Kalapi
Date
Page

Theory Assignment : 1

Name : Bhatt Srushti Daxeshkumar

Roll No : 006

Sem : 7th Sem (BSC-IT)

Subject : Application Development using Full Stack (701)

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

A:1 Introduction :-

→ Node.js is an open-source, cross-platform Javascript runtime environment that allows developers to execute Javascript code outside of a we browser.
→ It was created by Ryan Dahl and was first released in 2009.
→ Node.js is built on Chrome's V8 Javascript engine, which provides high-performance and efficient execution of Javascript code.
→ Traditionally, Javascript was mainly used for client-side scripting within web browsers, but Node.js extends its capabilities to server-side programming as well.
→ This means that developers can now use Javascript to build scalable and high-performance applications that can handle a large number of concurrent connections, making it well-suited for real-time applications like chat applications, streaming services, online gaming and more.

* Features :-

1. Asynchronous and Non-blocking I/O :

- Nodejs uses an event-driven, non-blocking I/O model, which allows it to effeciently handle multiple concurrent connections without waiting

for previous operations to complete. This asynchronous nature makes it ideal for handling I/O-heavy tasks, such as file system operations and network requests, resulting in improved performance and scalability.

2. Single-threaded Event Loop:

- Node.js operates on a single-threaded event loop, which means that it can handle numerous concurrent connections using a single thread. The event loop efficiently manages callbacks and events, avoiding the overhead of creating and managing multiple threads, making it lightweight and resource-efficient.

3. V8 JavaScript Engine:

- Node.js is built on the V8 JavaScript engine, developed by Google for its chrome browser. V8 compiles JavaScript code to native machine code, making Node.js highly performant and allowing it to execute JavaScript at impressive speeds.

4. npm (Node Package Manager):

- Node.js comes with a powerful package manager called npm, which is one of the largest package ecosystem in the world. npm allows developers to easily install,

manage, and share reusable packages and libraries, streaming the development process and fostering collaboration within the community.

5. Cross-platform Compatibility:

- Node.js is designed to be cross-platform, allowing it to run on various operating system, including Windows, macOS and Linux. This ensures that developers can write code once and deploy it across multiple platforms without any major modifications.

6. Server-side Web Applications:

- Node.js is commonly used to build server-side web applications. Developers can create web servers using built-in modules like 'http' or utilize frameworks like Express.js, enabling them to handle HTTP requests and build RESTful APIs efficiently.

7. Real-Time Applications:

- Node.js is well-suited for building real-time applications such as chat applications, online gaming platforms, collaborative tools, and live streaming services. Its asynchronous and event-driven nature allows it to handle simultaneous connections and deliver real-time updates to clients effectivity.

8. Scalability :- Due to its non-blocking I/O and event-driven architecture, Node.js can handle a large number of concurrent connections and scale effectively. This makes it an excellent choice for applications that need to handle a high volume of traffic and users.

9. Large community and Support :

- Node.js has a large and active community of developers, contributing to extensive documentation, libraries and modules. The vibrant community ensures continuous improvement, support and regular updates to the platform.

* Execution Architecture :-

→ The execution architecture of Node.js is based on an event-driven, non-blocking I/O model.
→ It revolves around a single-threaded event loop that efficiently handles asynchronous operations, making it highly performant and well-suited for handling concurrent connection
→ Components of Node.js execution architecture:

1. Event-Loop :
- The event loop is at the core of Node.js execution architecture.

- It is responsible for handling I/O operation is initiated, such as reading a file or making a network request, it is added to the event loop's queque, and Node.js continues executing the next piece of code without waiting for the operation to complete.

2. Event Queue :
- The event queue holds all the events and callbacks generated during the execution of Node.js code.
- When an asynchronous operation is completed or a timer expires, its corresponding callback is pushed into the event queue.

