UNIT IV

4 Back End Technologies

Syllabus

***Node.JS** : Introduction to Node.JS, Environment Setup, Node.JS Events, Node.JS Functions, Node.JS Built- in Modules, File System, NPM, Install External Modules, Handling Data I/O in Node.JS, Create HTTP Server, Create Socket Server, Microservices- PM2.*

***ExpressJS** : Introduction to ExpressJS, Configure Routes, Template Engines, ExpressJS as Middleware, Serving Static Files, REST HTTP Method APIs, Applying Basic HTTP Authentication, Implement Session Authentication.*

***MongoDB** : NoSQL and MongoDB Basics, MongoDB-Node.JS Communication, CRUD Operations using Node.JS, Mongoose ODM for Middleware, Advanced MongoDB.*

Contents

Part I : Node.JS

4.1 Introduction to Node.JS

- NodeJS is an open source technology for server.
- Using Node.js we can run JavaScript on server.
- It runs on various platform such as Windows, Linux, Unix, and MacOS.

Uses of Node.js

It can perform various tasks such as -

1) It can create, open, read,delete, write and close files on the server.
2) It can collect form data.
3) It can also add, delete, modify data in databases.
4) It generate dynamic web pages.

Features

Following are some remarkable features of node.js –

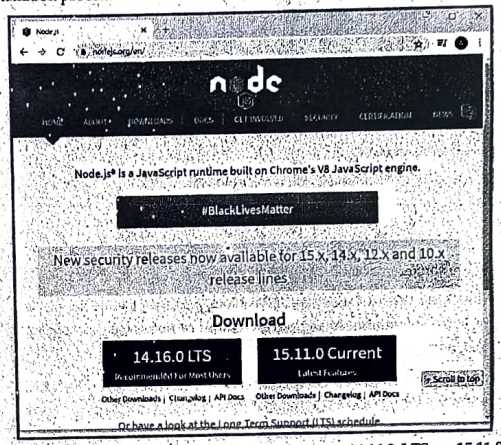
1) **Non blocking thread execution** : Non blocking means while we are waiting for a response for something during our execution of tasks, we can continue executing the next tasks in the stack.
2) **Efficient** : The node.js is built on V8 JavaScript engine and it is very fast in code execution.
3) **Open source packages** : The Node community is enormous and the number of permissive open source projects available to help in developing your application in much faster manner.
4) **No buffering** : The Node.js applications never buffer the data.
5) **Single threaded and highly scalable** : Node.js uses a single thread model with event looping. Similarly the non blocking response of Node.js makes it highly scalable to sever large number of requests.

Review Question

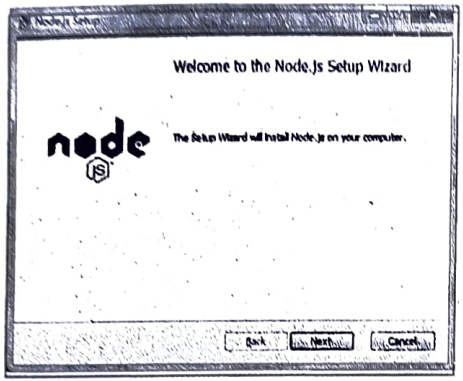
1. *What is Node.js? Enlist the features of Node.js*

4.2 Environment Setup

- For executing the Node.js scripts we need to install it. We can get it downloaded from https://nodejs.org/en.
- The installation can be done on Linux or Windows OS. It has very simple installation procedure.

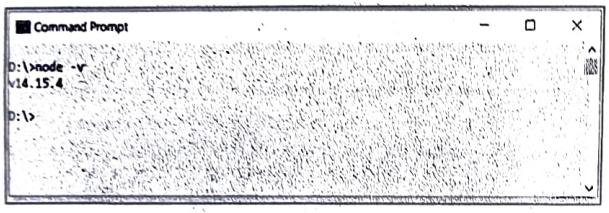


- Download node MSI for windows by clicking on 14.16.0 LTS or 15.11.0 Current button.
- After you download the MSI, double-click on it to start the installation as shown below.



- **Click Next button to read and accept the License Agreement and then click Install. It will install Node.js quickly on your computer. Finally, click finish to complete the installation.**
- **Verify Installation**

  In order to verify the installation we have to open a command prompt and type the following command –

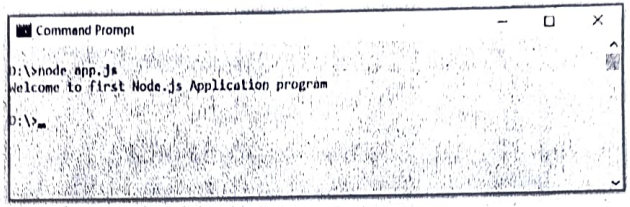


- **If the node.js is successfully installed, then some version number of the node.js which you have installed in your PC will be displayed.**
- **After successful installation we can now execute the Node.js document. The Node.js file has extension .js.**
- **Following is a simple node.js program which can be written in notepad.**

app.js

```
console.log("Welcome to first Node.js Application program");
```

Output



4.3 Handling Data I/O in Node.JS

The console object is a global object which is used to print different levels of messages to stdout and stderr.

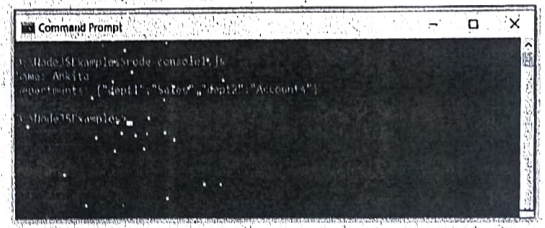
The most commonly used methods console object are console.log console.info, console.warn and console.error. Let us discuss them with illustrative examples

1) **console.log([data][, ...]) :** It is used for printing to stdout with newline. This function can take multiple arguments similar to a printf statement in C.

console1.js

```
var emp_name = 'Ankita';
        emp_depts = {
        dept1: 'Sales', dept2: 'Accounts'
        };
console.log('Name: %s \n\t Departments: %j', emp_name, emp_depts);
```

Output



2) **console.info([data][, ...]) :** This method is similar to console.log.

console2.js

```
var emp_name = 'Ankita';
  emp_depts = {
    dept1: 'Sales',
    dept2: 'Accounts'};
  console.info('Name: %s'+'\n'+'Departments: %j',emp_name,emp_depts);
```

Output

```
Name: Ankita
Departments: {"dept1":"Sales","dept2":"Accounts"}
```

3) **console.error ([data][, ...])** : This method prints to stderr with newline

console3.js

```
var code = 401;
console.error('Error!!!',code);
```

Output

```
Error!!! 401
```

4) **console.warn([data][, ...])** : This method is similar to console.error() method.

Consider following example – in which using the **fs** module we open a text file. If the file does not open successfully then console.warn() method displays the warning message.

console4.js

```
var fs = require('fs');
fs.open('input.dat','r',function(err,fs){
   if(err)
        console.warn(err);
   else
        console.log("File opening successfull!!");
});
```

Output

Review Question

1. *Explain any four methods of console object in node.js with suitable examples.*

4.4 Node.JS Events

4.4.1 Introduction to Events

- Event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads.

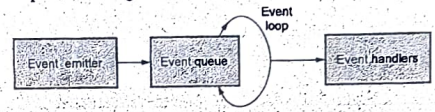


Fig. 4.4.1 Event driven architecture in node.js

What Is Event ?

- Every time when user interacts with the webpage, event occurs when user clicks mouse button or presses some key on the keyboard.
- When events are triggered, some functions associated with these events get executed.
- Event driven programming makes use of the following concepts –
  i) When some events occur, an event handler function is called. This is basically a call back function.
  ii) The main program listens every event getting triggered and calls the associated event handler for that event.

Concept of Event Emitter :

- The **Eventmitter** is a module that facilitates communication between objects.
- The emitter objects performs following tasks –
  1) It emits named events.
  2) Registers and unregisters listener functions.
- Basically **EventEmitter** is a class which can be used to raise and handle custom events.

Steps for using event handling in Node.js

Step 1 : import the 'events' module and create an instance of it using following code –

```
var events = require('events');
```

Step 2 : Then create an instance of **EventEmitter()** class. This instance is created because we have to call on() and emit() functions using this instance.

Step 3 : Then define a callback function which should be called on occurrence of event. This function is also called as event listener funciton.

Step 4 : Then register the event using **on()** function. To this function name of the event and callback function is passed as arguments.

Step 5 : Finally raise the event using **emit()** function.

Example Code

eventDemo1.js

```
// get the reference of EventEmitter class of events module
var events = require('events');

//create an object of EventEmitter class by using above reference
var em = new events.EventEmitter();

var myfunction = function() {
    console.log('When event occurs, My Function is called');
}

//Bind FirstEvent with myfunction
em.on('FirstEvent', myfunction);

// Raising FirstEvent
em.emit('FirstEvent');
```

Output

```
When event occurs, My Function is called
```

Program Explanation : In above program,

1) We have to import the 'events' module.
2) Then we create an object of EventEmitter class.
3) For specifying the event handler function, we need to use on() function.

4) The **on()** function requires two parameters - Name of the event to be handled and the callback function which is called when an event is raised.
5) The **emit()** function raises the specified event.

Example Code for Passing data along with event

We can pass some data inside the event, following example illustrate the same -

eventDemo1.js

```
//get the reference of EventEmitter class of events module
var events = require('events');
//create an object of EventEmitter class by using above reference
var em = new events.EventEmitter();
var myfunction = function(data1,data2) {
    console.log('When event occurs, My Function is called');
    console.log('Data passed is, '+data1+' '+data2);
}
//Bind FirstEvent with myfunction
em.on('FirstEvent', myfunction);
// Raising FirstEvent
em.emit('FirstEvent',data1,data2);
```

Output

```
D:\Node.JS\Examples\Node event>node eventDemo1.js
When event occurs, My Function is called
Data passed is, data1 data2
```

4.4.2 Concept of Event Loop

- Node Js is a single threaded system. So it executes the next task only after completion of previous task.
- The event loop allows Node.js to perform non-blocking I/O operations evenif the JavaScript is single threaded.
- Event loop is an endless loop which waits for tasks, executes these tasks and sleeps until it receives next task.
- The event loop allows us to use callback functions.
- The event loop executes task from event queue only when the call stack is empty. Empty call stack means there is no ongoing task.
- Event loop makes execution fast.

Event Loop Execution Model

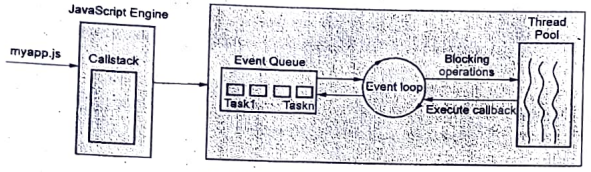


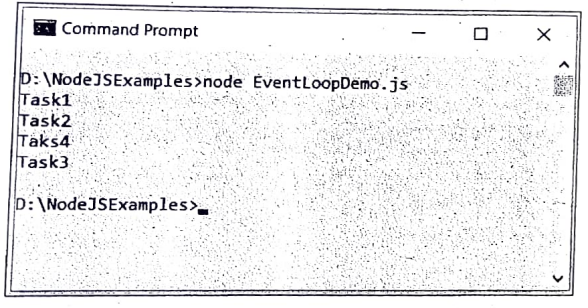
Fig. 4.4.2 Event loop execution

- The Event Loop is a main loop **running continuously** for executing the callback functions. When an async function executes, the callback function is pushed into the queue. The JavaScript engine doesn't start processing the event loop until the code after an async function has executed.

Example Code

```
console.log("Task1");
console.log("Task2");
setTimeout(function() {console.log("Task3") }, 1000);
console.log("Taks4");
```

Output



Phases of Event Loop

Various phases of event loop are –

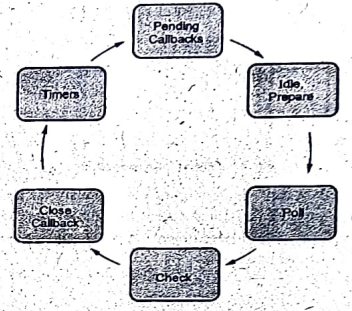


Fig. 4.4.3

1) **Timers :** This is the first phase in the event loop. It finds expired timers in every iteration (also known as Tick) and executes the timers callbacks created by setTimout and setInterval.
2) **Pending callbacks :** It handles I/O callbacks deferred to the next iteration, such as handling TCP socket connection error.
3) **Idle, prepare :** It is used internally.
4) **Poll :** The Poll phase calculates the blocking time in every iteration to handle I/O callbacks.
5) **Check :** This phase handles the callbacks scheduled by setImmediate(), and the callbacks will be executed once the Poll phase becomes idle.
6) **Close callback :** This phase handles callbacks if a socket or handle is closed suddenly and the 'close' event will be emitted.

4.5 Node.JS Callback Function

What is callbacks ?

Callbacks is a function which is usually passed as an argument to another function and it is usually invoked after some kind of event.

How it works ?

The callback function is passed as an argument to another function. When particular event occurs then only it is invoked.

By the time callback function is executing another task gets executed.

Hence execution of callbacks is asynchronous execution.

Example Code

Step 1 : Create a simple text file named "myfile.txt" as follows –

myfile.txt

```
How are you?
Jack and Jill went up the hill
to fetch a glass of water
```

Step 2 : Create a JavaScript file(.js) that contains the callback function

callbackDemo.js

```
var fs = require('fs');
console.log('Serving User1');

fs.readFile('myfile.txt', function (err, data) {
  if (err) return console.error(err);
  console.log("*** Contents of the File are : ***");
  console.log(data.toString());
});
console.log('Serving User2');
console.log('Serving User3');
console.log('Good Bye!!!');
```

Output

```
D:\Node_Examples>node callbackdemo.js
Serving User1
Serving User2
Serving User3
Good Bye!!!
*** Contents of the File are : ***
How are you?
Jack and Jill went up the hill
to fetch a glass of water
```

Program Explanation : In above program,

1) We have to read a file named **myfile.txt**, but while reading a file it should not block the execution of other statements, hence a call back function is called in which the file is read.

2) By the time file gets read, the remaining two console functions indicating "Serving user 1" and "Serving user 2" are not blocked, rather they are executed, once the contents are read in the buffer, then those contents of the file are displayed on the console.

Review Question

1. *Explain the callbacks in node js with suitable example.*

4.6 Node.JS Built- in Module

Module is nothing but the set of functions that can be used in our application. Node.js has various built in modules that can be used in our application without any installation.

To include the module, we use **require()** function with the name of module. Let us discuss some important built-in modules

(1) assert

This module is used during the testing of expressions. If an expression evaluates to 0 or false, an error is thrown and program gets terminated.

Step 1 : Open Notepad and create a .js file as follows. I have created application using the name app.js. Here is the code.

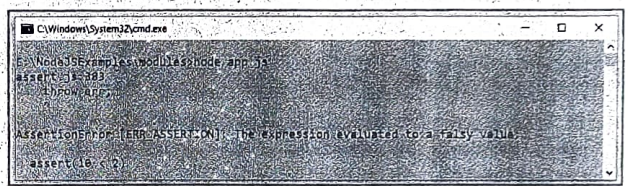
app.js

```
var assert = require('assert');
assert(10 < 2);
```

Step 2 : Now, open the command prompt and issue the command

```
node app.js
```

Please refer following screenshot



Note that as the 10 is not less than 2, the above code will raise the error named **AssertionError**

(2) cluster

The cluster module helps in creating a child process. Both the main and child processes can run simultaneously and share the same server port. Let us see its use with the help of demo example.

Step 1 : Open Notepad and create a .js file as follows. I have created application using the name app.js. Here is the code.

