Name : Savaliya Heli Pareshbhai

Semester : 7th

Roll No : 71

Subject : Application Development using Full Stack

ASSIGNMENT - 1

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

* Introduction:

Node.js is an open-source and cross-platform runtime environment for executing Javascript code outside a browser.

→ Nodejs is not a framework and it's not a programming language.

→ We often use Node.js for building back-end services like APIs like Web App or Mobile App.

Node.js = Runtime Environment + Javascript Library

* Features:

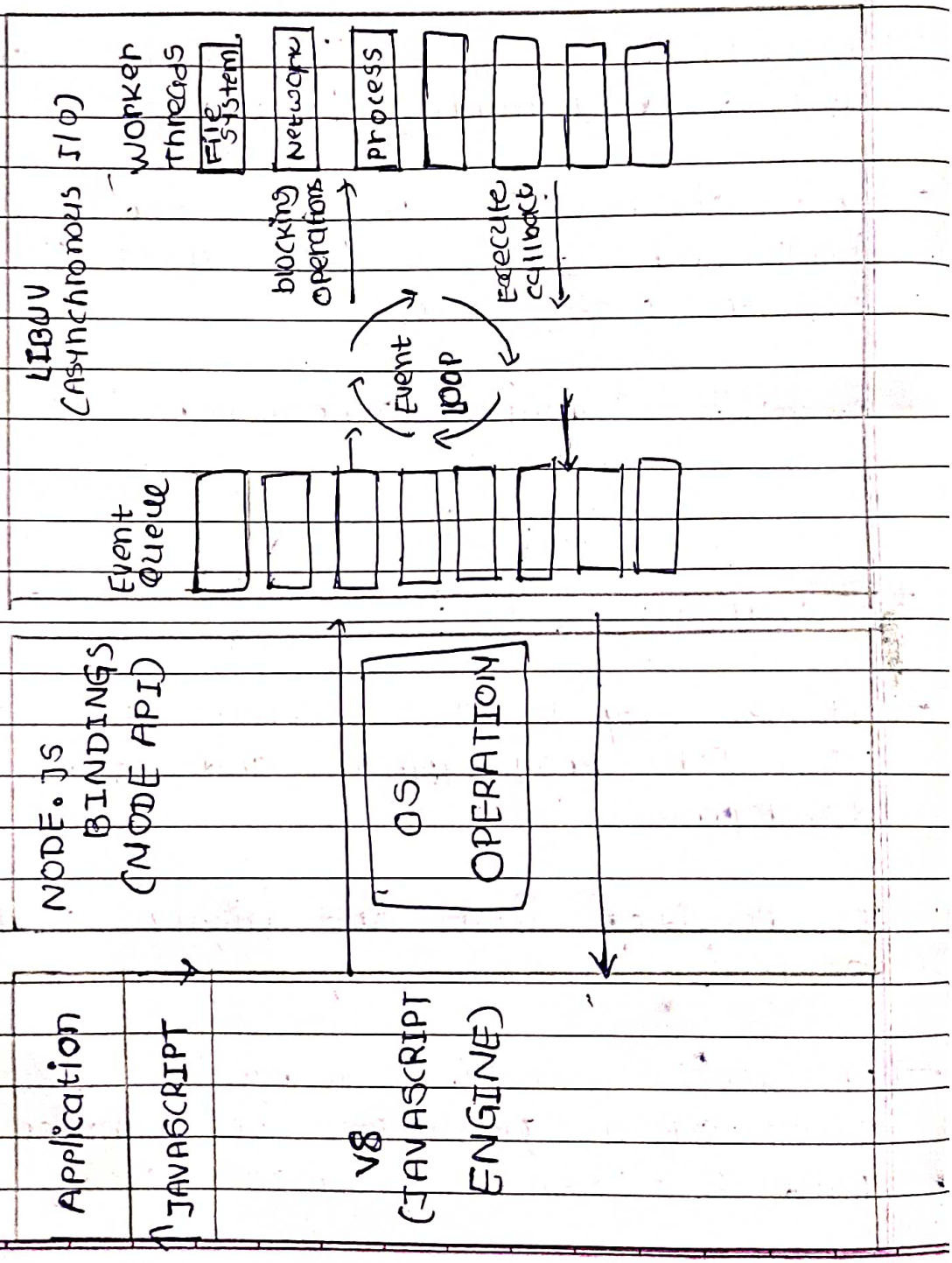
→ It is easy to get started and can be used for prototyping and agile development.

