**WEB TECHNOLOGIES**

**NodeJS & MongoDB**

**Prof. Spurthi N Anjan**

Department of Computer Science and Engineering **spurthianjan@pes.edu**

**Backend Development **

Almost every application is incomplete without a functioning server and database to process data. Now that we can easily create front ends using React, let us understand why leading companies like Netflix, Uber, and LinkedIn use Node.js as their the

backend development platform

**PESU Department of CSE**

**2**

**What is NodeJS? **

Node.js is an open-source server side runtime environment

built on Chrome’s V8 JavaScript engine.

Simply put, **Node.js** is a platform that executes server-side 

JavaScript programs that can communicate with I/O sources

like networks and file systems.

It is essentially JavaScript outside of a browser.

It’ is designed to build scalable network applications

**PESU Department of CSE**

**3**

**Why NodeJS? **

When Ryan Dahl created Node in 2009 he argued that I/O was being handled incorrectly, blocking the entire process due to synchronous programming.

Traditional web-serving techniques use the thread model, meaning one thread for each request. Since in an I/O operation the request spends most of the time waiting for it to complete, intensive I/O scenarios entail a large amount of unused resources (such as memory) linked to these threads. Therefore the “one thread per request” model for a server doesn’t scale well.

Instead of the thread model, he said the right way to handle several concurrent connections was to have a single-thread, an event loop and non-blocking I/Os.

**PESU Department of CSE**

**4**

**The Essence of NodeJS**

**V8 Engine Event Loop**

**PESU Department of CSE**

****

**Asynchronous**

**Input and Output**

**Event Driven**

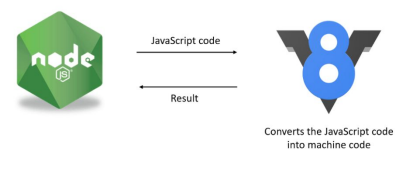
**Architecture**

**5**

**V8 Engine **

Node.js is built on the V8 engine of Google. Node.js cannot understand the javascript code we write without V8.

It is the fastest javascript engine.

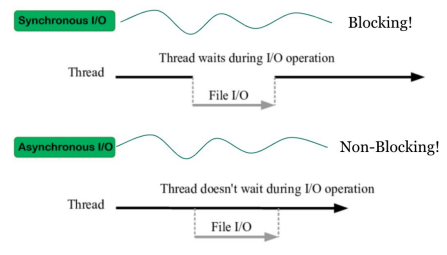
The V8 engine converts the javascript code into the machine code which the computer actually understands. The result is then generated and returned to node.js. 

**PESU Department of CSE**

**6**

**Asynchronous I/O **

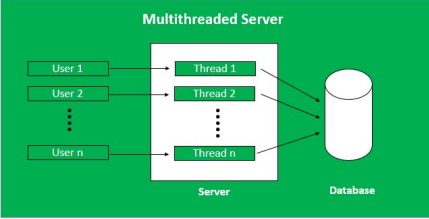
The Asynchronous I/O allows applications to overlap processing with I/O operations. In simple words, the goal is that the program should never block

**PESU Department of CSE** 

**7**

**Event Loop **

Generally, the server-side technologies like PHP, ASP.NET, Ruby & Java Servers all follow a multi-threaded model. In this traditional architectural approach, each client request creates a new thread or a process.



**PESU Department of CSE**

**8**

**Event Loop *Node js in contrast works differently:***

When we use Node.js on a computer, it means that there is a node process running on that computer. The process is just a program in execution.

Now in that process, Node.js runs in a single thread.

A thread is basically just a sequence of instructions.

Therefore, if we have 4 different tasks then all these four tasks will happen in one single thread.

**PESU Department of CSE**

**9**

**Event Loop **The event loop is called the heart of the node.js.

It executes all the callback functions( functions that are called as soon as some work is finished) in a single thread and it also offloads heavy or expensive tasks like compressing a file to a thread pool.

Eventloop makes asynchronous programming possible in node.js.

So, this single thread doesn't have to wait for the request to complete and is free to handle the next request. When asynchronous I/O work completes then it processes the request further and sends the response.

**PESU Department of CSE**

**10**

**Event Loop **

**PESU Department of CSE**

**11**

**Event-Driven Architecture **

In Node, there are certain objects called event emitters that emit named events as soon as something important happens in the app, like a request hitting server or a file finishing to read.