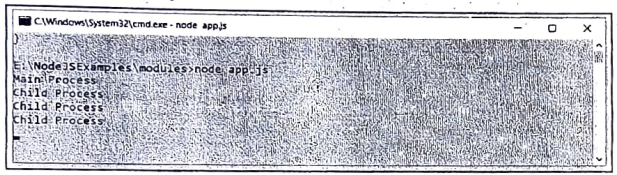
app.js

```
var cluster = require('cluster');
if (cluster.isWorker) {
 console.log('Child Process');
} else {
    console.log('Main Process');
    cluster.fork();
    cluster.fork();
    cluster.fork();
}
```

Step 2 : Now, open the command prompt and issue the command

```
node app.js
```

Please refer following screenshot



(3) os

The os module provides the information about operating system of your computer.

Step 1 : Open Notepad and create a .js file as follows. I have created application using the name app.js. Here is the code.

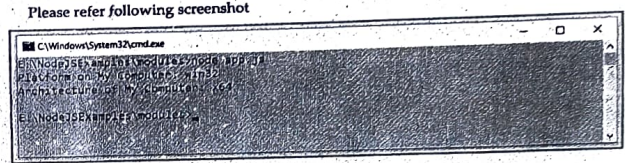
app.js

```
var os = require('os');
console.log("Platform on My Computer: " + os.platform());
console.log("Architecture of My Computer: " + os.arch());
```

Step 2 : Now, open the command prompt and issue the command

```
node app.js
```

Please refer following screenshot



(4) Timers

- Timers module in Node.js provides a way for scheduling the functions to be executed on some given time.
- For using the timers module in Node.js it is not required to import it via **require()** function as all the methods are globally available to JavaScript API. Hence we do not have to use **require** in the source code while use timer functions.
- There are **three** set timer functions used in Node.js

1. setTimeout() 2. setImmediate() 3. setInterval()

Let us understand them with suitable examples -

1) setTimeout() : This function is used to set one time callback after delaying by some milliseconds.

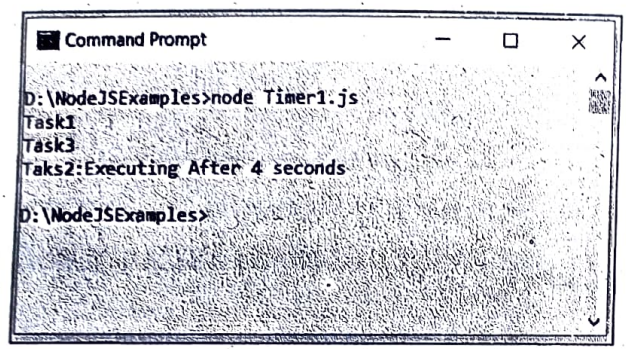
Syntax

```
setTimeout(function,delay_in_milliseconds);
```

Timer1.js

```
console.log('Task1');
setTimeout(function() {console.log('Task2 Executing After 4 seconds')},4000);
console.log('Task3');
```

Output



Example code for the function having some parameters

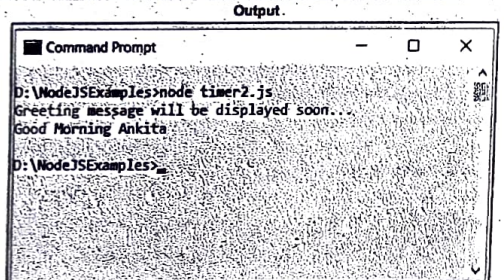
We will now discuss to use **setTimeout** function for accepting a function with parameters. The function that will be passed to setTimeout along with parameters has following syntax

```
setTimeout(function_name, delay_in_milliseconds,parameter1,parameter2 ... );
```

Timer2.js

```
console.log("Greeting message will be displayed soon...")
var myfun = (user) => {
    console.log("Good Morning "+user)
};
setTimeout(myfun,1000,'Ankita');
```

Output



2) setImmediate() : This function schedules the immediate execution of callback function after I/O event.

The syntax is

```
setImmediate(() => {
  //function body
})
```

We don't need to pass any time interval to this method. The first argument is the function to execute. You can also pass any extra arguments to this function .

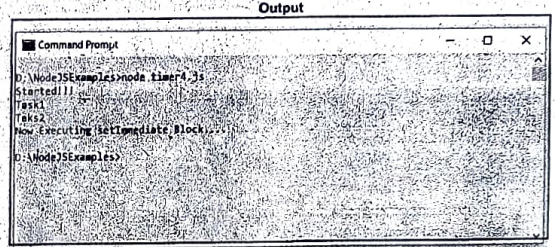
This method will execute a function right after the next code blocks which is present after the **setImmediate** function.

For example -

```
console.log("Started!!!")
setImmediate(() => {
    console.log("Now Executing setImmediate Block...")
});
console.log("Task1")
console.log("Taks2")
```

Note that these **console.log** statements will execute first and then immediately after that the **setImmediate** function will execute.

Output



3) setInterval() : This function helps to execute a task in a loop after some time interval. This function has the same syntax as that of **setTimeout.**

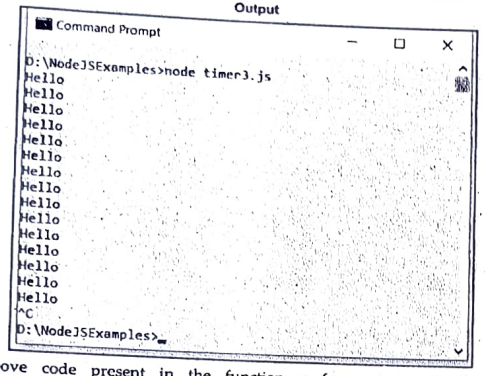
Syntax

```
timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

For example

```
var myfun = () => {
    console.log("Hello")
};
setInterval(myfun,1000)
```

Output



The above code present in the function **myfun** gets executed after every 1000 milliseconds. To break this program running in a loop we have to press control+c.

Review Question

1. *State and explain any three built in modules in node.js*

4.7 Buffers

- Buffer is class that allows us to store raw data similar to arrays.
- It is basically a class that provides the instances for storing the data.
- It is a global class so we can use it without any need of importing a buffer module.

Creation of Buffer

Creating a buffer is very simple. For example

```
var buffer = new Buffer(10);
```

This will create a buffer of 10 units.

Similarly we can create a buffer as follows –

```
var buffer = new Buffer([1,2,3,4,5], "utf8");
```

The second parameter which we have passed here is encoding scheme. It allows various encodings such as utf8(this is default one), ascii,ucs2,base64, or hex.

Writing to Buffer

The syntax for writing to the buffer is

```
write(string[,offset][,length][,encoding])
```

where

string : It is the string to be written to the buffer.

offset : It is the index from where we can start writing. The default value is 0.

length : It indicates the number of bytes to be written to the buffer.

Encoding : It indicates the desired encoding. The default one is "utf8".

Example Code

Following is a simple node js program that shows how to write data to the buffer.

BufferExample.js

```
var buffer = new Buffer(50);
buffer.write("Good Morning");
console.log(buffer.toString('utf8'));
```

Output

```
Good Morning
```

Program Explanation: In above program,

We have used **write** method for writing to the buffer. For displaying the string written to the buffer we have to use **toString** method along with the encoding scheme.

Reading from Buffer

The syntax for reading from buffer is as follows –

```
toString([encoding][,start][,end])
```

where

encoding : It indicates the encoding scheme. The default one is "utf8".

start : Beginning index is indicated by this value. The default is 0.

end : Ending index is indicated by this value.

Example Code

```
var buffer = new Buffer(50);
buffer.write("Good Morning");
console.log(buffer.toString('utf8'));
```

4.8 Streams

Stream can be defined as objects that reads data from source or writes the data to the destination in continuous fashion.

The stream is an EventEmitter instance that emits several events. Following are the events that occur during stream operations –

1) **data :** This event is fired when we want to read the data from the file.
2) **end :** When there is no more data to read then this event is fired.
3) **error :** If any error occurs during reading or writing data then this event is fired.
4) **finish :** When data is completely flushed out then this event occurs.

There four types of streams,

1) **Readable :** This stream is used for read operation only.
2) **Writable :** This stream is used for write operation only.
3) **Duplex :** This type of stream is used for both read and write operation.
4) **Transform :** In this type of stream the output is computed using input.

Write Operation

The steps that are followed to write data to the stream using node.js are as follows -

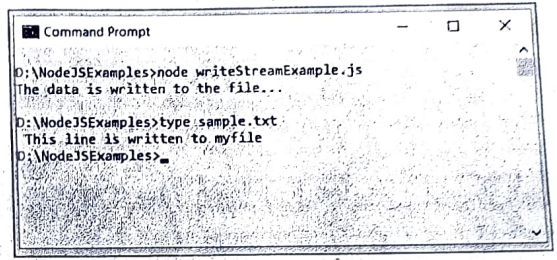
Step 1 : Import the file stream module fs using **require**.
Step 2 : Create a writable stream using the method **createWriteStream.**
Step 3 : Write data to the stream using **write** method.
Step 4 : Handle the events such as finish or error(if any)

Following Nodejs script shows how to write data to the string,

writeStreamExample.js

```
var fs = require("fs");
var str = " This line is written to myfile";
var ws = fs.createWriteStream('sample.txt');
ws.write(str,'utf8');
ws.end();
console.log("The data is written to the file... ");
```


Output



The above code is simply modified to handle the error event as follows -

```
var fs = require("fs");
var str = " This line is written to myfile";
var ws = fs.createWriteStream('sample.txt');
ws.write(str,'utf8');
ws.end();
console.log("The data is written to the file... ");

ws.on('error',function(err) {
      console.log(err.stack);
});
```

Read Operation

For reading from the stream we use createReadStream. Then using the **data** event the data can be read in chunks.

Example Code

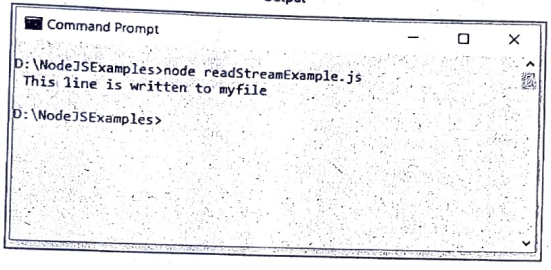
```
var fs = require("fs");
var str = '';
var rs = fs.createReadStream('sample.txt');
rs.setEncoding('utf8');

//handling stream event 'data'
rs.on('data',function(chunk){
    str += chunk;
```

```
});

//handling stream event 'end'
rs.on('end',function(){
    console.log(str)
});

//handling stream event 'error'
rs.on('error',function(err){
    console.log(err.stack);
});
```

Sample.txt

```
This line is written to myfile
```

Output



Pipe Operation

The pipe operation is a kind of operation in which output of one stream acts as an input of another stream.

There is no limit on pipe operation that means, all the output of one stream can be fed as input to input to another stream.

Following are the steps used for using the pipe operation in your node.js program –

Step 1 : Import the 'fs' module using **require.**

Step 2 : Create readable stream using **createReadStream** function.

Step 3 : Create writable stream using **createWriteStream** function.

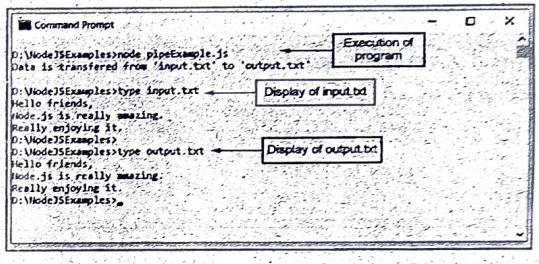
Step 4 : Use the pipe() function for passing the data from readable stream to writable stream.

Following example shows the use of pipe operation -

pipeExample.js

```
var fs = require("fs");
var rs = fs.createReadStream('input.txt');
rs.setEncoding('utf8');
var ws = fs.createWriteStream('output.txt');
ws.setEncoding('utf8');
rs.pipe(ws);
console.log("Data is transfered from 'input.txt' to 'output.txt'");
```

Output



4.9 File System

File system fs is implemented in the node.js using require function.

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

The common operations used with the file system module are -

1. Read file 2. Create file 3. Update file 4. Delete file

1. Read Operation

The read file operation can be performed **synchronously** or **asynchronously**.

For reading the file synchronously we use **readFileSync** method. And for reading the file synchronously **readFile**

Normally while reading the file using asynchronous form we use callback function as the last parameter to the **readFile** function. The form of this callback function is,

```
function(err, data)
{
  //function body
}
```

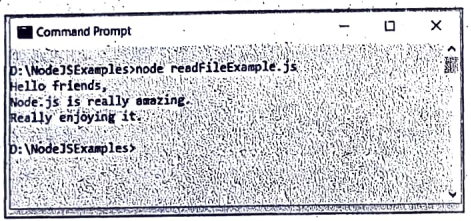
Example code

```
var fs = require("fs");
fs.readFile("input.txt",function(err,data){
   if(err)
      console.log(err);
   console.log(data.toString());
});
```

input.txt

```
Hello friends,
Node.js is really amazing.
Really enjoying it.
```

Output



Similarly we can read the file synchronously using following command -

readFileExample.js

```
var fs = require("fs");
var data = fs.readFileSync('input.txt');
console.log(data.toString());
```

2. Create File Operation

We can create an empty file using the command **open()**

Syntax

```
open(path,flags,mode,callback)
```

where

path : It is the path of the filename.

flag : It indicates, the behaviour on opening a file. That means "r" is open file for reading, "w" is for writing, "rs" is for synchronous mode, "r+" or "w+" means open a file for both read and write purpose and so on.

mode : It indicates readable or writable mode.

Callback : It is basically a function which has two arguments(err,data)

Example Code

```
var fs = require("fs");
fs.open('myfile.txt', 'w', function(err,file){
   if(err)
      console.log(err)
   console.log("File created!!!");
});
```

3. Update / Write File Operation

For writing to the file two operations are used most commonly -

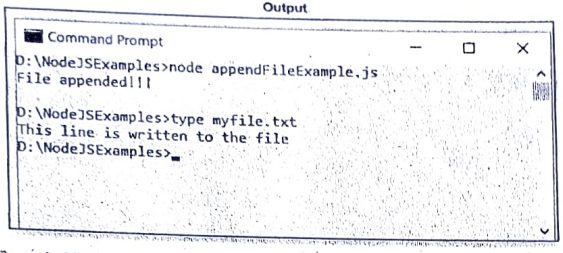
1. appendFile() and 2. writeFile()

The **appendFile()** method is used to write the contents at the end of the specified file.

Example Code

```
var fs = require("fs");
str = "This line is written to the file";
fs.appendFile('myfile.txt', str,function(err){
   if(err)
      console.log(err)
   console.log("File appended!!!");
});
```

Output

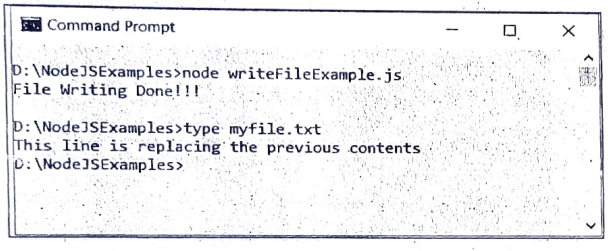


The **writeFile()** method is used to write the contents to the file

Example Code

```
var fs = require('fs');
str = "This line is replacing the previous contents";
fs.writeFile('myfile.txt', str,function(err){
    if(err)
        console.log(err)
    console.log('File Writing Done!!!');
});
```

Output



4. Delete File Operation

To delete the file, we use **unlink()** method.

Syntax