→ It provides fast and highly scalable services.

→ It uses Javascript everywhere, so it's easy for a Javascript programmer to built back-end services using Node.js

→ Source code cleaner and consistent.
→ Large ecosystem for open source library.
→ It has Asynchronous or Non-blocking nature.

★ Execution architecture



* Single Threaded Event loop
→ The Event loop in Node.js is single-threaded.
→ All requests may have two parts - synchronous and asynchronous.
→ The event loop takes up the synchronous part and assigns the asynchronous part to a background thread to execute.

*Non-blocking I/O model
→ The main thread in Node.js allocates the time-consuming asynchronous operations to a background thread.
→ It does not wait for the background thread to complete that operations it takes up the next operation in the event queue.

* Requests:
→ Requests are sent by clients to fetch some server or by the server itself to some other server to fetch resources from it.

* Node.js server:
→ It is the server-side or the backend platform that receives requests from users at various endpoints defined.

* Event loop:
→ It is an infinite loop that has six phases. All the six phases are repeated

until their is no code left to execute

The six phases of event loop are:
[1] Timers
[2] I/O callbacks
[3] waiting / preparation
[4] I/O polling
[5] setImmediate() callbacks
[6] close events

Date.
Page.

1=2. Note on modules with example.

In Node.js, Modules are the blocks of encapsulated code that communicate with an external application on the basis of their related functionality.

Modules are of three types:
- Core Modules
- Local Modules
- Third-party modules

[1] core modules

Node.js has many built-in modules that are part of the platform and come with Node.js installation. These modules can be loaded into the program using the required function.

*Syntax:

```
const module = require('module name');
```

*Example:

```
const http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {content-Type: 'text/html'});
    });
res.write('welcome to this Page!');
    res.end();
}).listen(3000);
```

Date
Page

[2] Local modules: Φ=3.

Unlike built-in and external modules, local modules are created locally in your Node.js application.

module.js

```
modules.exports.add = function (x, y){
    return x+y;
}
```

Index.js

```
const calc = require('./module');
let x=10, y=50;

console.log('Addition =>' + ~~modu~~ calc.
                add(x,y));
```

[3] Third-party modules:

→ Third-party modules are modules that are available online using the Node package Manager (NPM). These modules can be installed in the project folder or globally.

→ Popular Third Party modules are mongoose, express, angular & React.

* Example:
  - npm install express
  - npm install mongoose
  - npm install -g @angular/cli

Q=5. Note on Package with example.

A package is a collection of code files or modules that serves a specific purpose or provide certain functionalities.

→ Packages are designed to be reusable and shareable, allowing developers to easily incorporate pre-built code into their projects without having to reinvent the wheel.

→ packages also help in organizing code, making it more maintainable and scalable

→ Packages often have dependencies, which are other packages required for them to function correctly.

☆ Example of a Node.js package:

1. Create the project directory & Navigate to it in the terminal.

2. Initialize the project
   ⇒ Run the following command in terminal to initialize the project & generate the 'package.json' file.

```
npm init -y
```

3. create the main module file.
→ Inside the package directory, create a file named 'fact.js' with content---

```
function factorial(n) {
    if (n===0 || n===1) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
module.exports = factorial;
```

4. Test the package

```
const fact = require('./fact');
console.log(factorial(4));
```

5. Package structure.

- package.json
- fact.js
- test.js

6. Publish the package.

```
npm login
npm publish
```

7. Using the package:

```
npm install your-package-name
```

Name: Savaliya Heli P.
Roll No: 71

Date
Page

J.1 Use of 'package.json' and 'package-lock.json'

[1] package.json:

The 'package.json' file is a manifest file that contains metadata about the project and lists all the dependencies required to run the application.