These events are then picked up by event listeners that we developers set up, which will fire off functions(callback functions) that are attached to each listener.

The event-driven architecture makes it way more straight forward to react multiple times to the same event.

**PESU Department of CSE**

**12**

**Pros and Cons **

Think about why

async

programming can

be

disadvantageous

**PESU Department of CSE**

**13**

**Applications of NodeJS**

**PESU Department of CSE**

****

**It is not**

**advisable to**

**use Node.js**

**for CPU**

**intensive**

**applications!**

**14**

**NodeJS in Use** 

****

**PESU Department of CSE**

**15**

**Setup **

Installation of NodeJS is straightforward using the installer package available at NodeJS official website. Have an IDE ready; Visual Studio is highly recommended.

● Download the installer from NodeJS WebSite: **https://nodejs.org/en/**

● Run the installer.

● Follow the installer steps, agree the license agreement and click the next button.

● Restart your system/machine.

Now, test NodeJS by printing its version using the following command in Command Prompt: **node -v**

Test npm by printing its version using command

**npm -v**

Have an IDE ready; Visual Studio is highly recommended.

**PESU Department of CSE**

**16**

**Welcome to NodeJS **Add the following to a file say “welcome.js”:

**console.log("Welcome to NodeJS!");**

Run the following line on command prompt:

Output:

**PESU Department of CSE**

**17**

**Node Modules **Node modules provide a way to re-use code in your Node application.

In Node. js, Modules are **the blocks of encapsulated code** that communicates with an external application on the basis of their related functionality. Modules can be a single file or a collection of multiples files/folders.

These modules have a unique context, thus, they never interfere nor pollute the scope of other modules. Node.js includes three types of modules:

➢ Core Module

➢ Local Module

➢ Third Party Module

**PESU Department of CSE**

**18**

**Core Modules **

Core Modules are the ones that are provided by node.js itself and loads automatically when Node.js process starts. However, it is required to import these modules using the syntax:

**var module = require('module\_name');**

****

**PESU Department of CSE**

**19**

**Some Core modules**

**const path = require('path'); //Use the core library file.**

**console.log("Directory of index file: " + path.basename(\_\_filename)); console.log("Extension of index file: " + path.extname(\_\_filename));**

**var util = require('util');**

**var txt = 'Congratulate %s on his %dth birthday!';**

**var result = util.format(txt, 'Linus', 6);**

**console.log(result);**

**Output:**

**Output:**

**PESU Department of CSE**

**20**

**Local Module **

The local modules of Node.js are custom modules that are created locally by user/developer in the application.

*Local\_module.js app.js*

**var detail = {**

**name: function (name) { console.log('Name: ' + name); },**

**domain:function (domain) { console.log('Domain: ' + domain);**

**}**

**};**

**module.exports = detail;**

*exported*

**var myLogModule =**

**require('./Local\_module.js'); myLogModule.name('School');**

**myLogModule.domain('Education');**

**exports is a special object to expose a module to another application. So, whatever you assign to**

**module.exports will be exposed as a module.**

**PESU Department of CSE**

**21**

**More on Local Modules ***myfirstmodule.js app.js*

**obj={**

**firstname: "Jane",**

**lastname: "Doe"**

**};**

**module.exports.obj=obj;**

**//module chaining is also possible module.exports.upper=function(name){ return name.toUpperCase(); }**

**module.exports.mydatetime=function(){ return Date();**

**};**

**Output:**

**var custom\_module =**

**require("./myfirstmodule.js");**

**console.log(custom\_module.mydatetime()); console.log(custom\_module.obj); console.log(custom\_module.upper("jane doe"));**

**You can export simple literals ,strings or objects!**

****

**PESU Department of CSE**

**22**

**Third Party Modules **Third party modules can be downloaded from NPM registry that can help make coding better. These modules are generally developed by other developers and are free to use.

Some of the best known packages include:

● Express

● Rectify

● Lodash

● Mongoose, etc

Third party modules can be install inside the project folder or globally, using npm.

**PESU Department of CSE**

**23**

**Third Party Modules: NPM **

**NPM** stands for Node Package Manager which as the name suggests is a package manager for Node.js packages/modules. From Node version 0.6.0. onwards, npm has been added as default in the node installation.

It saves you from the hassle of installing npm explicitly.

