Client Server Architecture

- Client-server architecture in Node.js involves building applications where one component acts as a server, providing services or resources, while another component acts as a client, consuming those services or resources.
- Node.js is particularly well-suited for building server-side components due to its non-blocking, event-driven architecture, making it efficient for handling concurrent connections.

Example : Implementing a simple client-server architecture in Node.js

server.js

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

// Create an HTTP server
const server = http.createServer((req, res) => {
  // Set response HTTP header with HTTP status and Content type
  res.writeHead(200, {'Content-Type': 'text/plain'});

  // Send the response body
  res.end('Hello Students, Hope you are doing well\n');
});

// Listen on port 3000
server.listen(3000, () => {
  console.log('Server is now running at http://localhost:3000/');
});
```

In the above code example, we create an HTTP server using the http module. The server listens for incoming HTTP requests. When a request is received, it sends back a simple "Hello World" response. This server listens on port 3000.

client.js

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

// Make a request to the server
const options = {
    hostname: 'localhost',
    port: 3000,
    path: '/',
    method: 'GET'
};

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

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

    // End of data reception
    res.on('end', () => {
        console.log('Response from server:', data);
    });
});

// Handle request errors
req.on('error', (error) => {
    console.error('Error making request:', error);
});

// End the request
req.end();
```

In the above code example, we use the http module to make an HTTP request to the server running on localhost at port 3000. We specify the path '/' to indicate the root route of the server. When the response is received from the server, we log the response data to the console.

Single Threaded Model

In Node.js, traditionally, JavaScript execution follows a single-threaded model. This means that all JavaScript code runs on a single thread, using non-blocking I/O operations to handle concurrency. This model is made possible by the event loop, which allows Node.js to perform asynchronous operations efficiently.

How the single-threaded model works in Node.js:

- **Event Loop:** Node.js relies on an event loop to handle asynchronous operations. The event loop continuously checks the call stack and the task queue. When the call stack is empty, it takes the first task from the task queue and pushes it onto the call stack to execute it.
- **Non-Blocking I/O:** Node.js uses non-blocking I/O operations to perform tasks such as reading files, making network requests, and accessing databases.
  When Node.js initiates an I/O operation, it does not wait for the operation to complete. Instead, it continues executing the rest of the program. Once the I/O operation is finished, a callback function is called to handle the result.
- **Callback Functions:** Callback functions are a key feature of Node.js. They allow you to define what should happen after an asynchronous operation completes. For example, when reading a file, you can specify a callback function to be executed once the file has been read.

Example:

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

// Asynchronous file read operation
fs.readFile('data.txt', 'utf8', function(err, data) {
  if (err) {
    console.error('Error reading file:', err);
    return;
  }
  console.log('File contents:', data);
});

console.log('File read operation initiated.');
```

Node JS : Core Modules

- Working with core modules in Node.js involves utilizing the built-in modules provided by Node.js without needing to install any additional packages.
- These modules provide various functionalities to help you build applications efficiently.

Here are some of the commonly used core modules in Node.js

- **fs (File System):** This module provides functions to work with the file system, allowing you to create, read, update, and delete files and directories.

  Example:

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

// Example: Reading a file
fs.readFile('data.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

- **http:** This module provides functionality to create HTTP servers and clients. You can create a web server or make HTTP requests using this module.

  Example:

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

// Example: Creating a basic HTTP server
const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
});

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

- **path:** This module provides utilities for working with file and directory paths. It helps in manipulating file paths in a platform-independent manner.
  Example:
  ```
  // Import the path module
  const path = require('path');
  // Joining 2 path-segments
  path1 = path.join("students/training/projects", "index.html");
  console.log(path1)
  // Joining 3 path-segments
  path2 = path.join("students", "section/mern", "index.html");
  console.log(path2)
  // Joining with zero-length paths
  path3 = path.join("students", "", "", "index.html");
  console.log(path3)
  ```
  or
  ```
  // Import the path module
  const path = require('path');
  //getting the directory path
  console.log(path.dirname("E:\myapp\public\index.html"))
  //getting the file extension
  console.log(path.extname("E:\myapp\public\index.html"))
  //getting the file name
  console.log(path.basename("E:\myapp\public\index.html"))
  //getting the complete details
  console.log(path.parse("E:\myapp\public\index.html"))
  ```
- **os:** This module provides operating system-related utility methods and properties, such as retrieving information about the operating system.
  Example:
  ```
  const os = require('os');

