can use the 'npm install' command followed by the package name.

Syntax :- npm install express

4) 'npm install <package-names> --save-dev' :- If you want to install a package as a development dependency (i.e, only required during development, not in production), you can use the '--save-dev' or '-D' flag.

Syntax :- npm install jest --save-dev.

5) npm uninstall <package-name> :- To remove a package from your project, use the 'npm uninstall' command followed by the package name.

Syntax :- npm uninstall lodash

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

Syntax :- npm update.

7 'npm search <package-name>' :- To search for package available on the NPM registry, you can use the 'npm search' command followed by the package name.

Syntax :- npm search moment.

8 'npm publish' :- To publish your own package to the NPM registry, you can use the 'npm publish' command. Before publishing, ensure you have an NPM account and have logged in using 'npm login'.

Syntax :- npm publish.

Q Describe use and working of following Node.js packages.

→

1) 'url' :-

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 object 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 = " ";
const parsedUrl = url.parse(urlString, true);
console.log('Parsed URL:', ParsedUrl);
```

2) 'process' :-

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 behaviour

* Important properties and Methods

- 'process.argv'
- 'process.env'
- 'process.exit([code])
- 'process.on(event, callback)

* Program :-

```
console.log('Command Line Arguments :', process.argv);
console.log('User Environment Variable :', process.env);
process.on('exit', (code) => {
    console.log(`Exiting with code : ${code}`);
});
process.on('UncaughtException', (error) => {
    console.error('Uncaught Exception :', error);
});
```

3) 'pm2' :-

'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 failure and providing features for scaling applications.

★ Important Properties and Methods :-

- 'pm2 start <app.js>':
- 'pm2 list'
- 'pm2 stop <app-name|id>
- 'pm2 delete <app-name|id>

★ Program :-

- npm install -g pm2
- Pm2 start app.js
- pm2 list
- pm2 stop app.js
- pm2 delete app.js

4) 'readline' :-

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

* Important properties and Methods :-
- 'readline.createInterface(options)'
- 'rl.question(query, callback)
- 'rl.close()'

* Program :

```
const readline = require('readline')
const rl = readline.createInterface({
    input : process.stdin
    output : process.stdout,
});
rl.question('what is your name ? ', (name) => {
    console.log('Hello, ${name}!');
    rl.close();
});
```

5) 'fs' :-

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

* Important Properties & Method :

- 'fs.readFile(path[, options], callback)'
- 'fs.readFileSync(path[, option])'
- 'fs.writeFile(file, data[, option], callback)'
- 'fs.writeFileSync(file, data[, options])'
- 'fs.readdir(path[, options], callback)'
- 'fs.readdirSync(path[, option])'

* Relevant Program :-

```
const fs = require('fs');
fs.readFile('example.txt', 'utf8', (err, data) =>
 {
    if(err) throw err;
    console.log('file content :', data)
  });

fs.writeFileSync('example.txt', 'Hello, Node.js!', 'utf8');
console.log('file written successfully.');
```

6) 'events' :-

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

* Important Properties and Methods.

- events.EventEmitter
- emitter.on(event, listener)
- eventEmitter.emit(eventname, [args])

* Example :-

```
const event = require('events');
class myemit extends EventEmitter{}

const myemitter = new myEmitter();
myemitter.on('greet', (name) =>
{
  console.log('Hello', ${name}!).
});
myemit.emit('greet', 'john');
```

7) 'Console' :-

The 'Console' module provides methods for printing data to the console and debugging Node.js application.

★ Important Methods.

- Console.log()
- Console.error()
- Console.warn()

★ Program.

```
Console.log('Hello, Node.js');
Const x = 5;
Const y = 10;
Console.log('The sum is:', x+y);
Console.error('This is an error message');
Console.warn('This is a warning message');
```

'Buffer':-

The buffer module provides a way to handle binary data in Node.js. It is useful for handling streams of data, such as ~~data~~ reading and writing files or handling network data.

★ Important properties & Methods

- 'Buffer'
- 'Buffer.from(data)'
- 'Buffer.alloc(size)'

★ Program

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

console.log(buf1.toString());
console.log(buf2.toString());
console.log(buf3.toString());
```

9) 'querystring' :-

The 'querystring' module provides methods for parsing and formatting URL query strings. It is useful for working with the query parameters of a URL.

* Important Methods & properties.

- querystring.parse(str[, sep[, eq[, options]]]);
- querystring.stringify(obj[, sep[, eq[, options]]]);

* Program.

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

const querystring = 'q=nodejs&lang=en';
const parsedQuery = querystring.parse(querystring);

console.log(parsedQuery);
```

10) 'http' :-

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.js.

★ Important properties & Methods.

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

const port = 3000;
const hostname = '127.0.0.1';

server.listen(port, hostname, () => {
  console.log("Server is running at port 3000");
});
```

11 'v8':-

The 'v8' module provides access to the V8 Javascript engine's API. It allows developers to access V8 engine information and statistics.

* Important methods & properties.

- 'v8.getHeapStatistics()'
- 'v8.getHeapSpaceStatistics()'

* Program :-

```
Const v8 = require('v8');
const heapstats = v8.getHeapStatistics();
Console.log('Heap Statistics:', heapstats);
const heapSpaceStats = v8.getHeapSpaceStatistics();
Console.log('Heap space statistics:', heapSpaceStats);
```

12) 'os' :-

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

★ Important properties & Methods.

- 'os.hostname'
- 'os.platform()'
- 'os.cpus()'

★ Program.

```
Const os = Require('os');

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

13) 'zlib' :-

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

* Important Methods & properties.

- 'zlib.gzip(input[, options], callback)'
- 'zlib.gunzip(buffer[, options], callback)'
- zlib.deflate(input[, options], callback)'
- zlib.inflate(buffer.[, options], callback)'

* Program :-

```
Const zlib = require('zlib');
Const input = 'Hello, ';
Const compressed = zlib.gzipSync(input);
Console.log('Compressed:', compressed);
```