*Globally Loading the 3rd party module:*

**npm install --g <module\_name>**

*Include your module file in your main application file:*

**npm install --save <module\_name>**

**PESU Department of CSE**

**24**

**File System Module **

The file system module, or simply fs, allows you to access and interact with the file system on your machine.

Since it is a core module, all you have to do is import it!

Using the fs module, you can perform actions such as:

➢ Reading files

➢ Creating files

➢ Updating files

➢ Deleting files

➢ Renaming files

**PESU Department of CSE**

**25**

**File System Module: Async vs Sync **

It's important to note that by default, all the fs methods are asynchronous. However, you can use the synchronous version by adding Sync at the end of the method.

For instance, a method such as writeFile becomes writeFileSync.

Synchronous methods complete the code synchronously, and thus they block the main thread. Blocking the main thread in Node.js is considered bad practice.

Asynchronous methods take the last parameter as the completion function callback and the first parameter of the callback function as error.

**PESU Department of CSE**

**26**

**Read from a File**

To read the physical file asynchronously use:



**const fs = require('fs')**

**fs.readFile(fileName[,options], callback) Shorthand!**

Where:

● filename: Full path and name of the file as a string.

● options: The options parameter can be an object or string which can include encoding and flag. The default encoding is utf8 and default flag is "r".

● callback: A function with two parameters err and fd. This will get called when

readFile operation completes.

**fs.readFile('./test.txt', 'utf8' , (err, data) => {**

**if (err) {**

**console.error(err)**

**return**

**}**

**console.log(data)**

**})**

**PESU Department of CSE**

**27**

**Write to a File**

To write to a physical file asynchronously use:

**fs.writeFile(filename,data[,options], callback)**

Where:

● filename: Full path and name of the file as a string.

● data: The content to be written in a file. ● options: Object or string which can include encoding, mode and flag. The default

encoding is utf8 and default flag is "r".

● callback: A function with two parameters err and fd. This will get called when write operation completes.



**const fs = require('fs')**

**const content = 'Some content!'**

**fs.writeFile('./test.txt', content, err => {**

**if (err) {**

**console.error(err)**

**return**

**}**

**//file written successfully**

**})**

**PESU Department of CSE**

**28**

**Opening a File **Alternatively, you can open a file for reading or writing using fs.open() method.

**fs.open(path, flags[, mode], callback)**

Where:

● path: Full path with name of the file as a string.

● flag: The flag to perform operation

● mode: The mode for read, write or readwrite. Defaults to 0666 readwrite.

● callback: A function with two parameters err and fd. This will get called when file open operation completes.

**PESU Department of CSE**

**29**

**Flags PESU Department of CSE**

**30**

**Renaming and Updating Files **

To asynchronously rename a file at the given old path to a given new path use:

**fs.rename( oldPath, newPath, callback )**

It will overwrite the destination file if it already exists.

To asynchronously append the specified content

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

**fs.rename('test.txt', 'world.txt', () => {**

**console.log("File Renamed!");**

**});**

at the end of the specified file use:**var fs = require('fs');**

**fs.appendFile( path, data[, options], callback )**

**fs.appendFile('mynewfile1.txt', ' This is my text.', function (err) { if (err) throw err;**

**console.log('Updated!');**

**});**

**PESU Department of CSE**

**31**

**File System Module: Deleting and Closing Files **

To asynchronously delete an existing file use: **fs.unlink(path, callback);**

It will overwrite the destination file if it already exists.

To close a file descriptor use:

**fs.close(fd, callback);**

Where:

● fd − This is the file descriptor returned by file fs.open() method.

● callback − This is the callback function. No arguments other than a possible exception are given to the completion callback.

**var fs = require('fs');**

**fs.unlink('test.txt', function () { console.log('operation complete.'); });**

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

**fd = fs.openSync("test.txt");**

**fs.close(fd, (err) => {**

**if (err)**

**console.error('Failed to close file', err);**

**else {**

**console.log(" File Closed**

**successfully");}**

**});**

**PESU Department of CSE**

**32**

**Some Important fs Methods **

**PESU Department of CSE**

**33**

**Buffer Module **

In Node.js, buffers are a special type of object that can store raw binary data. A buffer represents a chunk of memory - typically RAM - allocated in your computer. Once set, the size of a buffer cannot be changed.