3. Callbacks and Event Handlers :
- Callbacks are functions that are registered to be executed when a specific event or asynchronous operation completes.
- In Node.js, callbacks play a significant role in handling the results of asynchronous operations
- When an event or operation is processed from the event queue, the associated callback is invoked, allowing the application to respond to the completed task.

4. Timers :
- Node.js includes timers that allow developers to schedule the execution of specific code at a later time.
- These timers can be one-time delays or periodic intervals, allowing for various scheduling tasks

- Timers are essential for handling tasks that need to occur at specific intervals or after a certain period.

5. Worker Threads :
- While Node.js primarily operates on a single-threaded event loop, it provides an option to use Worker Threads for CPU-intensive tasks.
- Worker Threads allow developers to create separate threads to execute computationally expensive operations, leveraging multiple CPU cores while still benefiting from Node.js's non-blocking I/O model.

6. Native Bindings :
- Node.js provides a C++ API that allows developers to create native bindings, enabling them to access low-level system resources or integrate with existing C/C++ libraries.
- These native bindings are often used in situations where higher performance is crucial or when interfacing with hardware devices.

Q:2 Note on modules with example.

A:2 There are major three types of modules:

1) Core Module / Built-in Module.
2) Local Module / User-defined Module.
3) Third-Party Module / External Module.

1) Core Module / Built-in Module :-

→ Core modules are built-in modules that come pre-installed with Node.js.
→ They provide essential functionalities and are part of the Node.js runtime.
→ We can directly use core modules in our code without the need for additional installations or third-party libraries.
→ Core modules are always available in Node.js and can be accessed using their name without specifying a file path.
→ Some example of core modules include 'fs', 'http', 'path', 'url', 'util' etc.

Ex:

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

fs.readFile('file1.txt', 'utf-8', (err, data) => {
        if (err) {
            throw err;
        else {
            console.log(data);
        }
});
```

2) User-defined Module / Local Module :-

→ These are modules created by developers to encapsulate reusable code and functionalities specific to their application.
→ User-defined modules are organized in separate files and can be shared and reused across different parts of the applications.

→ To use a user-defined module, we need to use the CommonJS module system and export the functionality using 'module.exports'.
→ Then, we can import and use it in other files using the 'require' function.

Ex: user-defined module
- rectangle.js

```
module.exports.calculate (width, height)
                = function ().
    {   return width * height;
    }
```

- Now, we needs to include it in main page.

main.js

```
const calculate = require ('./rectangle.js');

const Area = calculate (5, 10);

console.log ('The area of rectangle is :' area);
```

3) Third-party Module / External Module :-

→ Third-party modules are external modules created by the Node.js community or other developers and are not part of the Node.js

core or the standard library.

→ These modules are typically hosted on the npm (Node Package Manager) registry, which is one of the largest package ecosystem in the world.

→ We can leverage third-party modules to extend the functionality of their applications, save development time, and reuse code developed and maintained by other.

→ These modules cover a wide range of use cases, including databases integrations, web frameworks, utility libraries, authentication solutions and much more.

→ To use external module, we need to install it via npm using the 'npm install' command, and then we can include it in our application by using the 'require' function just like local modules.

Ex: npm install node-static

- file1.js

```
const http = require('http');
const static = require('node-static');
const fileServer = new static.Server('./files');

var server = http.createServer((req, res) => {
    req.addListener('end', () => {
        fileServer.serve(req, res);
    }).resume();
})
```

```
server.listen(8000, () => {
    console.log("Listening on 8000 port");
});
```

Q:3 Note on packages with example.

Ans:3 In Node.js, packages are collections of modules that are typically distributed together to provide specific functionalities or features.

→ These packages are published on the npm registry, which is a vast ecosystem of open-source modules maintained by developers from all around the world.

→ The npm registry allows developers to share their code, making it available for others to use in their Node.js applications.

→ Packages available in Node.js are:
- Express
- Loadsh
- Async
- Request
- Nodemailer
- React

Ex: npm install express