```
fs.unlink(path, callback)
```

where

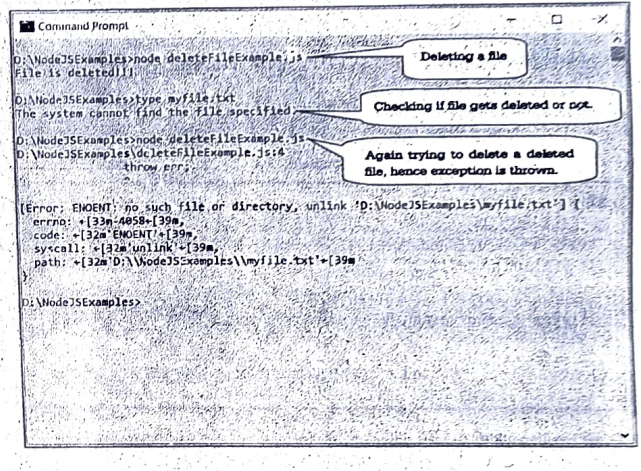
path : Is a path of the file to be deleted.

callback : It is a callback function with the only one argument for 'error'.

Example Code

```
var fs = require('fs');
fs.unlink('myfile.txt',function(err){
    if(err)
        throw err;
    console.log('File is deleted!!!');
});
```

Output

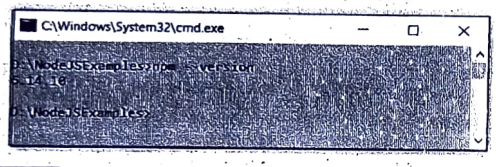


Review Question

1. *With necessary source code, explain how to use file I/O operations in node.js*

4.10 NPM

- A package in Node.js contains all the files you need for a module.
- Modules are JavaScript libraries you can include in your project.
- The NPM stands for node package manager.
- NPM consists of two main parts :
  1) a **CLI (command-line interface)** tool for publishing and downloading packages,
  2) an **online repository** that hosts JavaScript packages.
- NPM gets installed along with the installation of node.js. To verify the presence of npm package on your machine install following command –

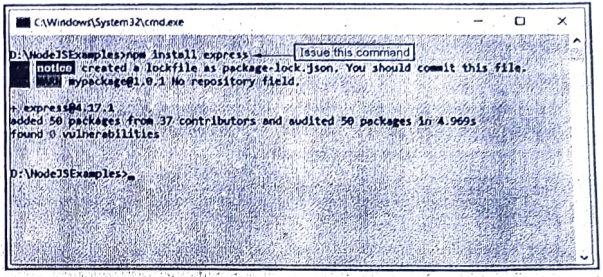


Review Question

1. *Explain the concept of NPM.*

4.11 Install External Modules

For installing any node.js module we have to use **install** command. For example –

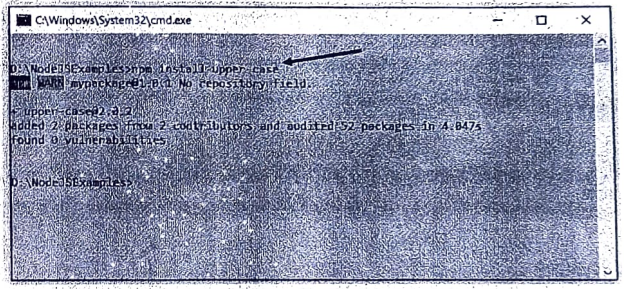


- As soon as we issue this command the package-lock file gets generated in the current folder.
- For using this module in our program we have to use following command –

```
var ex = require('express');
```

Example of using the existing module

Step 1 : We will install the module **upper-case** by issuing the install command

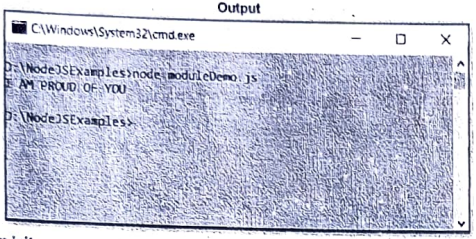


Step 2 : You can locate this module inside **your_folder/node_modules**. Now open any suitable text editor and write node js script that makes use of this module.

moduleDemo.js

```
var upp_ch = require('upper-case');
console.log(upp_ch.upperCase('I am proud of you'));
```

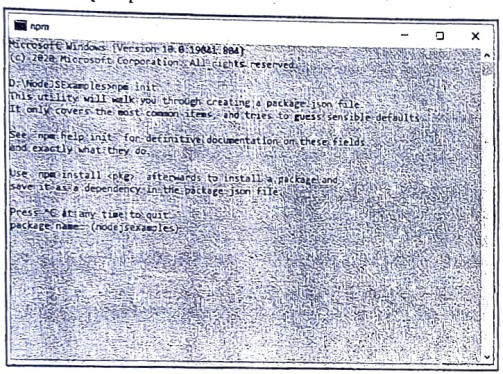
Output



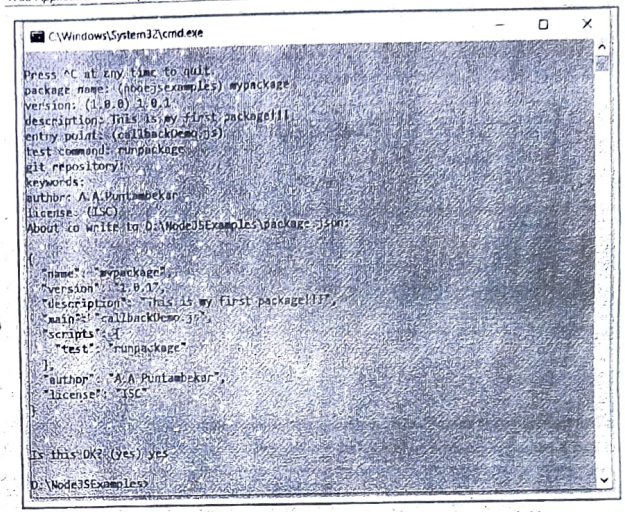
Use of npm init

- We can create our own package using following steps

Step 1 : Create a sample folder. Open command prompt and go to that directory location. For instance – I have created a folder named **nodejsexamples** . Issue the command **npm init** at the command prompt.



Step 2 : Now fill the information about the package.



Once you execute the above commands, the **package.json** gets created. You can open and test it using simple notepad.

4.12 Create HTTP Server

The Web module is mainly controlled by a web server that handles the requests and provides the responses. These requests and responses are handled by HTTP(Hyper Text Transfer Protocol) protocol

The user submits the request using web browser. Thus the communication between web server and web browser is handled by HTTP protocol. Following steps explain this communication -

- When user submits a request for a web page, he/she is actually demanding for a page present on the web server.
- When web browser submits the request for a web page, the web server responds this request by sending back the requested page to the web browser of the client's machine.

Fig. 4.12.1 Client-Server communication

Step 1 : Web client requests for the desired web page by providing the IP address of the website.

Step 2 : The web server locates the desired web page on the website and responds by sending back the requested page. If the page doesn't exist, it will send back the appropriate error page.

Step 3 : The web browser receives the page and renders it as required.

Example Code

Node.js has a built in module named **http** which allows us to transfer data over the Hyper Text Transfer Protocol(HTTP)

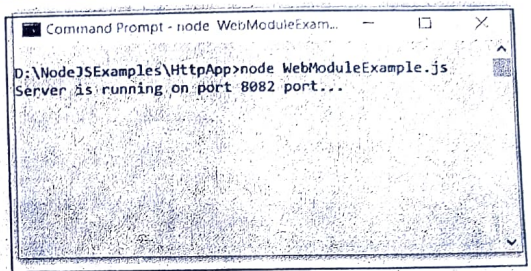
In this section we will build a web server using **http** module.

WebModuleExample.js

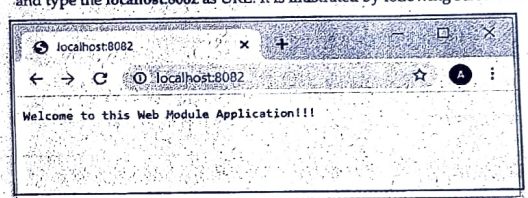
```
var http = require("http");
var server = http.createServer(function(request,response) {
    response.writeHead(200,{"Content-Type": "text/plain"});
    response.end("Welcome to this Web Module Application!!!");
});
server.listen(8082);
console.log("Server is running on port 8082 port...");
```

Output

Step 1 : First of all open command prompt and type following command.



Step 2 : Now open some suitable web browser such as Mozilla Firefox or Google Chrome and type the **localhost:8082** as URL. It is illustrated by following screenshot



Script Explanation : In above given Node.js script,

1) We have imported **http** module.
2) The **http.createServer()** method includes **request** and **response** parameters which is supplied by Node.js.
3) To send a response, first it sets the response header using **writeHead()** method and then writes a string as a response body using **write()** method.
4) Finally, Node.js web server sends the response using **end()** method. But we can directly write the response body using end method.
5) Using **server.listen** method we can specify the port number on which our web server is running.
6) Just refer the illustrative output given above.

Handling various Http Requests

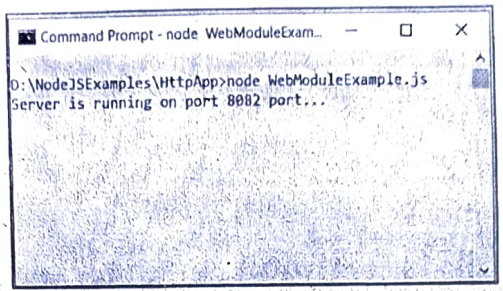
The **Http.request** object is used to get information about the current Http request. The response object is used to send the response for current HTTP request.

Following Node.js script shows how different HTTP requests are made and how to respond them using response object of HTTP module.

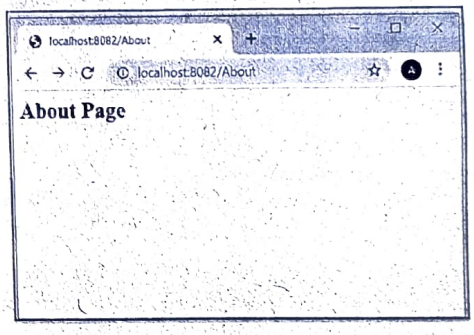
WebModuleExample.js

```
var http = require("http");
var server = http.createServer(function(request,response) {
   if(request.url == '/') {
      response.writeHead(200,{"Content-Type": "text/html"});
      response.write("<html><body><h1>Welcome to this Web Module
      Application!!</h1></body></html>");
      response.end();
   }
   else if(request.url == "/About"){
      response.writeHead(200,{"Content-Type": "text/html"});
      response.write("<html><body><h2>About Page</h2></body></html>");
      response.end();
   }
   else if(request.url == "/Contact"){
      response.writeHead(200,{"Content-Type": "text/html"});
      response.write("<html><body><h2>Contact Information</h2></body></html>");
      response.end();
   }
   else
      response.end("This is invalid request");

});
server.listen(8082);
console.log("Server is running on port 8082 port...");
```


Output



We can open a web browser and get the about page by make the request using the URL as "localhost:8082/About". The illustrative output is as given below -



4.13 Create Socket Server

Basics of Network Programming

- Socket is the most commonly used term in network programing. Socket provides the way by which the computers can communicate using **connection oriented** or **connection less communication**.
- A socket is basically an **endpoint** of a two-way communication link between two programs running on the network. Typically these programs are Server program and Client program.
- Thus socket is OS-controlled interface into which the applications can send or receive messages to and fro from another application.
- A socket is bound to a port number so that the TCP/UDP from transport layer can identify the corresponding application at destination.

Creating TCP server-client communication Application

- Transmission Control Protocol(TCP) is a connection-oriented, reliable protocol which supports the transfer of data in continuous streams.
- The addressing scheme used in TCP is makes use of **ports**. On separate ports the communication can be established concurrently by the system. During this communication, the server waits for the client to get connected to a specific port for establishing communication. This type of communication is called as **socket programming.**
- There are some ports which are reserved for specific service. These ports are called as reserved ports. Various port numbers specifying their services are given in the following table

Port number	Service
21	FTP
23	Telnet
25	SMTP
80	HTTP
110	POP3

- **User level processes** or services generally use port numbers >=1024.
- A socket is bound to a port number so that the TCP/UDP from transport layer can identify the corresponding application at destination.

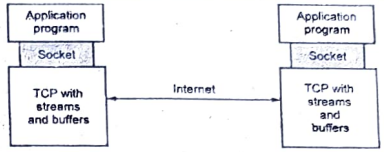


Fig. 4.13.1 Client Server communication using TCP

- **Server** is a device which has resources and from which the services can be obtained. For example : there are various types of servers such as web server which is for storing the web pages, there are print servers for managing the printer services or there are database servers which store the databases.
- **Client** is a device which wants to get service from particular server.
- First of all server starts and gets ready to receive the client connections. The server-client communications occurs in following steps

Sr. No.	Client	Server
1.	Creates TCP socket.	Creates TCP socket.
2.	Client initiates the communication. i) Communicate.	Creates another socket for listening client. i) Accept message from client and repeatedly communicate.
3.	Close the connection.	Close the connection.

We have two build to node.js programs - One for server and other for client.

TCP Server Program

To build the TCP server program we have to follow the steps as given below -

Step 1 : We use "net" module by adding following line at the beginning of both the programs -

```
var net = require('net');
```

Step 2 : Now we use the **createServer()** method for creating the stream based TCP server. For that purpose we have to add following code in the program.

```
var server = net.createServer();
```

Step 3 : We use the event handler **on** for the server when any client gets connected to it. In this event handling code we use the **socket** class. The attributes

remoteAddress and **remotePort** will return the remote address and port number of the client which is connected currently to the server.

Step 4 : Then using socket instance we can handle three events such as **data**,**close** and **error**.

- For **data** event, the data received from the client is displayed on the console using **cosole.log** method.
- For **close** event, the connection closing message is displayed to the client.
- If any error occurs during establishment of connection, the **error** event is fired, and error message is displayed on the console.

Step 5 : During this entire, client server communication Server must be in listening mode. This can be done using **listen** method. To this method, the port number is passed. The **same port number** must be used in the client program, so that both the server and client can communicate on the same port number.

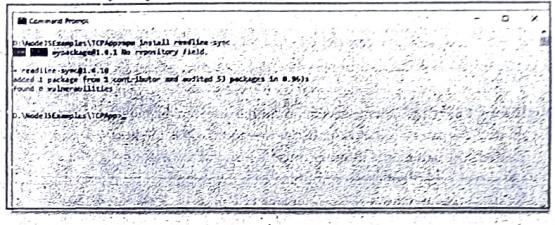
The complete code for server program using TCP is as given below -

server.js

```
var net = require("net");

var server = net.createServer();

server.on("connection",function(socket){
    var clientAddress = socket.remoteAddress+":"+socket.remotePort;
    console.log("Connection with the client %s is established!!!",clientAddress);

    socket.on("data",function(mydata){
        console.log("\n Data received from client is: %s",mydata.toString());
        //sever sending data to the client
        socket.write("I am fine");
    });
socket.once("close",function(){
        console.log("Connection with %s is closed!!!",clientAddress);
});
socket.on("error",function(err){
cosole.log("Connection error %s",err.message);
});
});
server.listen(8082, function(){
    console.log("Server is listening to %j",server.address());
});
```


TCP Client Program

Following is a client program, in which we are getting some input from the user through command prompt and sending that data to the server. For reading the input from keyboard we need to install **readline-sync** module. For installation of this module, open the command prompt. The installation of this module is as follows -



To build the TCP client program we have to follow the steps as given below -

Step 1 : We use "net" module by adding following line at the beginning of the client program.

```
var net = require("net");
```

Step 2 : As our server is running on port number 8082 and the **localhost** as the host, the client must run on the same for establishing the communication between server and the client. The instance of socket class is created and it is named as var **client.**

```
const client = new net.Socket();
```

Step 3 : Now it is possible to establish a connection with the server using **connect** method. For establishing the connection we should pass the same hostname and port number as parameter.

Step 4 : Client can send the data to the server using **write** method. Here we are getting the data when user enters some data through keyboard. For getting the data from the console, we use the **readline-sync module's question method. The** required code for this operation is -

```
var read_line = require('readline-sync');

var mydata = read_line.question("Enter some data for sending it to server: ")
    client.write(mydata);
```

Step 5 : Two more events can be handled by the client program those are – **data** and **end.**

- On the **data** event, the data received from the server is displayed on the console window, using **console.log**

```
client.on('data',function(d){
    //body
});
```

- On the **end** event, the we can acknowledge the server than now client is going to end.