A buffer stores bytes. A byte is a sequence of eight bits. Bits are the most basic unit of storage on your computer, they can hold the value of either 0 or 1.

Node.js exposes the Buffer class in the global scope (you don't need to import or require it like other modules). With this API, you get a series of functions and abstractions to manipulate raw binaries.

Node.js displays bytes using

the hexadecimal system!!

.

**<Buffer 61 2e 71 3b 65 2e 31 2f 61 2e>**

**PESU Department of CSE**

**34**

**Creating a Buffer: Buffer.alloc() **

The .alloc() method is useful when you want to create empty buffers, without necessarily filling them with data. By default, it accepts a number and returns a buffer of that given size filled with zeroes:

**Buffer.alloc(*size, [fill*, *encoding]*);**

**Buffer.alloc(6);**

**// --> <Buffer 00 00 00 00 00 00>**

**You can later on fill the buffer**

**with any data you want:**

**You can also fill the buffer with**

**other content than 0 and a given**

**encoding:**

**// Creates a buffer of size 1 filled with 0s (<Buffer 00>)**

**const buff = Buffer.alloc(1);**

**// Fill the first (and only) position with content**

**buff[0] = 0x78 // 0x78 is the letter "x" console.log(buff.toString('utf-8');**

**Buffer.alloc(6, "x", "utf-8"); // --> 'x'**

**// --> <Buffer 78 78 78 78 78 78>**

**PESU Department of CSE**

**35**

**Creating a Buffer: Buffer.allocUnsafe() **

With .allocUnsafe() , the process of sanitizing and filling the buffer with zeroes is skipped. The buffer will be allocated in a area of memory that may contain old data (that's where the "unsafe" part comes from).

**Buffer.allocUnsafe(*size*);**

**// Allocates a random area of memory with size 10000**

**// Does not sanitizes it (fill with 0) so it may contain**

**old data**

**const buff = Buffer.allocUnsafe(10000);**

**// Prints loads of random data**

**console.log(buff.toString("utf-8"));**

**PESU Department of CSE**

**36**

**Reading a Buffer **The toString() method returns the buffer object according to the specified encoding.

***buffer*.toString([*encoding, start*, *end]*);**

**var buffer = new Buffer.alloc(5);**

**for (var i = 0; i < 5; i++) {**

**buffer[i] = i + 97;**

**}**

**console.log(buffer.toString());**

**console.log(buffer.toString('utf-8', 1, 4));**

**console.log(buffer.toString('hex'));**

**PESU Department of CSE**

**37**

**Writing to a Buffer **

By default, buffer.write() will write a string encoded in utf-8 with no offset (starts writing from the first position of the buffer). It returns a number, which is the number of bytes that were written in the buffer:

***buffer*.write(*value*, *start*, *bytes*, *encoding*);**

**const buff = Buffer.alloc(9);**

**buff.write("hey there"); // returns 9 (number of bytes**

**written)**

**// If you write more bytes than the buffer supports,**

**// your data will truncated to fit the buffer.**

**buff.write("hey christopher"); // returns 9 (number of bytes**

**written)**

**console.log(buff.toString());**

**// -> 'hey chris'**

**PESU Department of CSE**

**38**

**Some Buffer Methods PESU Department of CSE**

**39**

**Streams **

Streams are a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way.

Instead of a program reading a file into memory all at once like in the traditional way, streams read chunks of data piece by piece, processing its content without keeping it all in memory.

Let’s take a “streaming” services such as YouTube or Netflix for example: these services don’t make you download the video and audio feed all at once. Instead, your browser receives the video as a continuous flow of chunks, allowing the recipients to start watching and/or listening almost immediately

**PESU Department of CSE**

**40**

**Why Streams? **Streams basically provide two major advantages compared to other data handling methods:

➢ **Memory efficiency:** you don’t need to load large amounts of data in memory before you are able to process it

➢ **Time efficiency:** it takes significantly less time to start processing data as soon as you have it, rather than having to wait with processing until the entire payload has been transmitted

**PESU Department of CSE**

**41**

**Streams in NodeJS **

➢ **Writable:** streams to which we can write data. For example, fs.createWriteStream() lets us write data to a file using streams.

➢ **Readable:** streams from which data can be read. For example: fs.createReadStream() lets us read the contents of a file.