```
const express = require('express');
const app = express();

app.use(express.json());
```

```
app.get('/', (req,res)=>{
    res.send('Hello');
});

app.post('/api/message',(req,res)=>{
    const message = req.body.message;
    res.send('Message Received :',message);
});

app.listen(3000,()=>{
    console.log("Listening on 3000 port");
});
```

* Benefits of Using Packages:

- Reusability:
  Packages provides pre-built functionalities that can be easily reused across different projects, saving developers time and effort.

- Modularity:
  Packages encourage modular development by allowing developers to break-down their applications into smaller, manageable pieces.

- Collaboration:
  By publishing packages on npm, developers can collabrate with others, contribute to open-source projects and benefit from community-driven solutions.

- Consistency:
  Packages often follow best practies and coding standards, leading to more consistent and reliable code.

Q:4 Use of package.json and package-lock.json in Node.js packages.

Ans:4 The use of 'package.json' and 'package-lock.json' files in Node.js projects, especially when working with npm, is crucial for managing dependencies and ensuring consistency across different environments.

1) package.json:

→ The 'package.json' file is a JSON file that serves as the configuration and metadata files for our Node.js project.
→ It includes important information such as the project's name, version, description, entry point, author, licence and more.
→ Most importantly, it contains a list of dependencies required for our project to function correctly.
→ Developers manually create the 'package.json' file or generate it using the 'npm init' command.
→ The dependencies are specified with their names and version ranges.

2) package-lock.json :

→ The 'package-lock.json' file is automatically generated by npm when we install or update packages in our project.
→ It serves as a lock file and contains the exact version of each installed dependency, including their sub-dependencies.
→ The 'package-lock.json' file ensures that the same version of dependencies are installed across different environments or when multiple developers work on the same project.
→ It prevents dependencies in dependency versions that can lead to issues.
→ This file should be committed to version control systems to ensure that everyone working on the project gets the same set of dependency versions.

Q:5 npm introduction and commands with its use.

Ans:5 Node.js has npm which is stands for Node Package Manager.
→ npm is a package manager for node.js applications and libraries.
→ It is one of the largest package ecosystems in the world, providing a vast collection of open-source packages that can be easily installed, updated, and managed in Node.js projects.
→ npm simplifies the process of sharing, reusing

and distributing code, making it an essential tool for Node.js developers.

* npm commands and their uses:-

- 'npm init' : Initializes a new npm project and creates a 'package.json' file. It guides you through setting up basic project details.

- 'npm install': Installs all the dependencies listed in the 'package.json' file. It reads the 'package.json' and installs all the required packages in the 'node_modules' directory.
  Ex: npm install express

- 'npm install <package-name>': Installs a specific package and saves it as a dependency in the 'package.json' file.
  Ex: npm install express

- 'npm install <package-names --save-dev': Installs a specific package and save it as a dependency in the 'package.json' file.
  Ex: npm install nodemon --save-dev

~ We can write as node i <package-name> also.

- 'npm uninstall <package-name>': Uninstalls a package and removes it from the 'node_modules' directory and 'package.json'.
  Ex: npm uninstall express.

- 'npm update': Updates all the packages listed in 'package.json' to their latest versions.

- 'npm outdated': Shows a list of installed package that are outdated.

- 'npm list': Displays a tree of installed packages and their dependencies.

- 'npm run <script>': Runs a script defined in the 'scripts' section of 'package.json'.
  Ex: "scripts": { "start": "node server.js" }
  ↝ Use 'npm run start' to execute script.

- 'npm search <package-name>': Searches the npm registry for packages matching the given name.
  Ex: npm search express.

- 'npm info <package-name>': Searches the npm registry
  Displays detailed information about a specific package.
  Ex: 'npm info express'

- 'npm publish' : Publishes our package to the npm registry, making it publicly available.

- 'npm version' : Updates the version number in the 'package.json' file and create a new Git tag.

→ There are some common command and its use.

Q:6 Describe use and working of following Node.js packages. import properties and methods and relevant programs.

Ans:6 1. url :-

→ In Node.js, the built-in 'url' module provides utilities for working with URLs, including parsing, formatting and resolving URLs.
→ It is particularly useful for handling HTTP requests, parsing request URLs, and manipulating URLs in Node.js applications.

* Working :-

→ 1) Parsing URLs ('url.parse(urlString [, parseQuery String [, slashesDenoteHost]])');
- The 'url.parse()' method takes a URL string as input and returns an object containing its components such as protocol, host, port

pathname, search (query string), has, etc.
- 'urlString': The URL string to be parsed.
- 'parseQueryString': Optional. If 'true', the query property will be parsed using the 'queryString' module. Default is 'false'.
- 'slashesDenoteHost': Optional. If 'true', two slashes ('//') after the protocol will be treated as the host. Default is 'false'.

2) Formatting URLs ('url.format(urlObject)':
- The 'url.format()' method takes a URL object and returns the formatting URL string.

3) Resolving Relative URLs ('url.resolve(from, to)')
- The 'url.resolve()' method resolves a relative URL against a base URL.

* Use :-

1) Parsing incoming HTTP request URLs to extract components like the path, query parameters or hash.
2) Building dynamic URLs with query parameters based on user input.
3) ~~Maintaining URLs with query~~
3) Manipulating URLs in web scraping or crawling applications.
4) Handling URL redirections or resolving relative URLs in web servers.
5) Validating and sanitizing URLs to prevent

security vulnerabilities.

6) Working with APIs that accept or return URLs as parameters or responses.

* Properties :-

- url.href
- url.protocol
- url.host
- url.hostname
- url.port
- url.pathname
- url.search
- url.hash

* Methods :-

- url.parse()
- url.format()
- url.resolve()

* Example :-