```
client.on('end',function(){
    //body
});
```

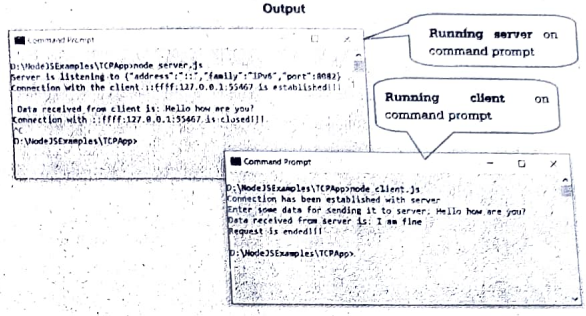
The complete code for client program using TCP is as given below -

client.js

```
var net = require('net');
var read_line = require('readline-sync');

const PORT = 8082;
const HOST = 'localhost';
// create a TCP client
const client = new net.Socket();

client.connect(PORT,HOST,function(){
    console.log('Connection has been established with server');
    //client sending data to server
    var mydata = read_line.question('Enter some data for sending it to server: ');
    client.write(mydata);

});
client.on('data',function(d){
    console.log('Data received from server is: %s',d);
    client.end();
});
client.on('end',function(){
    console.log('Request is ended!!!');
});
```

For getting the output, open the two command prompt windows - One for executing the server and other for executing the client program. Execute the server program first and then client. The following output illustrates this application.

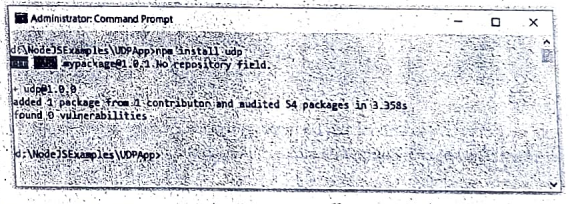
Output



Creating UDP server-client Communication Application

- A *datagram* is an independent, self-contained message sent over the network whose arrival, arrival time and content are not guaranteed.
- It is a basic transfer unit associated with a packet-switched network. The applications that communicate via datagrams send and receive completely independent packets of information.
- For using UDP client server communication, the first and foremost thing which is required is installation of UDP module.
- Following screenshot shows how to install **udp** module in the current working directory.

Installing UDP Module



We have two build to node.js programs - One for server and other for client.

UDP Server Program

To build the UDP server program we have to follow the steps as given below -

Step 1 : We use "udp" module by adding following line at the beginning of both the programs -

```
var udp = require('udp');
```

Step 2 : Now we use the **createSocket** method for creating the UDP sockets. For that purpose we have to add following code in the program

```
var server = udp.createSocket("udp4");
var client = udp.createSocket("udp4");
```

we create a UDP socket which will be udp4 for IPV4.

Step 3 : We use the event handler on for the server when any client gets connected to it.we can handle three events such as **message ,listening, close** and **error.**

- For **message** event, the data received from the client is displayed on the console using **cosole.log** method. Similarly we can send the data to the server using **send** method.
- For the **listening** event, the port number and ip address of the server on which it is running is displayed.
- For **close** event, the connection closing message is displayed to the client.
- If any error occurs during establishment of connection, the **error** event is fired, and error message is displayed on the console.

Step 4 : During this entire, client server communication Server must bind to one particular **port number**. This can be done using **bind** method. To this method, the port number is passed. The same port number must be used in the client program, so that both the server and client can communicate on the same port number.

The complete node.js code for UDP server is as given below -

server.js

```
var udp = require('udp');

var server = udp.createSocket("udp4");
var client = udp.createSocket("udp4");

server.on("error",function(err){
    console.log("Error!!!" + err.message);
    server.close();
});
```



```
server.on("message",function(msg,info){
    console.log("Data received from client is: " + msg.toString());
    var msg="Hello friend";
    //sending data to the server
    server.send(msg,info.port,"localhost",function(err){
        if(err)
            client.close();
        else
            console.log("Data sent!!!");
    });

});

server.on("listening",function(){
    var address = server.address();
    var port = address.port;
    var ipaddr = address.address;
    console.log("Server is listening to port " +port);
    console.log("Server's IP Address: " +ipaddr);
});
server.on("close",function(){
    console.log("Closing the server socket now!!!");
});
server.bind(8082);
```

To build the UDP client program we have to follow the steps as given below -

Step 1 : We use "udp" module by adding following line at the beginning of the client program.

```
var udp = require('udp');
```

Step 2 : Now we use the **createSocket** method for creating the UDP sockets. For that purpose we have to add following code in the program

```
var client = udp.createSocket("udp4");
```

Step 3 : We use the event handler on for the client when any server gets connected to it. For that purpose we the **message** event handler.For **message** event, the data received from the server is displayed on the console using **cosole.log** method.

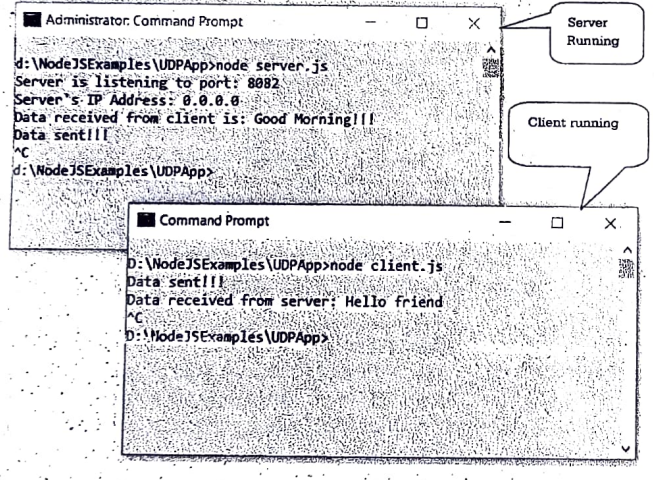
Step 4 : Similarly we can send the data to the server using **client.send** method. Note that during this entire communication the same port number and host name is used which was used for server program.

The complete code for client program using TCP is as given below -

client.js

```
var udp = require('udp');
var client = udp.createSocket('udp4');

var mydata = 'Good Morning!!!';

client.on('message', function(msg,info){
    console.log('Data received from server :'+msg.toString());
});
client.send(mydata, 8082,'localhost', function(err){
    if(err)
        client.close();
    else
        console.log('Data sent!!!');
});
```

Output



Review Question

1. *What is socket ? Write a client server communication in node.js using socket programming.*

4.14 Microservices- PM2

- Microservices are architectural approaches which allows the developers to compartmentalize individual components of a larger application infrastructure. That means each component in the application can work independently.
- The developers can upgrade or modify the components without impacting the larger applications.
- The basic idea behind the creation of microservices is that - Instead of containing everything in a single unit, the microservices-based application is broken down into smaller, lightweight pieces based on a logical construct. The application consists of independent small (micro-) services, and when we deploy or scale the app, individual services get distributed within a set. Refer Fig. 4.14.1.

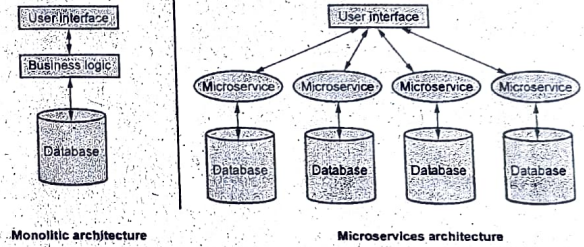


Fig. 4.14.1 Concept of microservice architecture

- PM2 stands for Process Manager 2. It is a versatile process manager written in Node.js.
- It is a free open source, efficient and cross platform process manager with **built in load balancer.**
- It is useful for real time app monitoring, efficient management of microservices and shutdown of applications.

Features of PM2

1) **Restarting after crashes** : PM2 allows to run the process running even after crashing.

2) **Restart persistence** : PM2 remembers all the running processes and restart them after booting.

3) **Remote monitoring and management** : With PM2 it is possible to keep track of remotely running processes and these processes can be easily managed.

4) **Performance** : Node.js event driven I/O model along with PM2 microservice strategy can handle an extreme amount of load with lesser response time.

5) **Built-in clustering** : PM2 handles all the logic internally so there is no need to make any changes in the code.

6) **Log management** : PM2 has built-in log management. It collects log data from your applications and writes it down into a single file.

Installation of PM2

The PM2 can be installed globally after installation of Node.js. The command for installing PM2 is as follows –

```
npm i -g pm2
```

To start pm2 just execute following command at the terminal window.

```
pm2 start <app>
```

Review Question

1. *Write a short note on PM2- microservices.*

Part II : ExpressJS

4.15 Introduction to ExpressJS

Express is web application framework used in node.js. It has rich set of features that help in building flexible web and mobile applications.

Features of Express

1) **Middleware** : Middleware is a part of program, that accesses to the database and respond to client requests.

2) **Routing** : Express JS provides a routing mechanism so that it is possible to reach to different web pages using the URLs.

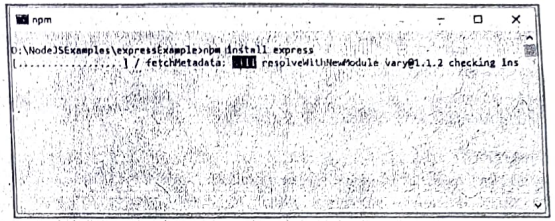
3) **Faster Server Side Development** : It is very convenient to develop server side development using Express JS.

4) **Debugging** : ExpressJs makes debugging easier by providing the debugging mechanism.

Installing Express

Express is installed using Node Package Manager(npm). The command issued for installing the express is,

Prompt:>**npm install express**

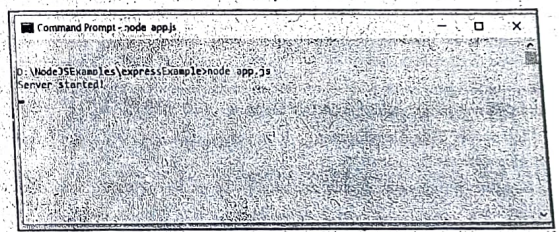


Following is a simple application program that uses Express JS for displaying "welcome message" on the web page.

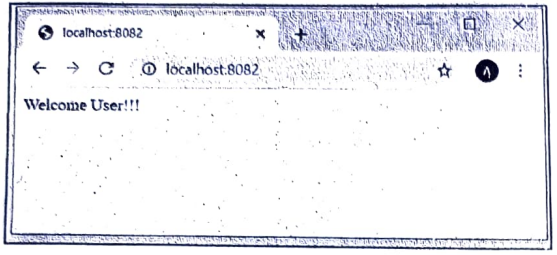
app.js

```
var express = require('express');
var app = express();
app.get('/', function(req,res) {
    res.send("Welcome User!!!");
});

var server = app.listen(8082, function(){
    console.log("Server started");
});
```

Output



Open some web browser and issue the URL "localhost:8082"



4.16 Configure Routes

Routing is a manner by which an application responds to a client's request based on particular endpoint. The sample endpoints are,

localhost:8082/aboutus

localhost:8082/contact

localhost:8082/home

There are various types of requests such as GET, POST, PUT or DELETE. While handling these requests in your express application, the request and response parameters are passed. These parameters can be used along with the **send** method.

Following application represents how to performing routing using express

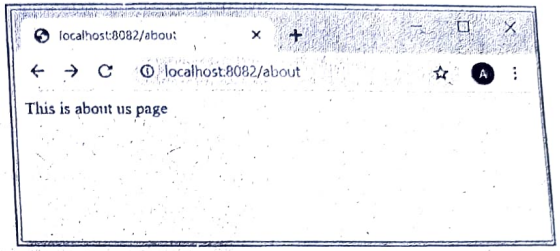
app.js

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

app.get('/', function(req,res) {
   res.send('Welcome User!!!');
});
//routing to 'about us' page
app.get('/about', function(req,res) {
   res.send('This is about us page');
});
//routing to 'contact' page
app.get('/contact', function(req,res) {
   res.send('This is displays contact information');
});
```



```
var server = app.listen(8082, function(){
   console.log('Server started!');
});
```

Output



Review Question

1. *Write a code using expressJs to illustrate the concept of routes.*

4.17 Template Engines

- Template engine is a package that renders the data or values in HTML pages.
- The most important feature of template engine is its **ability to inject data** from server into an HTML template and then send the final html page to the client side.
- Basically with the help of template engines a variable is created in our server(let us say in our expressJS script) and then at run time inserted in an html template.
- There are several template engines that work with ExpressJS. Some of the popularly

used template engines are –

- pug(Formerly known as jade)
- mustache
- dust
- handlebars
- ejs

- For demonstration of template engines, we use the template engine named **ejs**.
- EJS files have the extension .ejs and contains html which has portions of server variables written in EJS syntax. This EJS file is stored in a folder called **views** where express checks by default for rendering web pages. Note that **views** is a standard directory or folder name for storing the template files.
- For using the template engine, first of all we have to set the view engine to ejs. Following code fragement is used for that purpose -

```
app.set('view engine', 'ejs')
```

- We also need to create a file with the file-extension .ejs in views folder. In the expressJS we need to render the variable to be displayed on the browser window.

Let us now know how to create and use template engines

Demo Example

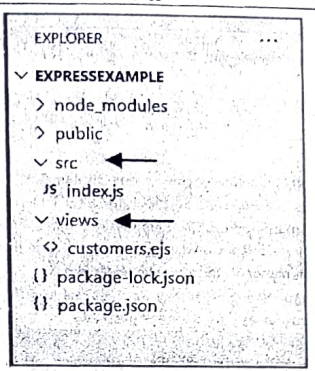
In this example we will create a json object for **customers**. The name of the desired customer is displayed at the runt time using template engine.

Step 1 : Install the template engine package ejs using following command

E:\NodeJSExamples\expressExample> npm install –save ejs

You can verify that the package **ejs** is added to your **expressJS** by just opening the **package.json file**.

Step 2 : Now create a folder named views in the current working directory. Also create **src** folder. Inside the **src** folder, create a **index.js** file. The following screenshot explains the folder structure.



The code for **index.js file is as follows –**

index.js

```
var express = require('express')
var app = express()

app.set('view engine', 'ejs');

app.get('/', function (req, res) {
 res.send('<h1>Welcome User</h1>')
})

var customers = {
 1:'Supriya',
 2:'Siddharth',
 3:'Aniket',
 4:'Swati',
 5:'Devendra'
};
app.get('/customers/:id',function(req,res){
 res.render('customers',{custname:customers[req.params.id]})
})
app.listen(3082)
```

Calling the template engine named 'ejs'

This is a default home page which will display simply the welcome message

Create json object which contains customer's id and name

This will send the value in the variable **custname** to template engine rendering page. We will create this page in the next step

Script Explanation :

- Once we have installed EJS, we can call it into your Express app using following code,

```
app.set('view engine', 'ejs');
```

- Then we create a json object which will store the id and name of the customers.
- Using **res.render** function we will pass the names of the customers to the **customers.ejb** file

Step 3 : Now create a **customers.ejb** file inside the views folder(Note that we have already created this views folder in step 2.

customers.ejb

```
<html>
  <head>
    <title>Template Engine Demo</title>
  </head>
  <body>
    <h2>Customer's Name is <%=custname %></h2>
  </body>
</html>
```

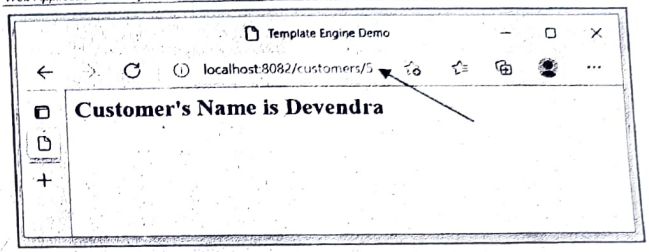
Script Explanation : In above script,

- We have written the script in simple HTML.
- EJS uses the syntax <%= ... %> to represent variables in HTML code.
- The variable name **custname** is obtained from **index.js** file. Recall that in step 2, in our **index.js** file we have rendered the **custname** variable. This is the name of the customer obtained from the customer object. This customer object is created in json format in index.js file itself. So the value of the variable is injected in this .ejb file with the help of template engine named **ejb**.