  // Example: Getting system information
  console.log('Platform:', os.platform());
  console.log('Architecture:', os.arch());
  console.log('Free memory:', os.freemem());
  console.log('Total memory:', os.totalmem());
  ```

- **util:** The util core package in Node.js provides utility functions that are helpful for various tasks, including debugging, formatting, and working with objects and functions. It offers a collection of commonly used utility functions that are not specific to any particular module or domain.

  Here are some key features of the util core package along with examples:

  Formatting: The format method is used to format strings with placeholders.
  Example:
  ```
  const util = require('util');

  const formattedString = util.format('%s %s', 'Hello', 'students');
  console.log(formattedString);
  ```

  Promisify: The promisify method is used to convert callback-based functions into Promise-based functions.
  Example:
  ```
  const util = require('util');
  const fs = require('fs');

  const readFileAsync = util.promisify(fs.readFile);

  readFileAsync('example.txt', 'utf8')
    .then(data => {
      console.log(data);
    })
    .catch(error => {
      console.error(error);
    });
  ```

  Inspecting Objects: The inspect method is used to inspect an object, providing a string representation of the object for debugging purposes.
  Example:
  ```
  const util = require('util');

  const obj = { name: 'neha mittal', age: 30 };
  console.log(util.inspect(obj));
  ```

- **crypto:** The crypto core package in Node.js provides cryptographic functionality that allows developers to perform cryptographic operations such as creating hashes, generating random bytes, encrypting and decrypting data, and working with digital signatures. It includes various cryptographic algorithms and methods for secure data handling.

  Hashing: The crypto module provides methods for creating hash digests of data using various hashing algorithms like SHA-256, SHA-512, MD5, etc.
  Example:
  ```
  const crypto = require('crypto');

  const data = 'Hello, Students!';
  const hash = crypto.createHash('sha256').update(data).digest('hex');

  console.log('SHA-256 Hash:', hash);
  ```

  Encryption and Decryption: The crypto module also supports encryption and decryption using symmetric and asymmetric encryption algorithms.
  Example:
  ```
  const crypto = require('crypto');

  // Generate a random key and IV
  const key = crypto.randomBytes(32); // 256-bit key
  const iv = crypto.randomBytes(16); // 128-bit IV

  const algorithm = 'aes-256-cbc';
  const cipher = crypto.createCipheriv(algorithm, key, iv);

  let encryptedData = cipher.update('Data for processing', 'utf8', 'hex');
  encryptedData += cipher.final('hex');
  console.log('Encrypted data:', encryptedData);

  const decipher = crypto.createDecipheriv(algorithm, key, iv);
  let decryptedData = decipher.update(encryptedData, 'hex', 'utf8');
  decryptedData += decipher.final('utf8');
  console.log('Decrypted data:', decryptedData);
  ```

Generating Random Bytes: The crypto module can be used to generate cryptographically strong pseudo-random data.
Example:

```
const crypto = require('crypto');
const randomBytes = crypto.randomBytes(16); // Generate 16 bytes of random data
console.log('Random Bytes:', randomBytes.toString('hex'));
```

- **url:** This module provides utilities for URL resolution and parsing.
  Example:

```
const url = require('url');
// Example: Parsing a URL
const parsedUrl =
url.parse('https://www.youtube.com/watch?v=BrocgPH2GYI', true);
console.log(parsedUrl);
```

- **dns:** This module provides DNS (Domain Name System) related functionality, such as resolving domain names to IP addresses.
  Example:

```
const dns = require('dns');
// Example: Resolving a domain name
dns.resolve4('www.facebook.com', (err, addresses) => {
  if (err) throw err;
  console.log('IP addresses:', addresses);
});
```

- **stream:** This module provides the Stream API, allowing you to work with streams of data, such as reading from or writing to files, HTTP requests, or TCP sockets.
  Example:

```
const fs = require('fs');
const stream = require('stream');
// Example: Creating a readable stream from a file
const readableStream = fs.createReadStream('example.txt', 'utf8');
// Example: Creating a writable stream to a file
const writableStream = fs.createWriteStream('output.txt', 'utf8');
```

Node JS : User Defined Modules

Working with user-defined modules in Node.js involves creating your own modules and then using them in other parts of your application. This helps in organizing your code into reusable components and promotes modularity.

create and use user-defined modules

- **Creating a User-Defined Module:** Create a new JavaScript file for your module. For example, let's create a module named calculate.js:

```
const add = (a, b) => {
  return a + b;
 };
 const sub = (a, b) => {
  return a - b;
 };
 module.exports = {
  add,
  sub
 };
```

  In the module, we define two functions, add and sub, and export them using module.exports.

- **Using the User-Defined Module:** Now, you can use the module you created (calculate.js) in another file in your application.
  Example : Let's create a file named app.js

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

console.log(math.add(15, 12));
console.log(math.sub(167, 32));
```

  In the file, we import the calculate.js module using require('./calculate.js') and then use the exported functions (add and sub) as properties of the imported math object.

- **Using User-Defined Modules in Different Folders:** If your user-defined module is in a different folder, you need to specify the relative path when requiring it.

  For example, if the calculate.js module is inside a folder named check:

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

  console.log(math.add(15, 12));
  console.log(math.sub(167, 32));
  ```

- **Exporting Multiple Values from a Module:** You can export multiple values (functions, variables, objects, etc.) from a module using module.exports.

  For example:

  ```
  // module.js
  const sayHello = () => {
    console.log('Hello Students!');
  };
  const sayWelcome = () => {
    console.log('Welcome to Learn2Earn Labs – Best Training Institute!');
  };
  module.exports = {
    sayHello,
    sayWelcome
  };
  ```

  then, you can use these functions in another file just like before:

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

  modules.sayHello();
  modules.sayWelcome();
  ```