```
const url = require('url');
var string = "http://localhost:8080/index.html
             ?year=2023;

var q = url.parse(string, true);

console.log(q.host);
console.log(q.pathname);
console.log(q.search);
```

2. Process or PM2 :-

* Working :

→) The 'process' module in Node.js provides

information and control over the current Node.js process.

→ It is global object, meaning it can be accessed from any module without requiring all explicit 'require()' statement.

→ The 'process' module is used to interact with the Node.js application's environment, handle signals, access command-line arguments, perform graceful termination and more.

* Properties:

- process.argv
- process.cwd
- process.version
- process.env
- process.pid
- process.versions
- process.platform

* Methods:

- process.exit()
- process.cwd()
- process.memoryUsage()
- process.on()
- process.chdir()
- process.uptime()

* Use:

1) Command-Line Arguments Handling.
2) Environment Variable Management.
3) Graceful Termination and Cleanup.
4) Process Event Handling.
5) Working with the current working directory.
6) Memory Usage information.
7) Uptime information.
8) Platform Information.

9) Version Information.

* Example:

```
console.log ('Command-line arguments :',
                process.argv);
```

3. readline :-

* The 'readline' module in Node.js provides an Interface for reading input from readable streams typically from the command-line or other input source.

* Working:

- The 'readline' module provides the 'createInterface()' method to create a new instance of the 'Interface' class.
- The 'Interface' class allows you to interact with input streams, such as 'process.stdin' and 'process.stdout'.

* Use :

1) Reading user input.
2) Handling user input.
3)

* Properties:

- readline.Interface

* Methods:

- question()
- ~~query~~ setPrompt()
- prompt()
- close()
- pause()
- resume()
- write()

* 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();
});
```

4. fs :-

→ 'fs' module in Node.js provides a set of methods for working with the file system.
→ It allows us to interact with files and directories on our computer or server.
→ The 'fs' module is essential for performing file-related operations, such as reading files, writing to files, creating directories, deleting files and more.
→ It is the one of the core modules in Node.js, so

we don't need to install any additional packages to use it.