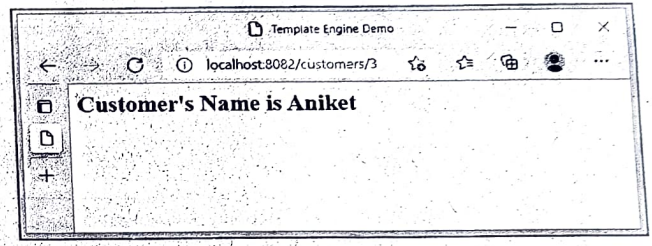
Step 4 : Open the terminal window and issue the command

E:\NodeJSExamples\expressExample> nodemon src/index.js

Now, open the web browser and enter URL localhost:8082/customers/5



Now, change the id of the customer. I have changed to id 3 and the output will be



Review Question

1. *What is template engine ? How create and use it using expressJS ?*

4.18 ExpressJS as Middleware

- Middleware comes in the middle of request and response cycles of Node.js execution.
- Middleware functions are functions that have access to the request object (req), the response object (res), and the next function in the application's request-response cycle.
- The **next** function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.

- Middleware functions can perform the following tasks :
  - Execute any code.
  - Make changes to the request and the response objects.
  - End the request-response cycle.
  - Call the next middleware in the stack.
- To load the middleware function, call **app.use()**, specifying the middleware function.

Demo Example

In the following code loads the **myLogfunction** middleware function before the route to the root path (/).Every time the app receives a request, it prints the message "Displaying contents..." with current date to the terminal.

Step 1 : Create a folder named **src** and store the .js file in it. I have named it as **index.js**.

index.js

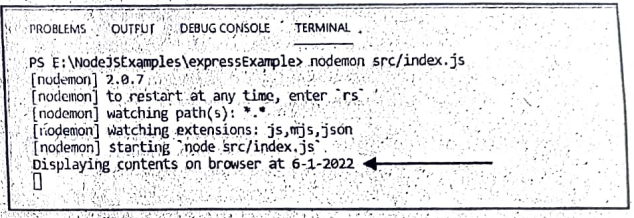
```
var express = require('express')
var app = express()
//creating middleware function
var myLogfunction = function (req, res, next) {
    let date_obj = new Date();
  console.log('Displaying contents on browser at '+date_obj.getDate()+'-'
+(date_obj.getMonth() + 1) + '-' + date_obj.getFullYear())
  next()
}
//using middleware function
app.use(myLogfunction)

app.get('/', function (req, res) {
  res.send('<h1>Welcome User</h1>')
})

app.listen(8082)
```

Step 2 : Output : Open the terminal window issue the command nodemon(you can use node command as well)

E:\NodeJSExamples\expressExample> nodemon src/index.js

Open the web browser and type the URL localhost:8082. The output will be as follows –

At the same time, see the terminal or command prompt window at which we have executed our expessJs application. It will be showing that our **myLogfunction** is executing and displaying the current date in the form of dd-mm-yyyy.



Review Question

1. *Write a short note on ExpressJS as middleware.*

4.19 Serving Static Files

Static files can be simple html, css or JavaScript files. It is possible to invoke those files using expressJs code.

For serving the static file, expressJS makes use of **app.use()** method.

The **app.use()** method mounts the middleware **express.static** for every request. The **express.static** middleware is responsible for serving the static assets of an Express.js

application. Inside the **express.static** method the path for the desired static file must be specified.

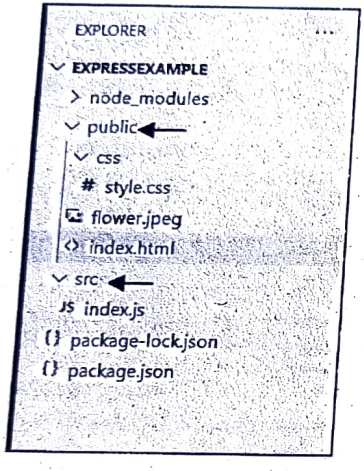
Demo Example

In this example, we will serve the static file .html. The elements in this static html file are styled using external .css file. We will write expressJS code which will serve the static html named **index.html**.

Let us see the step by step procedure to do so.

Step 1 : Create a folder for storing the ExpressJS application programs. I have created it by the name **expressExample**.

Step 2 : Inside **expressExample** create a two folders named **src** and **public**. The idea here is that the **src** folder will store the expressJS code(named it as **index.js**) for executing the application. The **public** folder will contain the **.html** file. I have named this file as **index.html**. The screenshot for the folder structure is given below for better understanding of the locations of the file.



Let see the code

The simple HTML file is created. This is the static file which will get served by the expressJS code.

index.html

```
<html>
  <head>
  <title>Serving Static File</title>
  </head>
  <link rel="stylesheet" href="css/style.css">
  <body>
    <h1>Welcome User</h1>
    <img src="flower.jpeg"/>
  </body>
  </html>
```

Step 3 : Inside the **public** folder created in step 2, we will create another folder for creating a CSS stylesheet. I have named the folder as css and inside this css folder I have created a stylesheet by the filename **style.css**. The stylesheet contains following code

style.css

```
h1 {
  background-color: khaki;
  color: red;
  padding: 10px;
  border:solid;
  text-align: center;
  border-color: black;
}
img {
  display: block;
  margin: 0 auto;
}
```

This will display the image in a block and will be aligned at center

Step 4 : Inside the src folder created in step 2, I will create **index.js** file. In this file the expressJS code we will be serving the static files(which we have created in Step 2 i.e. index.html and style.css).

index.js

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

```
const filePath = path.join(__dirname,"./public");
app.use(express.static(filePath));

var server = app.listen(8082, function(){
    console.log("Server started!");
});
```

Script Explanation : In above script,

1) We have obtained the html file path using the statement

```
const filePath = path.join(__dirname,"./public");
```

The variable **filePath** will store the relative path for the **index.html** file present in our **public** folder. The **__dirname**(Here dirname is preceded by two underscores) is an environment variable that tells you the absolute path of the directory containing the currently executing file. It is joined with "./public" because out index.html file is stored inside the **public** directory.

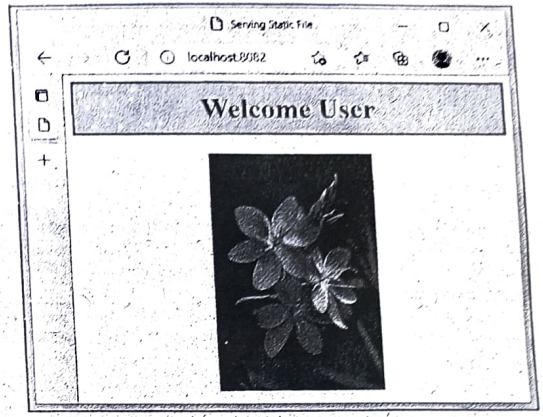
We need to import path module. So just add **const path = require('path');** at the beginning of this **index.js** file. By the above line of code we get the index.html file path in **filePath** variable.

2) Then we use **app.use** method. The **app.use()** method mounts the middleware **express.static** for every request. The **express.static** middleware is responsible for serving the static assets of an Express.js application. The **express.static()** method specifies the folder from which to serve all **static resources**. Hence inside the **express.static** method we passed the variable **filePath** which actually contains the path for our static **index.html** file.

Step 5 : Output : Open the terminal window issue the command nodemon(you can use node command as well)

E:\NodeJSExamples\expressExample> nodemon src/index.js

Open the web browser and type the URL localhost:8082. The output will be as follows –



4.20 REST HTTP Method APIs

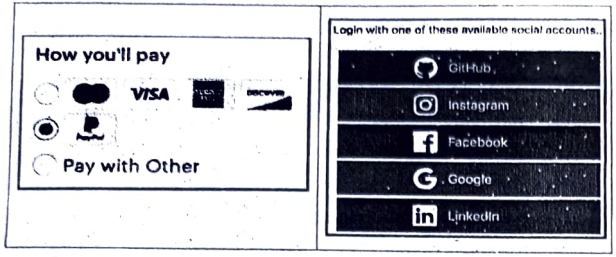
What is API ?

API stands for **Application Programming Interface**. APIs are a set of functions and procedures that allow for the creation of applications that access data and features of other applications, services, or operating systems.

In other words, if you want to interact with a computer or system to retrieve information or perform a function, an API helps you communicate what you want to that system so it can understand and fulfil the request.

It is basically a software intermediary that allows two applications to talk to each other. For example - When we use facebook, send a message or check the weather on our mobile phone, we use API.

For example - If we want to buy a movie ticket, we go to the online ticket booking site, enter movie name, use credit card information, get the ticket, print it. The software that interacts with us to perform all these task is nothing but the API. Following screenshots some well known examples of API.



What is REST API ?

REST stands for Representational State Transfer. It is a set of rules that developers follow while creating their API.

Each URL made by the client is a **request** and data sent back to the client is treated as response.

The request consists of,

1. End Point 2. Method 3. Headers 4. Data

(1) Endpoint

The endpoint (or route) is the URL you request. For example -

http://localhost:8082/

(2) Methods

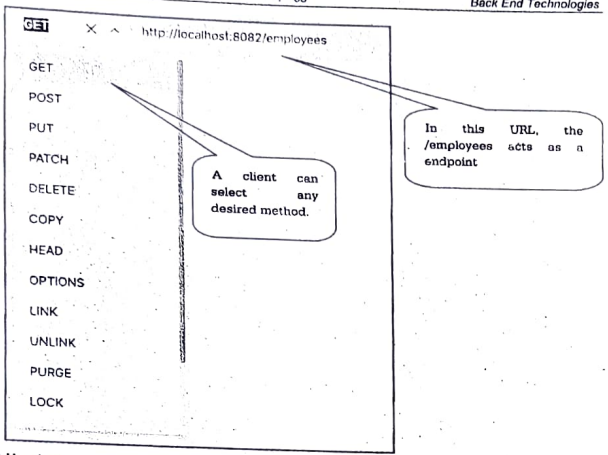
There are various types of methods. But the most commonly used one are as follows -

1) GET 2) POST 3) DELETE
4) PUT 5) PATCH

These methods are basically used for performing CRUD operations. Here C- stands for Create, R- stands for Read, U- stands for Update and D-stands for Delete.

Method	Purpose
GET	For getting the data from the server, this request is used. The server looks for the data the client has requested and sends it back to the client.
POST	This method is used when new resource is to be created on the server. When client uses this method and provides the data, then the server creates a new entry in the database and acknowledges the client whether the creation is successful or not.
PUT and PATCH	These are the two methods that are used when the client wants to update some resource. When PUT method is used, then that means client wants to update entire resource. And when PATCH method is used, then that means client simply wants to update small part of the resource.
DELETE	This request is used to delete a resource from the server.

There are some software tools such as Postman, that help the client to select the appropriate method and issue the URL. Following screenshot is of a software tool **Postman** , using which we can select appropriate method.



(3) Headers

Header are used to provide authentication and other useful information to client and server. The HTTP headers are property-value pairs separated by Colons(:).

For example -

"Content-Type: application/json"

(4) Data

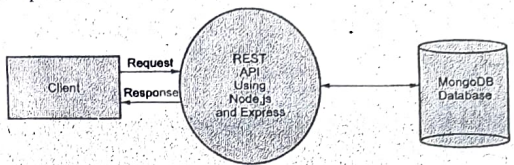
The DATA contains information which the client wants to send to the server. While sending the DATA to the server the methods such as POST, PUT, PATCH or DELETE are used.

It is preferred to send the data in JSON format. The format is,

```
{
  Property1:value1,
  Property2:value2
}
```


Concept of Node JS API

The Node.js script is popularly used to create REST API. We will write the server script in **Node.js.** The database is created using **MongoDB**. The client interface is handled with the help of **Postman.**



Let us understand how to create Node JS REST API with the help of following example

Example Code

Prerequisite: For creating the Node.js Rest API we need following software to be installed in our machine

1) **Postman :** This is an API client. This client software allows us to enter the request in JSON format and select the commands such as GET, POST, UPDATE, DELETE and so on.
2) **MongoDB :** This is a database package. We need MongoDB compass to be installed along with the mongoDB server application. The MongoDB compass is a graphical tool for handling database operations.
3) **NodeJS :** This is of course used for creating an API in .js file.

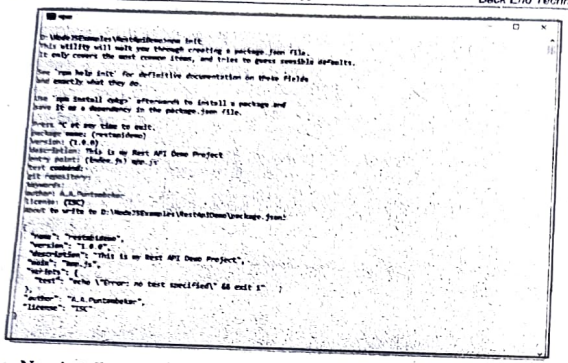
Step 1 : Make a new folder in your current working directory for creating Rest API. Here I am creating a folder named RestApiDemo

Step 2 : Now open the command prompt window and create package.json file using **npm init** command. Following screenshot illustrates the same.

At the command prompt simply type the command

```
npm init
```

Then hit the enter button. You can add the values of your choice or simply go on pressing enter key and accept the default value.

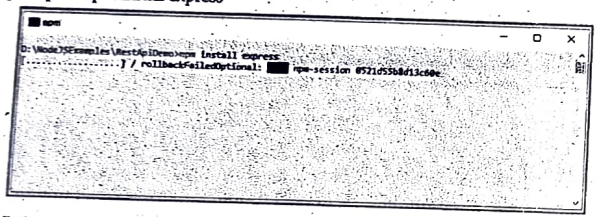


Step 3 : Now install some more packages that are required for creating this Rest API

Installation of express module

Issue the following command

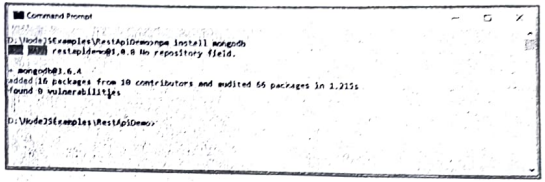
prompt:\>npm install express



Installation of mongodb module

Issue the following command

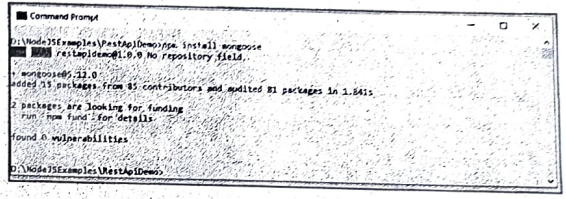
prompt:\>npm install mongodb



Installation of mongoose module

Issue the following command

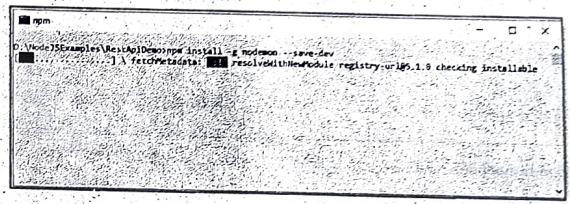
prompt:\>npm install mongoose



Installation of nodemon

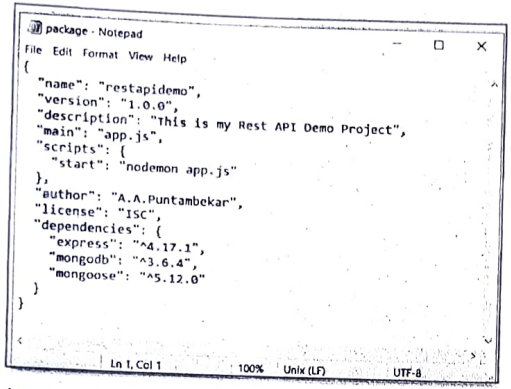
Issue the following command at the command prompt

prompt:>npm install -g nodemon –save-dev



The sample **package.json** file will show you all these dependencies of the modules you have installed just now

package.json



Note that I have added following line in **package.json** file so that as soon as I save any changes in the file, the output gets reflected immediately. I need not have to run the **app.js** file every time.