➢ **Duplex:** streams that are both Readable and Writable. For example, net.Socket

➢ **Transform:** streams that can modify or transform the data as it is written and read. For example, in the instance of file-compression, you can write compressed data and read

de-compressed data to and from a file. Node.js comes with a variety of transform streams in the Core API:

● zlib - for gzip compressing and decompressing

● crypto - for encrypting, decrypting, and calculating message digests

**PESU Department of CSE**

**42**

**Streams in NodeJS **

**PESU Department of CSE**

**43**

**Events in Streams **The streams throw several events since they are eventEmitter instances. These events are used to track and monitor the stream.

Some of the most commonly used events are:

➢ Data - Data event is emitted when readable data is available.

➢ Finish - Finish event is emitted when the stream is done writing data.

➢ Error - Error event is emitted when an error occurs while reading/writing data. ➢ End - End event is emitted when the read stream has finished reading data.

**PESU Department of CSE**

**44**

**Reading from a Stream **

**const fileSystem = require("fs");**

**var data = "";**

**const readStream = fileSystem.createReadStream("input.txt");**

**readStream.on("data", (chunk) => {**

**data += chunk;**

**});**

**readStream.on("end", () => {**

**console.log(data);**

**});**

**readStream.on("error", (error) => {**

**console.log(error.stack);**

**});**

**PESU Department of CSE**

**45**

**Writing to a Stream **

**const fileSystem = require("fs");**

**var data = "Sample text";**

**const writeStream = fileSystem.createWriteStream("output.txt");**

**writeStream.write(data, "UTF8");**

**writeStream.end()**

**writeStream.on("finish", () => {**

**console.log("Finished writing");**

**});**

**writeStream.on("error", (error) => {**

**console.log(error.stack);**

**});**

**PESU Department of CSE**

**46**

**Piping Streams **Piping is a mechanism that involves using the output of another stream input of the other.

**const fileSystem = require("fs");**

**const readStream = fileSystem.createReadStream("input.txt");**

**const writeStream = fileSystem.createWriteStream("output.txt");**

**readStream.pipe(writeStream);**

**console.log("Program finished");**

**PESU Department of CSE**

**47**

**Transforming a Stream ***Compression: Decompression:*

**var fs=require("fs");**

**var zlib= require('zlib');**

**fs.createReadStream('test.txt') //reads .pipe(zlib.createGzip()) //compresses .pipe(fs.createWriteStream("data.txt.gz")); //writes into the file**

**console.log("File is compressed"); //zip file is created**

**var zlib= require('zlib');**

**var fs=require("fs");**

**fs.createReadStream('data.txt.gz') //reads .pipe(zlib.createGunzip()) //decompresses .pipe(fs.createWriteStream("data.txt")); //writes**

**console.log("File is decompressed");**

**PESU Department of CSE**

**48**

**Callbacks **

Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task.

Node makes heavy use of callbacks. All the APIs of Node are written in such a way that they support callbacks.

For example, a function to read a file may start reading file and return the control to the execution environment immediately so that the next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as a parameter. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process a high number of requests without waiting for any function to return results.

**PESU Department of CSE**

**49**

**Blocking I/O**

**var fs = require("fs");**

**var data =**

**fs.readFileSync('data.txt');**

**console.log(data.toString()); console.log("Reading Complete");**

****

**Output:**

****

The program blocks until it reads the file and then only it proceeds to end the program.

A blocking program executes very much in sequence. From the programming point of view, it is easier to implement the logic for a blocking program.

**PESU Department of CSE**

**50**

**Non-Blocking I/O**

**var fs = require("fs");**

**fs.readFile('data.txt', function (err, data) {**

**if (err) return console.error(err); console.log(data.toString()); });**

**console.log("Reading Complete");**

****

**Output:**

****

The program does not wait for file reading and proceeds to print "Reading complete" and at the same time, the program continues reading the file.

Non-blocking programs do not execute in sequence

**PESU Department of CSE**

**51**

**HTTP Module **

Node.js has a built-in (core) module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

The HTTP module creates a HTTP server that listens to server ports. The server created can read HTTP requests made by a client through a browser or console.

Let’s look at how to build a simple server using the http module.