* Working :

- To use the 'fs' module, we need to import require() method.

* Use :

1) File operations.
2) Directory operation.
3) File system operation.
4) File Information and Directory Listing.

* Properties :

- fs.constants

* Methods :

- fs.readFile()
- fs.readFileSync()
- fs.writeFile()
- fs.writeFileSync()
- fs.appendFile()
- fs.appendFileSync()
- fs.mkdir()
- fs.mkdirSync()
- fs.rmdir()
- fs.rmdirSync()
- fs.rename()
- fs.renameSync()
- fs.unlink
- fs.unlinkSync()
- fs.stat()
- fs.statSync()
- fs.readdir()
- fs.readdirSync()

* Example:

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

//read file.
fs.readFile('file1.txt', 'utf-8', (err, data) =>
    { console.log(data);
    });
```

5. events :-

→ In Node.js, the 'events' module is a core module that provides an implementation of the Event Emitter pattern.
→ Events are fundamental parts of the Node.js asynchronous, non-blocking architecture.
→ They allow objects to raise custom events, and other objects can register to listen for those events and react accordingly.

* Working :

1) Event Emitter creation
2) Event Registration.
3) Event Emission.

* Use :

1) Custom Event Handling.
2) Asynchronous Communication
3) Error Handling

4) Stream and File Handling.

* Properties :

- EventEmitter.defaultMaxListeners

* Methods :

- eventEmitter.on()
  eventEmitter.addListener()
  eventEmitter.off()
  eventEmitter.removeListener()
  eventEmitter.once()
  eventEmitter.emit()
- eventEmitter.eventNames()
- eventEmitter.listenerCount()
- eventEmitter.listener()
- eventEmitter.getMaxListeners()
- eventEmitter.setMaxListeners()

* Example :

```
const ee = require('events');
const myEE = new EventEmitter();

myEE.on ('greet', (name) => {
        console.log ('Hello, ${name}');
    });
myEE.emit ('greet', 'Soushti');
```

6. Console :-

→ In Node.js 'console' is a built-in module that provides a set of methods to interact with the standard output (stdout) and standard error (stderr) streams.

→ It allows you to log information, debug your

code, and display our message on the console during the execution of our application.

* Working:

- It provides various methods that accept different types of data as input and display that data on the console.
- The output is typically shown in Terminal where we are running our application.

* Methods:

- console.log()
- console.error()
- console.warn()
- console.clear()
- console.memory()
- console.info()
- console.debug()
- console.assert()
- console.count()
- console.dir()

* Example:

```
var name = 'Srushti';
var age = 22;

Console.log("name:', name, 'Age:', age);
```

7. buffer :-

→ In Node.js, the 'buffer' class is a built in module that provides a way to work with binary data directly.

→ Buffers are used to handle raw binary data, such as reading and writing binary files, dealing with ~~bi~~ network protocols, cryptographic operations and other situation where data needs to be handled in its raw form.

→ A buffer is essentially an array of bytes that represented binary data.

* Working:

1) Creating Buffers.
2) Writing to and Reading from the Buffers.
3) Converting Buffers to other formates.

* Use:

1) File I/O operations:
2) Network operations.
3) Cryptography.
4) Data parsing and Transformation.

* Properties:

- Buffer.byteLength

* Methods:

- Buffer.from()
- Buffer.alloc()
- Buffer.allocUnsafe()
- Buffer.allocUnsafeSlow()
- buf.length()
- buf.toString()
- buf.toJSON()
- buf.slice()

- but.copy()
- Buffer.compare()
- Buffer.concat().

* Example :