"start" : "nodemon app.js"

Step 4 : Now create your main server in a file named **app.js**.

app.js

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

url = 'mongodb://localhost/EmployeeDB';

const app = express();
mongoose.connect(url, {useNewUrlParser:true})

const con = mongoose.connection  //getting the connection object

con.on('open',() => { //on opening the connection, connecting with database
    console.log('Connected to Database')
})
```

URL for the database name

Connecting to MongoDB database using Mongoose package


```
app.use(express.json())
const employeeRouter = require('./routes/employees') //initial endpoint
app.use('/employees',employeeRouter)
app.listen(8082, () => {
    console.log('Server Started!!!')
})
```

Step 5 : Create a folder named **routes**. Inside the **routes** folder create a file named **employees.js**. This file will handle the GET and POST requests of REST API. Using POST request we can create the API by inserting the data into the database. The GET request will retrieve and display the data from the database. Thus routing of GET and POST requests is done in this file.

employees.js

```
const express = require('express')
const router = express.Router()
const Employee = require('../models/employee')

router.get('/', async(req,res) => {
    try {
        const employees = await Employee.find()
        res.json(employees)
    }
    catch(err){
        res.send('Error is ' +err)
    }
})

router.post('/', async(req,res) => {
    const employee = new Employee({
        name: req.body.name,
        designation: req.body.designation
    })
    try {
        const e1 = await employee.save()
        res.json(e1)
    }
    catch(err){
        res.send('Error is ' +err)
    }
})
module.exports = router
```

This is a router program that routes on receiving the particular type of request

Handling GET command issued by the client, and using find(), displaying data present in the Employee database

Handling the POST request. The data is received here from client. Hence we use **req.body** followed by name of the data field

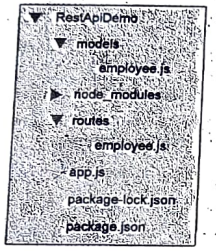
Step 6 : Now create another folder named **models**. Inside **models** folder create a file named **employee.js**. This file will describe the Schema. The fields of the document are described in this file along with data type and some other description is mentioned in this file. Note that the schema is specified in JSON format.

employee.js

```
const mongoose = require('mongoose')
const employeeSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true
    },
    designation: {
        type: String,
        required: true
    }
})
module.exports = mongoose.model('Employee',employeeSchema)
```

Preparing the schema for the database

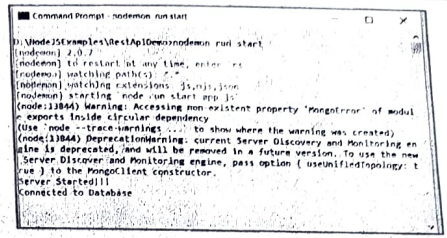
Step 7 : For better understanding, the folder structure of this project can be viewed in the following explorer window



Step 8 : For getting the output, Open the command prompt window and issue the following command.

```
D:\Node.JSExamples\RestApiDemo>nodemon run start
```

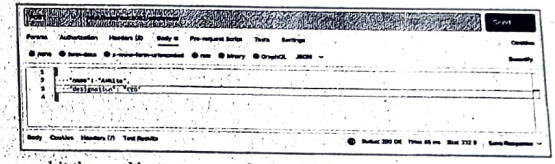
You should get the "Server started" and "Connected to Database" messages



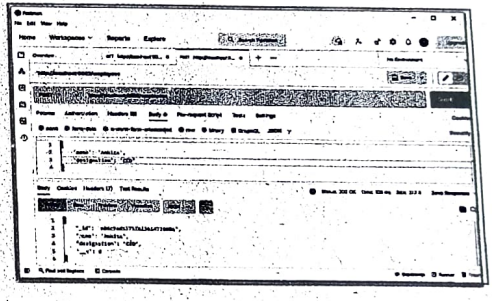
Now, Open the Postman and click on **Create Request**. Select the **POST** command issue the URL as

http://localhost:8082/employees

And then click on **Send** button

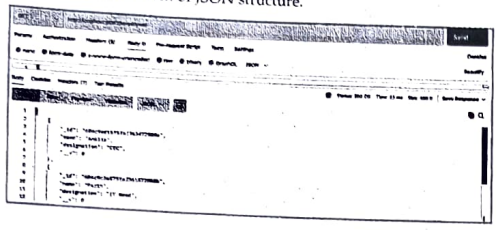


Once we hit the send button we get the result by generating the unique **_id** for the data which we have inserted. It can be viewed as follows,

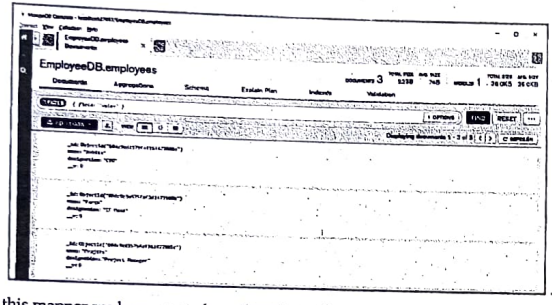


In this, manner we can insert more data by selecting the Post command. This data is sent to the **app.js** server and then getting stored in the MongoDB database.

In order to retrieve data being stored, we can use **GET** method on Postman client and see the complete data in the form of JSON structure.



We can verify the data by opening the database created in MongoDB



In this manner we have created our REST API for employee details.

Review Question

1. *Write and explain a simple application using REST HTTP method APIs in node.js.*

4.21 Applying Basic HTTP Authentication

- Basic authentication works by prompting a Web site visitor for a username and password. This method is widely used because most browsers and Web servers support it.
- **HTTP basic authentication** is a simple **challenge and response mechanism.** In this method, a server can request authentication information such as user ID and password from a client.
- The client passes the authentication information to the server in an **Authorization header.**
- If a client makes a request for which the server expects authentication information, the server sends an HTTP response with a **401 status code,** indicating an **authentication error,** and a **WWW-Authenticate** header. A browser that receives a 401 error understands that it is required to supply correct user name and password.
- Most web clients handle this response by requesting a user ID and password from the end user.
- The **HTTP WWW-Authenticate header** is a response-type header. The WWW-Authenticate header field for basic authentication is constructed as following :

```
Test set (WWW-Authenticate: Basic)
```

- HTTP basic authentication does not have a logout function and the browser will store the credentials until it has been restarted.
- In expressJs the simple package for performing basic HTTP authentication is available which can be installed using the command

```
npm install express-basic-auth
```

- The middleware will check incoming requests for a basic Authorization header, parse it and check if the credentials are submitted or not.
- If there are any credentials, an **authentication** property will be added to the request, containing an object with **user** and **password** properties, filled with the credentials, no matter if they are valid or invalid. If a request is found as not be authorized, it will respond with HTTP 401.

Following is a demo example that demonstrates the basic HTTP authentication

Demo Example

Before running the following application install the **basic-auth** package by issuing the command

```
npm install express-basic-auth
```

node.js

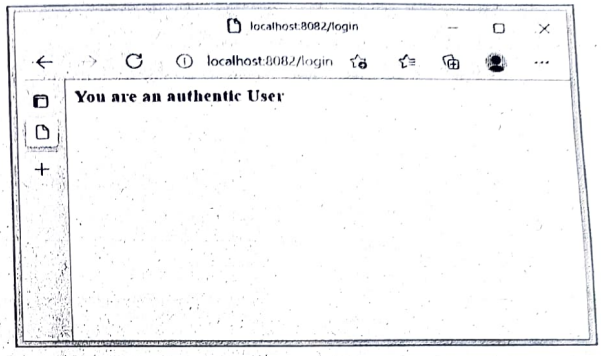
```
var express = require('express');
var app = express();
var basicAuth = require('basic-auth');

var authFunction = function (req, res, next) {
 var user = basicAuth(req);
 if (!user || !user.name || !user.pass) {
  res.set('WWW-Authenticate', 'Basic');
  res.sendStatus(401);
  return;
 }
 if (user.name === 'admin' && user.pass === 'mypassword') {
  next();
 } else {
  res.set('WWW-Authenticate', 'Basic');
  res.sendStatus(401);
  return;
 }
}
app.get('/login', authFunction, function (req, res) {
 res.send('<h3>You are an authentic User.</h3>')
});

app.listen(8082);
console.log('Server running on 8082');
```

The HTTP **WWW_Authenticate** header is a response-type header which is set here.

For invalid user name and password the status code 401 is set.

Output

If we click on **Sign in** we get



Script Explanation : In above script,

1. We have imported **basic-auth** module in variable **basicAuth.**
2. The authentication function named **authFunction** takes three parameters - **req** for request object, **res** for response object and callback function **next** which is called upon successful authentication.
3. The **app.get("/login", authFunction, function (req, res)** function is called from express app instance. The first argument is the URL path, the second argument is the authentication function which will be called for authentication (It is **authFunction**) and the third parameter is function that will handle the incoming request.
4. When the browser get the request **localhost:8082/login** it first calls the **authFunction.**
5. Inside the **authFunction** we set the WWW-Authenticate header field for basic authentication using **res.set('WWW-Authenticate', 'Basic');** Here the valid user name is provided as **admin** and the password as **mypassword.** If user enters this information then the next function will be returned.

6. If the function returns the callback function **next** then the third parameter-function of **app.get(...)** will get executed and the "You are an authenticate User" message will be displayed on the console.
7. If the **authFuntion** finds that the credentials(user name and password) are not valid it sends status code 401 and no callback function is returned.

4.22 Implement Session Authentication

4.22.1 Sessions

Http is a **stateless** protocol. That means that when we load a page in our browser, and then navigate to another page of same website the server has no means of knowing that these requests have originated from same browser or client.

Browser communicating with the server and server responding to the client via browser is called as session.

Sessions is a mechanism which allows to store data for individuals on sever side.

Following is an example of session handling mechanism. In this example we use the **cookie-parser** package. It is basically a middleware which parses the cookies attached to the client request object. The **express-session** package is useful in creating the middleware session.

Example Code

Step 1 : Create a folder in which you can store your source code for demonstrating sessions. I have named this folder as **SessionAppDemo**.

Now create a **package.json** file using following command

```
D:\NodeJSExample\SessionAppDemo>npm init
```

Then install all the required packages such as **express**, **cookie-parser** and **express-session**. Use following commands

```
D:\NodeJSExample\SessionAppDemo>npm install express
D:\NodeJSExample\SessionAppDemo>npm install --save cookie-parser
D:\NodeJSExample\SessionAppDemo>npm install --save express-session
```

Step 2 : Now we will create a source file using .js extension in which the session is created. The source code is as follows -

sessionDemo.js

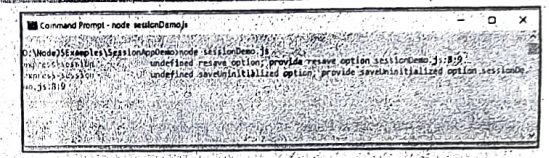
```
var express = require('express');
var cookieParser = require('cookie-parser');
var session = require('express-session');
```



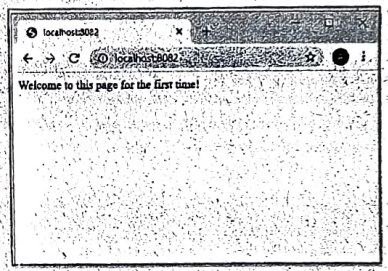
```
var app = express();

app.use(cookieParser());
app.use(session({secret: 'A secret Key'}));

app.get('/', function(req, res){
  if(req.session.viewCount){
    req.session.viewCount++;
    res.send('You visited this page ' + req.session.viewCount + ' times');
  } else {
    req.session.viewCount = 1;
    res.send('Welcome to this page for the first time!');
  }
});
app.listen(8082)
```

Step 3 : For getting the output, first open the command prompt and Issue the command

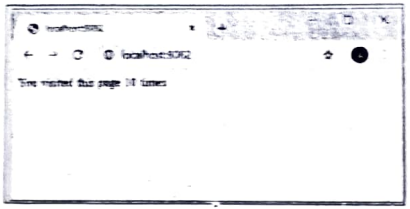
```
D:\NodeJSExample\SessionAppDemo>node sessionDemo.js
```



Now open the suitable web browser and enter the URL



If you just go on refreshing the above page for several times, the output will get changed. For instance it could be something like this -



Script Explanation :

```
var express = require('express');
var cookieParser = require('cookie-parser');
var session = require('express-session');
```

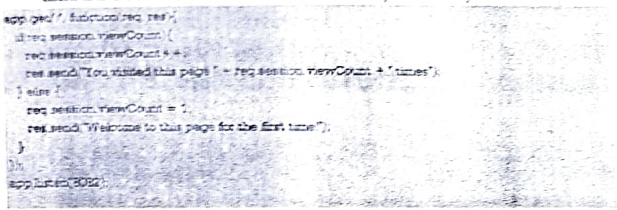
- In this script we use the above three modules

```
var app = express();
```

- Create instance of express in variable **app**

- The **app.use** is a method used to configure the middleware such as **cookie-parser** or **session**.
- The middleware functions determine the flow of request-response cycle. They are executed after every incoming request. These functions are functions that have access to the **request object** (req), the **response object** (res).
- The **cookie-parser** and session are called **third-party middlewares.**
- **cookie-parser**: is a middleware which parses cookies attached to the client request object. Using session middleware we can perform various tasks such as creating the session, setting the session cookie and creating the session object in request object(req).
- **Secret :** Here, the secret is the hash key passed to the header file so that this information can be securely decoded at the server end only. This is important because without a session secret, any third party app can easily look into the cookie with the help of browser and hijack it. Having a secret ensures that when

authenticating, the cookie data is encoded using the secret key only known to the client and the server. this data can then not be easily decoded by a third listener.



- When user visits the web page a new session is created for the user and a cookie is assigned to it. So when next time when user visits the same page the cookie is checked and page view counter updated accordingly.

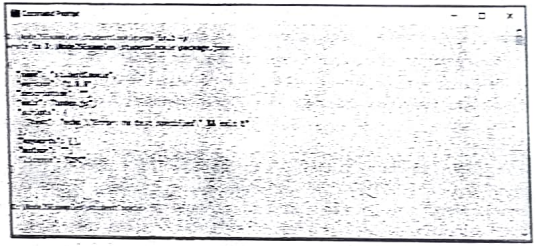
4.22.2 Cookie

Cookie is a small piece of information used to store name-value pair. This cookie data is sent to the client with server request and stored at the client's machine. Every time when user loads the web site, this cookie is sent with the request. This helps in keeping track of the user's actions.

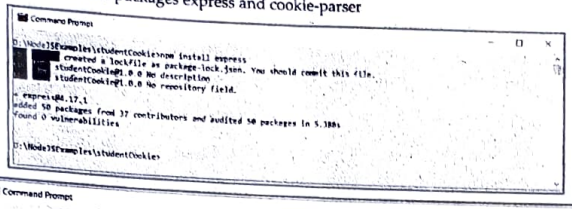
Following example illustrates how to create cookie, display cookie data and clear the cookie.

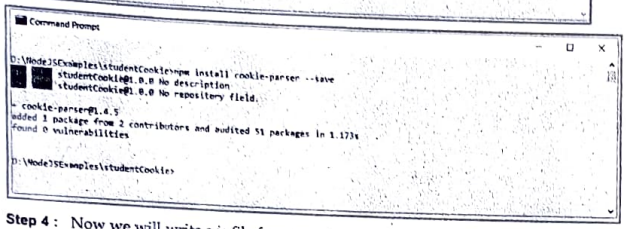
Step 1 : Create a folder by some suitable name in a current working directory. I have created a directory named **studentCookie**

Step 2 : Open the command prompt window and issue the **npm init** command for the created folder. Refer the following screenshot -



Step 3 : Install the packages express and cookie-parser





Step 4 : Now we will write a js file for setting the cookie and reading the cookie. The code is as follows -

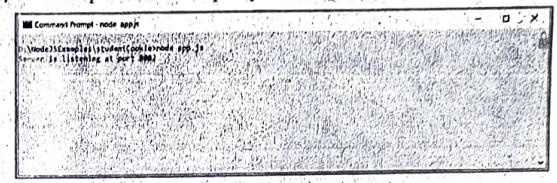
app.js