**Step 1:**

Create a new file called server.js and include the http module by using the require() function **const http = require('http');**

*There is also a HTTPS module for secure HTTP requests*

**PESU Department of CSE**

**52**

**HTTP Module**

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

**const server = http.createServer((req, res) => {**

**if (req.url === '/') {**

**res.write('<h1>HTTP Server says hi!</h1>');**

**}**

**res.end();**

**});**

**server.listen(5000);**

**console.log('The HTTP Server is running on port 5000');**

****

**Step 2:**

Create an HTTP server using the createServer() method of the http object.

The createServer() accepts a callback that has two parameters:

HTTP request (req) and response (res). Inside the callback, we send an HTML string to the browser if the URL is ‘/’ and end the request.

**Step 3:**

Listen to the incoming HTTP request on the port 5000

**Step 4:**

Verify that the server is up and running on localhost:3000

**PESU Department of CSE**

**53**

**Another Simple Server **

**var http = require('http'); // Import Node.js core module var server = http.createServer(function (req, res) { //create web server if (req.url == '/') { //check the URL of the current request**

**// set response header**

**res.writeHead(200, { 'Content-Type': 'text/html' });**

**Header as an object**

**status code**

**// set response content**

**res.write('<html><body><p>This is home Page.</p></body></html>');**

**res.end();**

**}**

**else if (req.url == "/student") {**

**res.write sends a chunk of the response body**

**res.writeHead(200, { 'Content-Type': 'text/html' });**

**res.write('<html><body><p>This is student Page.</p></body></html>'); res.end();**

**//to be continued**

**PESU Department of CSE**

**54**

**Another Simple Server **

**//continued**

**else if (req.url == "/admin") {**

**res.writeHead(200, { 'Content-Type': 'text/html' }); res.write('<html><body><p>This is admin Page.</p></body></html>'); res.end();**

**}**

**else**

**res.end('Invalid Request!');**

**signals to the server that all of the response headers and body have been sent**

**});**

**server.listen(8081);**

**Server listening at http://localhost:8081/**

**Observe what happens when you give http://localhost:8081/student and http://localhost:8081/admin !!**

**console.log('Node.js web server at port 8081 is running..');**

**PESU Department of CSE**

**55**

**Handling HTTP Requests **

So far we have made requests to the server through a browser by just changing the URL. This is however not very practical.

Usually an application initiates a HTTP request and the server is informed about the request.

The server then decodes the HTTP request and sends it to the corresponding application or server for further processing.

For the server we created, let’s look at how to send a request for ‘/admin’ from a client application.

**PESU Department of CSE**

**56**

**Handling HTTP Requests **

The common programming task of making a HTTP request to a web server can be easily performed using the default http module with the help of http.request

**http.request(options, callback)**

This method is used to issue a HTTP request where:

➢ Options is an object that specifies host, port, path and other header information. ○ host: the domain or IP address of the server

○ port: the port (e.g. 80 for HTTP)

○ path: the request path, including the query string (e.g. 'index.html?page=12') ➢ The callback passed to the method will receive a http.ClientResponse object when the request is made. The ClientResponse is a Readable Stream.

**PESU Department of CSE**

**57**

**Handling HTTP Requests: Client **

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

**const options = {**

**hostname: 'localhost',**

**port: 8081, path: '/admin', method: 'GET' };**

**the resource under request . In this case, it could be / or / admin or /student To make a POST requests , just change method to ‘POST”**

**var callback= function(response){ var body="";**

**response.on('data', function(data){ body+=data;**

**});**

**response.on('end', function(){**

**console.log(body);**

**});**

**response.on('error', (err) => {**

**console.error(err);});**

**};**

**const req = http.request(options, callback); req.end();**

**Binding events to their event handlers. Recall node events...**

**Note: Run your server first. Open up a new terminal and run your client**

**PESU Department of CSE**

**58**

**Working with Queries: Server **Real world applications will also include search strings. Let us try to handle them!

**var http = require('http'); var url = require('url');**

**The core URL module splits up a web address into readable parts. url.parse() will create an object to easily**

**http.createServer(function (req, res) { res.writeHead(200, {'Content-Type': 'text/html'}); var q = url.parse(req.url, true).query;**