→ It is typically located at the root of the project directory.

* project metadata:
This includes information like the project name, version, description, author, license, etc.

* Dependencies:
These are the external packages that your project depends on.

* Devdependencies:
Similar to dependencies, but these are only required during development and not in the production environment.

* Scripts:
You can define custom scripts in the 'package.json' file that can be executed using npm or yarn.

Name: Savaliya Help
Roll No: 71
Date
Page

To create a 'package.json' file, you can use the following command in root of your project.

npm init

[2] Package-lock.json.

The 'package-lock.json' file is automatically generated when you run 'npm install' or 'npm update' to install or update dependencies in your project.

→ It is used to lock down the exact versions of the packages and their dependencies that were installed.

→ This file ensures that everyone working on the project installs the same versions of the dependencies.

→ It is recommended to commit the 'package-lock.json' file to version control so that other developers or the production environment can install the exact same versions of dependencies used during development.

Name: Savaliya Heli P.
Roll No: 71

Date
Page

2.5 Node.js Packages.

Node.js has a rich ecosystem of packages available through the Node Package Manager (NPM), which is the default package manager for Node.js.

→ Here are some commonly used Node.js packages:

[1] Express:
→ A fast, minimalist, and flexible web application framework for building web applications and APIs.

[2] Lodash:
→ A utility library that provides a wide range of helpful functions for manipulating arrays, objects, strings and more.

[3] Mongoose:
→ An Object Data Modeling (ODM) library for MongoDB, simplifying the interaction with the MongoDB database.

[4] Axios:
→ A popular HTTP client that allows making HTTP requests from Node.js supporting promises and async/await.

[5] Body-parser:
→ Middleware for parsing incoming request bodies in a Node.js application.

Name: Savaliya Heli P.
Roll No: 71

Date
Page

Q-6. NPM introduction and commands with its use.

* Introduction:

→ NPM (Node package Manager) is the default package manager for Node.js It allows developer to easily insta manage, and share packages or librarie of code written in JavaScript.

→ These packages can be reasable module frameworks, or tools that can be integrated into our Node.js projects.

→ NPM also manages dependencies betw Packages, ensuring that the required packages are installed to run our application successfully.

* Commands.

[1] Initializing a new Project (- npm ini
To start a new Node.js project, navigate to our project folder and ru

[2] Installing Packages.
To install a package and add it as a dependency to your project user the 'npm install' comment.

npm install <package-name>

Name: Savaliya Heli P.

Roll No: 71

Date
Page

[3] Installing packages globally.

Some packages are meant to be used globally as command-line tools.

```
npm install -g <Package-name>
```

[4] Installing Packages as dev dependencies:

Dev dependencies are packages that are only needed during development and not for the production runtime. To add a dev dependency, use the '--save-dev' flag.

```
npm install <Package-name>
```

[5] Updating Packages:

To update packages to their latest versions, use the following command:

```
npm update
```

[6] Uninstalling Packages:

To remove a package from your project, use the 'uninstall' command.

```
npm uninstall <Package-name>
```

[7] Running scripts:

In the 'Package.json' file, you can define scripts that execute specific commands.

```
npm run <script-name>
```

Name: Savaliya Heet P.
Roll No: 71

Date
Page

[8] Viewing Installed packages:

→ To see a list of installed Packag and their versions, use the following command.

```
npm list.
```

[9] search for packages:

→ To search for packages on the NP registry, use the 'search' command.

```
npm search <package-name>
```

Name: Savaliya Heli P.
Roll No: 71
Date
Page

Q7 Describe use and working of following Node.js Packages

[1] url

→ The 'url' Package in Node.js is a built-in module that provides utilities for URL resolution and parsing.