```
const express = require('express');
const cookieParser = require('cookie-parser');
const app = express();
app.use(cookieParser());


app.get('/',(req,res) => {
    res.send("Cookie Demo");
});

var student = {
    rollno: '101',
    name: 'Parth'
};
//Cookie is added
app.get('/setstudent',(req,res) => {
    res.cookie('studentData',student)
    res.send('Student data added to Cookie');
});
```

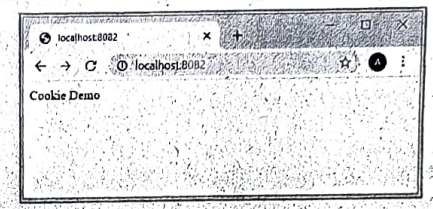


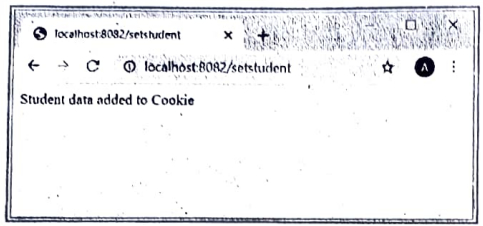
```
//Display Cookie
app.get('/getstudent', (req,res) => {
    res.send(req.cookies);

});

//server listening
app.listen(8082, ()=> {
    console.log("Server is listening at port 8082");
})
```

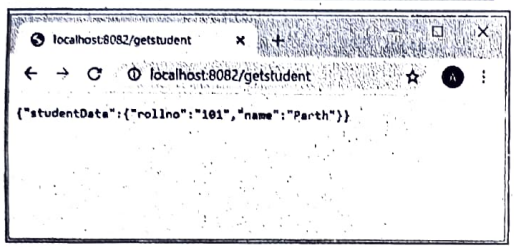
Step 5 : Now open the command prompt and execute the js file as follows -



Now open a web browser and type the url for setting and getting the cookie. Here is the demonstration







Destroy Cookie

For deleting the cookie we use

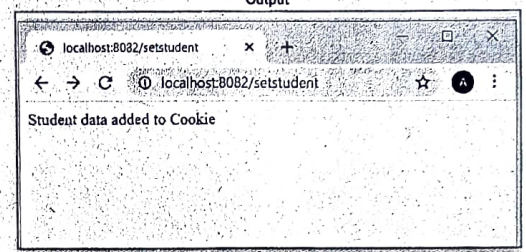
```
clearCookie(cookie Name)
```

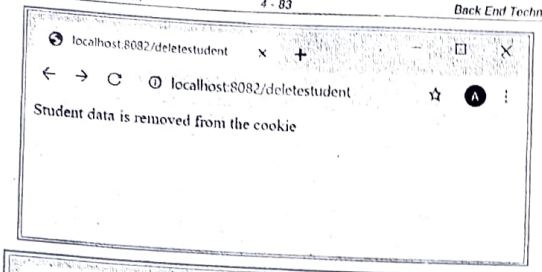
Following code shows how to delete a cookie

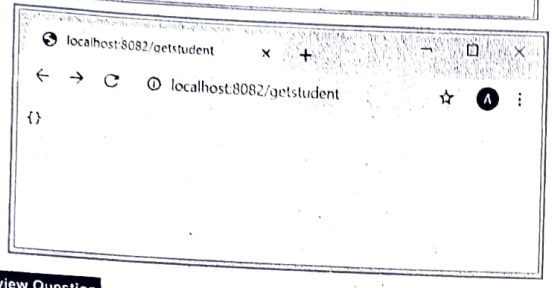
app.js

```
const express = require('express')
const cookieParser = require('cookie-parser');
const app = express();
app.use(cookieParser());


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

var student = {
    rollno: '101',
    name: 'Parth'
};
//Cookie is added
app.get('/setstudent', (req,res) => {
    res.cookie('studentData', student)
    res.send('Student data added to Cookie');
});

//Display Cookie
app.get('/getstudent', (req,res) => {
    res.send(req.cookies);

});

//Delete Cookie
app.get('/deletestudent', (req,res) => {
    res.clearCookie('studentData');
    res.send('Student data is removed from the cookie');
});
//server listening
app.listen(8082, () => {
    console.log('Server is listening at port 8082');
})
```


Output







Review Question

1. *How to create and use cookies using node.js ?*

Part III : MongoDB

4.23 Introduction to NoSQL

- NoSQL stands for not only SQL.
- It is **nontabular database system** that store data differently than relational tables. There are various types of NoSQL databases such as document, key-value, wide column and graph.
- Using NoSQL we can maintain flexible schemas and these schemas can be scaled easily with large amount of data.

Need :

The NoSQL database technology is usually adopted for following reasons -

1) The NoSQL databases are often used for handling **big data** as a part of fundamental architecture.
2) The NoSQL databases are used for storing and modelling structured, semi-structured and unstructured data.
3) For the efficient execution of database with high availability, NoSQL is used.
4) The NoSQL database is non-relational, so it scales out better than relational databases and these can be designed with web applications.
5) For easy scalability, the NoSQL is used.

Features :

1) The NoSQL does not follow any relational model.
2) It is either schema free or have relaxed schema. That means it does not require specific definition of schema.
3) Multiple NoSQL databases can be executed in distributed fashion.
4) It can process both unstructured and semi-structured data.
5) The NoSQL have higher scalability.
6) It is cost effective.
7) It supports the data in the form of key-value pair, wide columns and graphs.

Review Question

1. *What is NoSQL ? What is the need for it. Enlist various feature of NoSQL.*

4.24 Introduction to MongoDB

- MongoDB is an **open source, document based database.**
- It is developed and supported by a company named 10gen which is now known as **MongoDB Inc.**
- The first ready version of MongoDB was released in March 2010.

Why MongoDB is needed ?

There are so many efficient RDBMS products available in the market, then why do we need MongoDB? Well, all the modern applications require Big data, faster development and flexible deployment. This need is satisfied by the document based database like MongoDB.

Features of MongoDB

1) It is a **schema-less, document based database system.**
2) It provides **high performance** data persistence.
3) It supports **multiple storage engines.**
4) It has a **rich query language support.**
5) MongoDB provides **high availability** and **redundancy** with the help of replication. That means it creates multiple copies of the data and sends these copies to a different server so that if one server fails, then the data is retrieved from another server.
6) MongoDB provides horizontal **scalability** with the help of **sharding.** Sharding means to distribute data on multiple servers.
7) In MongoDB, every field in the document is indexed as primary or secondary. Due to which data can be searched very efficiently from the database.

SQL Structure Vs. MongoDB

Following figure shows the terms in SQL are treated differently in MongoDB. In MongoDB the data is not stored in tables, instead of that, the there is a concept called **collection** which is analogous to the **tables.** In the same manner the **rows** in RDBMS are called **documents** in MongoDB, likewise the **columns** of the record in RDBMS are called **fields.**

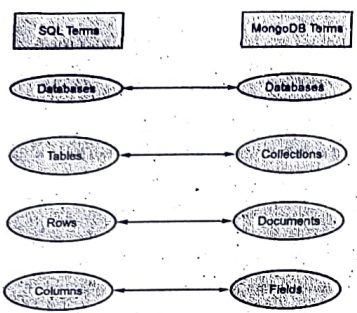
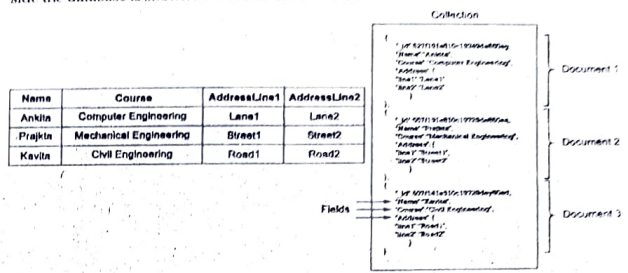


Fig. 4.24.1

Consider a student database as follows –

To the left hand side we show the database in the form of table and to the right hand side the database is shown in the form of collection.



Name	Course	AddressLine1	AddressLine2
Ankita	Computer Engineering	Lane1	Lane2
Prajkta	Mechanical Engineering	Street1	Street2
Kavita	Civil Engineering	Road1	Road2

Fig. 4.24.2

Review Question

1. *List and explain various features of MongoDB.*

4.25 Data Types in MongoDB

Following are various types of data types supported by MongoDB.

1) **Integer :** This data type is used for storing the numerical value.
2) **Boolean :** This data type is used for implementing the Boolean values i.e. true or false.
3) **Double :** Double is used for storing floating point data.
4) **String :** This is the most commonly used data type used for storing the string values.
5) **Min/Max keys :** This data type is used to compare a value against the lowest or highest BSON element.
6) **Arrays :** For storing an array or list of multiple values in one key, this data type is used.

7) **Object :** The object is implemented for embedded documents.

8) **Symbol :** This data type is similar to string data type. This data type is used to store specific symbol type.

9) **Null :** For storing the null values this data type is used.

10) **Date :** This data type is used to store current date or time. We can also create our own date or time object.

11) **Binary data :** In order to store binary data we need to use this data type.

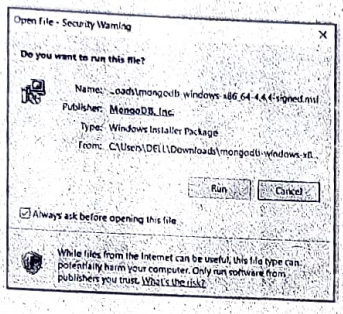
12) **Regular expression :** This data type is used to store regular expression.

4.26 MongoDB Installation

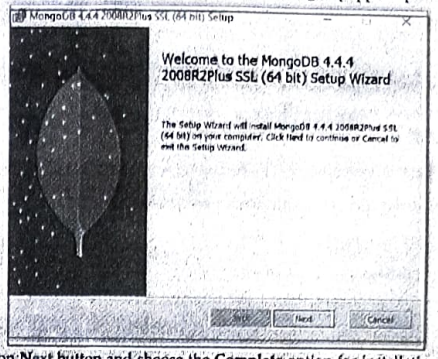
For installing the MongoDB go to the web site

https://www.mongodb.com/try/download/community

Choose **Software->Community Server.** The executable file gets downloaded. Click on **Run** to execute it.

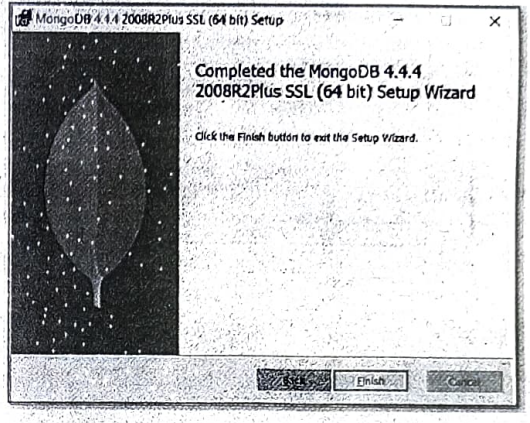


When the installation process starts following window gets popped up



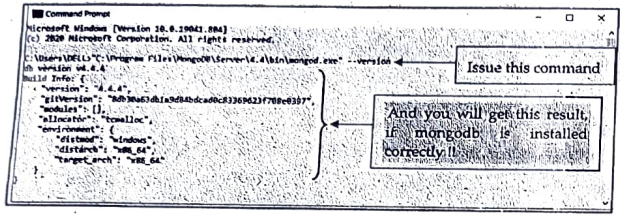
Just click on **Next** button and choose the **Complete** option for installation. Follow the normal procedure of installation by clicking the Next button.

Finally you will get following window on successful installation.

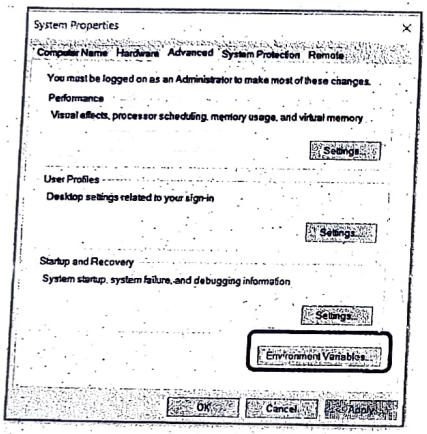


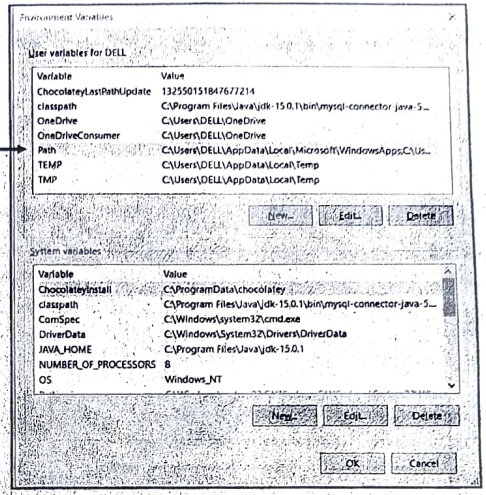
Click on Finish button, to complete the installation process.

In order to verify whether it is installed correctly or not, go to command prompt window and execute the command for mogod.exe file's version. It is illustrated as follows –

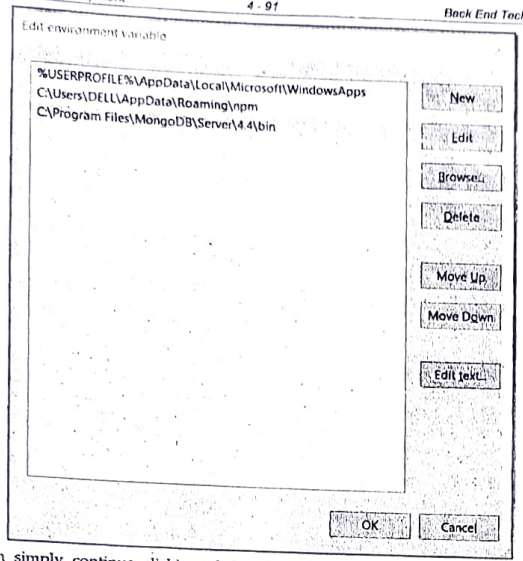


To set the environment variable, Open System Properties and click on **Environment Variables**





Then click on the **path** variable and set the path of MongoDB by clicking **New** button.

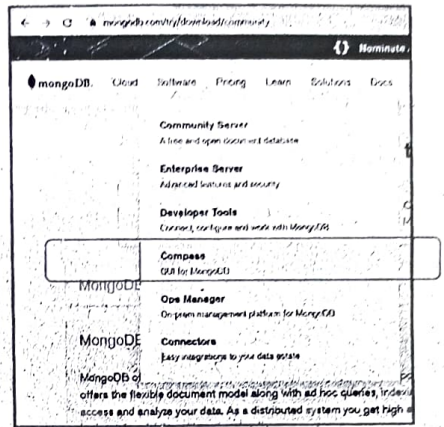


Then simply continue clicking ok button and just come out of the environment variable window.

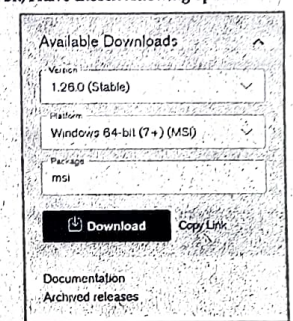
Restart your command prompt window and now simply issue the command **mongod** and then **mongo** at the command prompt window. It will recognise this command and > prompt will appear

Installing Mongodb Compass (Graphical Tool)

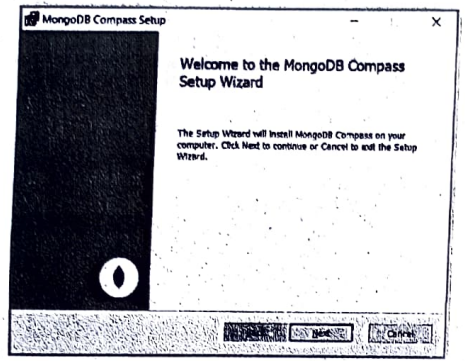
This tool is very useful for handling MongoDB database with the help of simple graphical user interface.



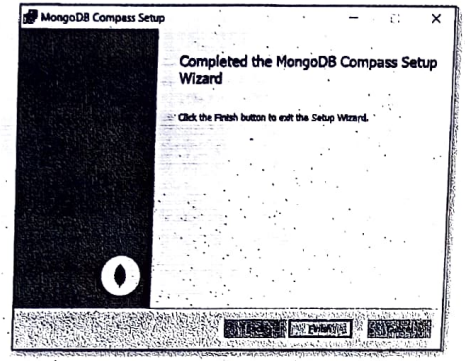
Select any suitable version as per your operating system. As mine is a Windows operating system of 64 bit, I have chosen following option.



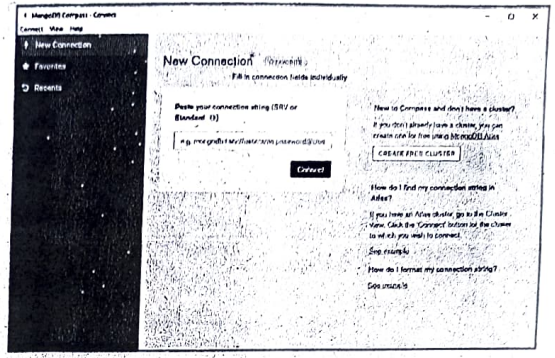
Click on the Download button. The exe file will get downloaded. Double click the installer file which is downloaded in your PC and the installation process for MongoDB Compass will start.



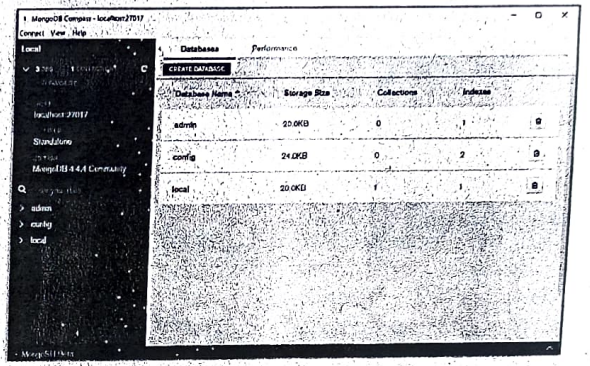
Simply go on clicking Next button, and then click on the install button on the subsequent window. Finally you will get the installation completion screen.



Just click Finish button. Just click the Start button of Windows, locate MongoDB Compass Application and simply click it to start the GUI for Mongo DB. You will get following GUI



If we click the connect button then we get following screen



4.27 Database Commands

In this section we will discuss how to create and handle database in MongoDB using various commands.

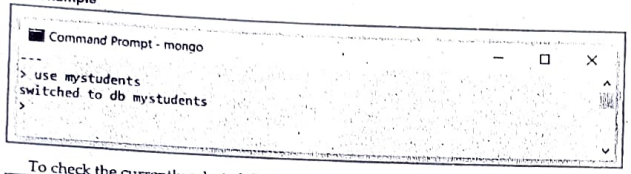
(1) Create Database

Open the command prompt and type the command **mongo** for starting the mongoDB. The > prompt will appear. For creating a database we need to use the "use" command.

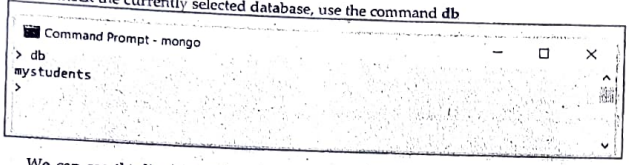
Syntax