**var txt = q.name + " " + q.srn;**

**res.write(txt);**

**access url parts as properties.** 

**Eg:**

**url.parse(req.url).host** → **returns hostname**

**res.end();**

**}).listen(8080);**

**Query parameters**

**url.parse(req.url).query** → **returns object containing query parameters like so {q: ‘val’}**

**PESU Department of CSE**

**59**

**Working with Queries: Client **

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

**const options = {**

**hostname: 'localhost',**

**port: 8080,**

**path: '/?name=ABC&srn=PES000', method: 'GET'** 

**};**

**var callback= function(response){ var body="";**

**response.on('data', function(data){ body+=data;**

**});**

**response.on('end', function(){ console.log(body);**

**});**

**response.on('error', (err) => { console.error(err);**

**});};**

**Query string wherein the parameter=value pairs are separated by ‘&’ . These parameters are used on the server side**

**const req = http.request(options, callback);**

**req.end();**

**PESU Department of CSE**

**60**

**The Data Store **

If your application stores any data (user profiles, content, comments, uploads, events, etc.), then you’re going to want to use a data store.

Not only do they allow you to store, search, filter and present information based on web requests from users, they also allow a wide variety of mathematical and statistical calculations on queries submitted from web browsers.

So, in MERN stack, this database tier is developed using :



**PESU Department of CSE**

**61**

**MongoDB **MongoDB is the database used in the MERN stack.

It is a NoSQL (non-relational) document-oriented database, meaning it’s essentially **not** a conventional database where you have tables with columns and rows and strict relationships among them

It makes use of a flexible schema and a JSON-based query language.

Not only do many modern companies (including Facebook and Google) use MongoDB in production, but some older established companies such as SAP and Royal Bank of Scotland have adopted MongoDB

**PESU Department of CSE**

**62**

**Components of MongoDB Architecture **

**PESU Department of CSE**

**63**

**MongoDB: Documents **MongoDB documents are composed of field-and-value pairs and have the following structure:

*Example*

**{** 

**field1: value1,**

**field2: value2,**

**...**

**fieldN: valueN**

**}**

**PESU Department of CSE**

**64**

**MongoDB: Collections **

MongoDB stores documents in collections.

Collections are analogous to tables in relational databases.

Collections do not enforce a schema, and documents in the same collection can have different

fields.



**PESU Department of CSE**

**65**

**MongoDB: Collections **A primary key is mandated in MongoDB, and it has the reserved field name \_id.

Even if \_id field is not supplied when creating a document, MongoDB creates this field and auto-generates a unique key for every document.



**PESU Department of CSE**

**66**

**MongoDB: Database **In MongoDB, databases hold one or more collections of documents

It is a logical grouping of many collections.

A database connection is restricted to accessing only one database, so to access multiple databases, multiple connections are required.

Thus, it is useful to keep all the collections of an application in one database, though a database server can host multiple databases.

**PESU Department of CSE**

**67**

**Drawing Parallels **

**PESU Department of CSE**

**68**

**Table vs Collection** 

Relational Database

**PESU Department of CSE**

MongoDB 

Documents in the

same collection can

have different fields

but all documents

in a collection must

have a unique \_id!

**69**

**MongoDB Installation **

Install mongodb by following the instructions for your respective OS:

https://docs.mongodb.com/manual/installation/

If you are using VS Code, install the MongoDB for VS Code extension. This makes working with MongoDB

so much easier!



The MongoDB syntax however remains the same even if you wish to work on your console. **PESU Department of CSE**

**70**

**MongoDB for VS Code**

****{databases **PESU Department of CSE**

****

**Connect to default**

**71**

**Creating Databases and Collections **

**Creates a new database if it doesn't exist, otherwise it will** 

**return the existing database.**

**Creates new collection**

**PESU Department of CSE**

**72**

**Creating Documents **

**You can see the** 

**documents once**

**inserted!**

**PESU Department of CSE**

**73**

**Reading from MongoDB**

**You can see that mongoDB** 

**assigns \_id for all**

**documents ALWAYS!**

*Result: *

**PESU Department of CSE**

**74**

**Reading from MongoDB**

**filter PESU Department of CSE**

*Result: ***75**

**Reading from MongoDB **

*Result:*

**

**PESU Department of CSE**

**76**

**Reading from MongoDB ***Result:*

**

**PESU Department of CSE**

**77**

**Reading from MongoDB **

*Result:*

**

**PESU Department of CSE**

**78**

**Reading from MongoDB **

**PESU Department of CSE**

**79**

**Updating in MongoDB**

**Finds all documents** 

**with id 1 and**

**updates age to 23**

**PESU Department of CSE**

****

*Result:*

**80**