→ It allows you to work with URLs, parse them, and manipulate their components easily.

→ This package is especially useful when dealing with web-related tasks, such as making HTTP requests, extracting parts of URLs, or building new URLs.

* working:

(1) Parsing URLs

→ The 'url.parse()' method is used to parse a URL and extract its various components like protocol, hostname, port, path, query parameters and more.

(2) Resolving URLs

→ The 'url.resolve()' method resolves a relative URL against a base URL, providing the absolute URL.

* Important properties:

⇒ protocol

⇒ hostname

Name: Savaliya Heli P.
Roll No: 71

Date
Page

=> port
=> pathname
=> query

Example:

```
const url = require('url');
const baseUrl = 'https://favicon.ico';
const relPath = '/path/to/page';
const resolvedUrl = url.resolve(baseUrl,
    relativePath);
console.log(resolvedPathUrl);
```

[2] process

The 'process' package is a built-in module in Node.js that provides access to information and functionality related to the current Node.js proce

* use & working of 'process'

→ Since the 'process' package is built we do not need to install it separately

→ We can directly use it in our Node.js application.

* Important properties & methods:
=> 'process.argv'
=> 'Process.env'
=> 'Process.pid'

Name: Gavaliya Heli P.
Roll No: 71

Date
Page

⇒ process.platform
⇒ process.cwd()
⇒ process.exitCode:

* Methods
⇒ process.exit([code]);
⇒ process.on(event, callback)
⇒ process.stdout.write(data)
⇒ process.stderr.write(data)

* Example:

```
console.log('command-line arguments:',
  process.argv.slice(2));
console.log('Environment variable "NODE_ENV":',
process.env.NODE_ENV);
```

[3] pm2 (external package)

'pm2' is a powerful process manager that allows us to manage and deploy Node.js applications.

→ It provides features such as process monitoring, automatic restarts, load balancing, log management and more.

(1) Installation:
```
npm install -g pm2
```
(2) starting an application:
```
pm2 start app.js
```

Name: Savaliya Heli P.
ROll NO: 71

Date
Page

* Important Methods:
→ pm2.start (scriptOrConfig, [options], [callback]);
→ pm2.list ([callback]);
→ pm2.stop (Process Id or Name, [callba

Example:

```
const pm2 = require('pm2');
const scriptPath = 'app.js';

pm2.start(scriptPath, (err, apps) => {
    if (err) throw err;
  console.log ('Application started succe
    -ssfully');

  pm2.stop(appName, (err) => {
    if (err) throw err;
    console.log ('Application stoped');

  pm2.disconnect ();
  });
});
```

* [4] readline

→ The readline Package works by crea an interface to read input and write output.

→ It uses an instance of the 'Interfac class to handle the interaction.

Name: Savaliya Heli P.
Roll No: 71

* Important properties:
→ readline.createInterface(options)
To create an instance of the 'Interface' class

* Important methods:
→ rl.question(query, callback):
→ rl.close()

* Example:

```
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

* use & working

It works synchronously and asynchronously. The synchronous methods block the execution of the code until the operation is completed.

Name: Savaliya Heli P.
Roll No: 71

Date
Page

Important Methods:

⇒ Reading files
* fs.readFile (path[, options], callback
* fs.readFileSync (path[, options]);

⇒ writing files.
* fs.writeFile (file, data[, options], c
* fs.writeFilesync (file, data[, optio

[6] events

It is not require to explicitly, as it is a core module and available by default in Node.js.

* Important methods!

⇒ on (eventName, listener)
⇒ once (eventName, listener)
⇒ removeListener (eventName, listener);
⇒ removeAllListeners ([eventName]);
⇒ emit (eventName [, ...args]);

* Example:

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter{
const myEmitter = new MyEmitter ();
myEmitter.on ('greet', (name) => {
    console.log ('Hello ${name}!');
});
```

Date.
Page.