```
const dataString = "Hello, World";
const buffer = Buffer.from(dataString,
                           'utf-8');
console.log(buffer);

const string = buffer.toString('utf-8');
console.log(string);
```

8. queryString :-

→ In Node.js, the 'queryString' module is a built-in module that provides utilities for parsing and formatting URL query strings.
→ It allows us to work with the query string data typically found in the 'query' part of a URL.
→ The 'queryString' module provides methods to convert query string data between objects and URL-encoded strings.

* Working :

1) Parsing Query String.
2) Formatting QueryString

* Use :

1) Handling HTTP requests.
2) Building HTTP requests.
3) Working with APIs.

* Properties :

- querystring.escape - querystring.unescape

* Methods :

- querystring.parse() - querystring.stringify()

* Example :

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

const query = 'name=Srushtif age=22';
const q = qs.parse(query);

console.log(q);
```

9. http :-

→ In Node.js, 'http' is a built-in module that provides functionality for creating and working with HTTP servers and clients.
→ It allows you to handle HTTP requests and responses, create web servers, send HTTP requests to external servers and interact with web APIs.

* Working:

1) HTTP server creation.
2) Handle request from the client.

* Use:

1) Creating a web server.
2) Building a web applications.
3) HTTP client request.

* Properties:

- http.GET
- http.STATUS_CODE[ ]

* Methods:

- http.createServer()
- http.request()
- http.get()

* Example:

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

server.listen(8000,() => {
    console.log('listening on 8000);
});
```

10. V8 :-

→ In Node.js, the 'V8' module is a built in module that provides access to the V8 JavaScript engine's functionality.
→ The V8 engine is the JavaScript engine developed by Google, which is used to execute JavaScript code in Node.js and the google chrome browser.

* Working :

1) The 'V8' module provides the methods and properties that allow us to interact with the V8 engine and access low-level information about memory usage and performance.

* Use :

1) Memory Management.
2) Profiling and Optimization.

* Properties :

- v8.cachedDataVersionTag
- v8.getHeapSpaceStatistics
- v8.getHeapCodeStatistics

* Example :

```
const v8 = require('v8');
const heapState = v8.getHeapStatistics();
```

console.log('Heap statistics:', heapStats);

11. OS :-

→ In Node.js, the 'os' is a built-in module that provides various utility functions to interact with the operating system.
→ It allows us to access informations about the os, such as the platform, architecture, network interfaces and more.
→ The 'os' module abstracts os-specific functionality providing a consistent interface to work with various operating system.

* Use :

1) Platform Information.
2) Network Interfaces.
3) Operating System Information.
4) Memory and CPU Information.
5) Endianness.

* Properties :

- os.constants

* Methods :

- os.arch()
- os.cpus()
- os.freemem()
- os.totalmem()
- os.hostname()
- os.networkInterfaces()
- os.platform()
- os.release()
- os.type()

* Example :

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

console.log('Platform:', os.platform());
console.log('Architecture:', os.arch());
```

12. zlib :-

→ In Node.js, module is a built-in module that provides compression and decompression functionalities using the zlib library.
→ The zlib library is a widely used compression library that supports various compression algorithms like gzip, deflate and zlib.

* Working :

- The 'zlib' module in Node.js allows us to compress and decompress data using different compression algorithms.
- It provides methods to create compressed streams and decompressed streams to handle data efficiently.

* Use :

1) Compression
2) Decompression.

* Properties:

- zlib.Z_NO_FLUSH
- zlib.zFINISH
- zlib.z_BEST_SPEED

* Methods:

- zlib.deflate()
- zlib.deflateSync()
- zlib.deflateRaw()
- zlib.deflateRawSync()
- zlib.gzip()
- zlib.gzipSync()
- zlib.gunzip()
- zlib.gunzipSync()
- zlib.inflate()
- zlib.inflateSync()
- zlib.inflateRaw()
- zlib.inflateRawSync()

* Example:

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

const data = 'Hello, world';

const compress = zlib.deflateSync(data);
const decompress = zlib.inflateSync(compress);

console.log('Data original :', data);
console.log('Compressed Data :', compress);
console.log('Decompressed Data :', decompressed);
```