[7] Console

The 'console' Package is available by default in Node.js and you do not need to require it explicitly.

Important Methods:

```
console.log([data][, ...args]);
console.info([data][, ...args])
console.error([data][, ...args])
console.warn([data][, ...args]);
console.time(label)
console.timeEnd(label)
```

Example

```
console.log('Hello world');
console.log('welcome to Node.js');
```

[8] Buffer

The buffer module provides a global Buffer class that can be used to create, manipulate, and convert binary data.

Important methods:

```
buffer.length
buffer.toString([encoding[, start[, end]]]);
buffer.slice([start[, end]])
```

buffer.concat(list[, totalLength]);
buffer.compare(targetBuffer)
buffer.copy(targetBuffer[, targetStart[, sourceStart[, sourceEnd]]])

☆ Example:

```
const buffer1 = Buffer.from('Hello, world!', 'utf8');

console.log('Buffer length:', buffer1.length);
```

[9] querystring

The 'querystring' module is commonly used to handle URL query parameters in web applications.

To use the 'querystring' module, we need to require it in our Node.js script.

☆ Important Methods:

querystring.parse(str[, sep[, eq[, options]]])
- str : string to be parsed
- sep : separator for spliting key-value pairs
- eq : equal for spliting key & value
- options: addition parsing options.

Date
Page

querystring.stringify([obj[, sep[, eq[, options]]]])

★Example:

```
const query = queryString.parse('name=abc
    &age=30');
console.log(query);

const params = {name: 'John', age:30};
const queryStr = queryString.stringify(params);
console.log(queryStr);
```

[10] http

To create an HTTP server, you need to require the http module in our Node.js script

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

★ Important methods:

⇒ http.createServer([options][, requestListener]);
⇒ http.server
⇒ http.IncomingMessage
⇒ http.ServerResponse

★ Example:

```
const http = require('http');
const operations = {
  hostname: 'localhost'
  path: '/endpoint';
  method: 'GET',
};
```

```
const req = http.request(options, (res) => {
  let data = '';

res.on('data', (chunk) => {
    data += chunk;
});

req.end();
```

[11] V8

To use the 'v8' module, we need to require it in our Node.js script

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

* Important methods:
⇒ v8.serialize(value)
⇒ v8.deserialize(buffer)
⇒ v8.getHeapSpaceStatistics()
⇒ v8.setFlagsFromString(flags)

* Example:

```
const v8 = require('v8');
const data = {name: 'John', age: 30};
const serializedData = v8.serialize(data);
console.log('Serialized data:', serializedData)
```

Date.
Page.

[12] OS

To use OS module, we need to require it in our Node.js script.

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

☆ Important methods:

⇒ os.hostname()
⇒ os.type()
⇒ os.platform()
⇒ os.arch()
⇒ os.totalmem()
⇒ os.cpus()
⇒ os.networkInterfaces()

☆ Example:

```
const os = require('os');
console.log('Hostname:', os.hostname());
console.log('Operating System:', os.type());
```

[13] zlib

To use the zlib module, we need to require it in our ~~.txt~~ Node.js script.

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

☆ Important methods:

⇒ zlib.deflate(buffer, options, callback)

=> zlib.deflateSync(buffer, [options])
=> zlib.gzip(buffer, [options], callback)
=> zlib.gzipSync(buffer, [options]);
=> zlib.brotliCompress(buffer, [options], callback)

★ Example:

```
const zlib = require('zlib');
const originalData = 'Hello friends';
const bufferData = Buffer.from(originalData);
zlib.deflate(bufferData, (err, compressedData) =>
  {
  if(!err) {
  console.log('compressed data:',
      compressedData.toString('base64'));
  }
});
```