```
use Database_name
```

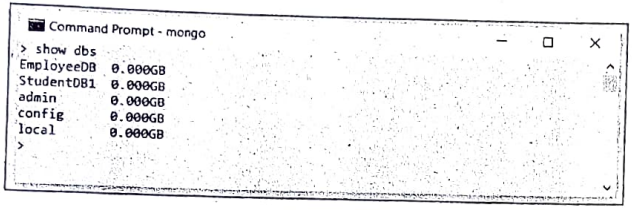
For example



To check the currently selected database, use the command **db**



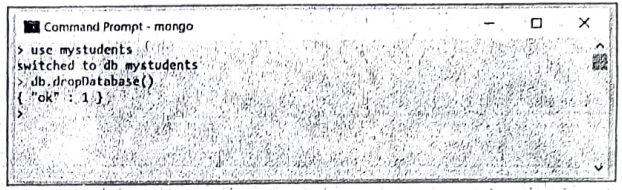
We can see the list of databases present in the MongoDB using the command **show dbs**



Note that in above listing we can not see the **mystudents** database. This is because we have not inserted any **document** into it. To get the name of the database in the listing by means of **show** command, there should be some record present in the database.

(2) Drop Database

The **dropDatabase()** command is used to delete the database. For example



(3) Create Collection

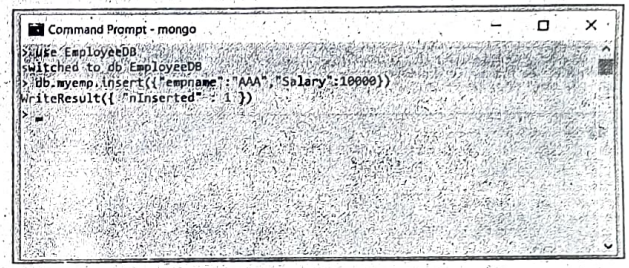
There are two approaches of creating a collection

Method 1 : We can create a collection directly when we insert a document.

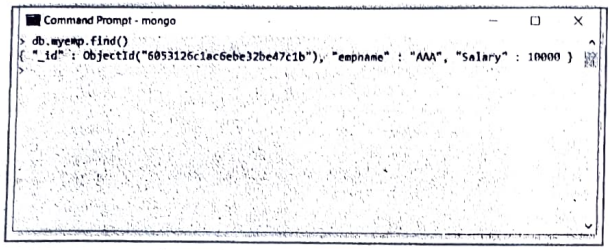
Syntax

```
db.collection_name.insert({key1:value1,key2:value2})
```

For example -



We can cross-verify whether the collection is created or not by using following command



Note that the one document(analogous to row) is getting inserted in the collection named **myemp**.

Method 2 : We can create **collection** explicitly using **createCollection** command.

Syntax

```
db.createCollection(name,options)
```

where

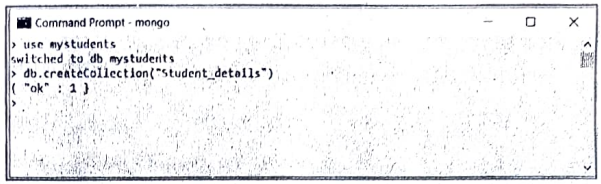
name is the name of collection

options is an optional field. This field is used to specify some parameters such as size, maximum number of documents and so on.

Following is a list of such options.

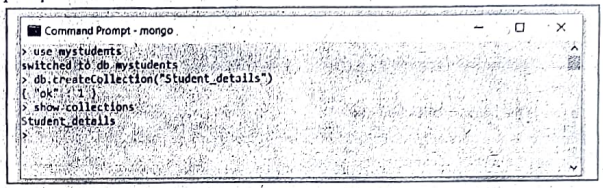
Field	Type	Description
capped	Boolean	Capped collection is a fixed size collection. It automatically overwrites the oldest entries when it reaches to maximum size. If it is set to true, enabled a capped collection. When you specify this value as true, you need to specify the size parameter.
autoIndexID	Boolean	This field is required to create index id automatically. Its default value is false.
size	Number	This value indicates the maximum size in bytes for a capped collection.
Max	Number	It specifies the maximum number of documents allowed in capped collection.

For example – Following command shows how to create collection in a database using explicit command



(4) Display Collection

To check the created collection use the command "show collections" at the command prompt



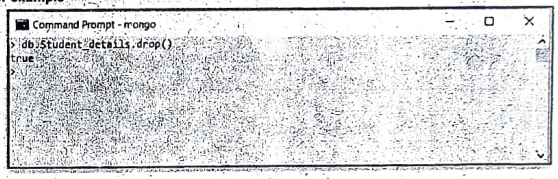
(5) Drop Collection

The drop collection command is actually used to remove the collection completely from the database. The drop command removes a collection completely from database.

Syntax

```
db.collection_name.drop()
```

For example



We can verify the deletion of the collection by using "show collections" in the database.

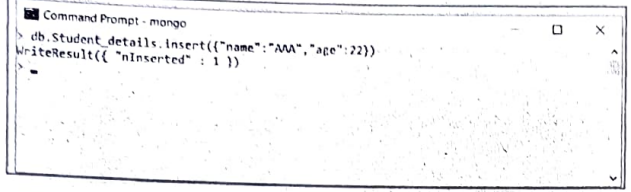
(6) Insert Documents

The document is inserted within the collection. The document is analogous to rows in database.

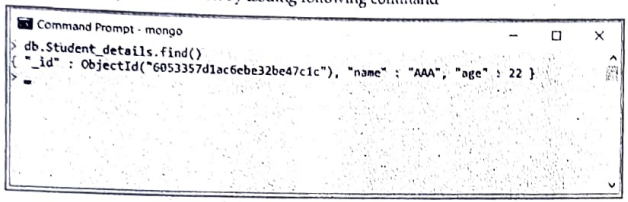
Syntax

```
db.collection_name.insert({key,value})
```

For example

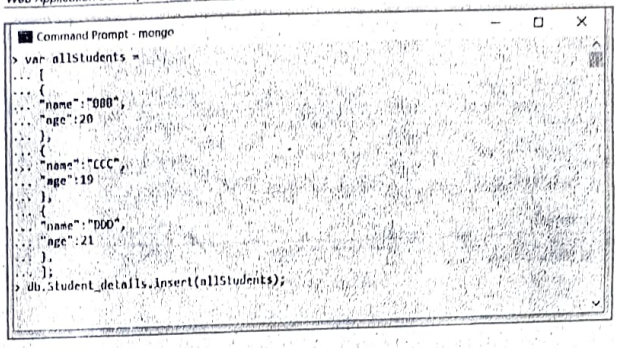


We can verify this insertion by issuing following command

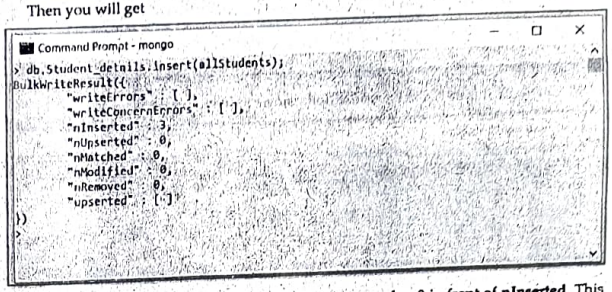


Inserting Multiple Documents

It is possible to insert multiple documents at a time using a single command. Following screenshot shows how to insert multiple documents in the existing collection.

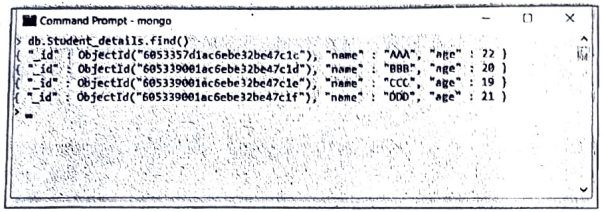


Then you will get



In above screenshot, as you can see that it shows number 3 in front of **nInserted**. This means that the 3 documents have been inserted by this command.

To verify the existence of these documents in the collection you can use **find** command as follows –



(7) Delete Documents

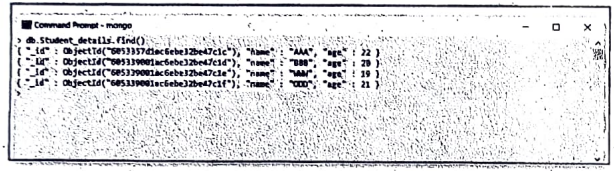
For deleting the document the **remove** command is used. This is the simplest command.

Syntax

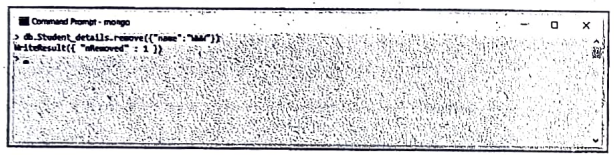
```
db.collection_name.remove(delete_criteria)
```

For example –

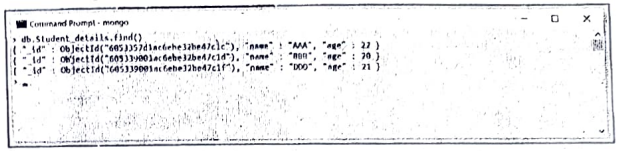
First of all we can find out the documents present in the collection using **find()** command



Now to delete a record with name "WWW" we can issue the command as follows –



Now using **find()** command we can verify if the desired data is deleted or not.



Deleting only one Document

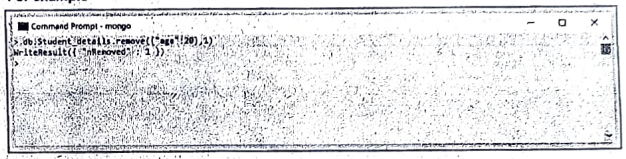
Sometimes the delete criteria matches for more than one records and in such situation, we can forcefully tell the MongoDB to delete only one document.

Syntax

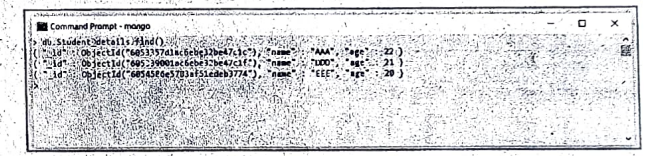
```
db.collection_name.remove(delete_criteria,justOne)
```

The **justOne** is a Boolean value it can be 1 or 0. If we pass this value as 1 then only one document will get deleted.

For example



Now it can be verified using **find()** command as follows –



Note that there were two records that were matching with age = 20 with name "BBB" and "EEE", but since we have passed **justOne** attribute as 1, we get the result by deleting the single record having name "BBB"

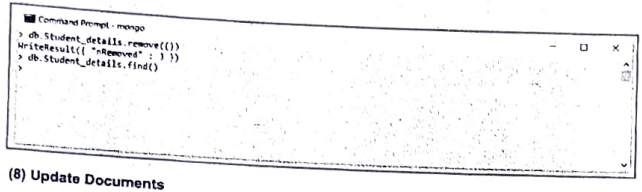
Remove all the documents

It is possible to remove all the documents present in the collection with the help of single command.

Syntax

```
db.collection_name.remove({})
```

For example



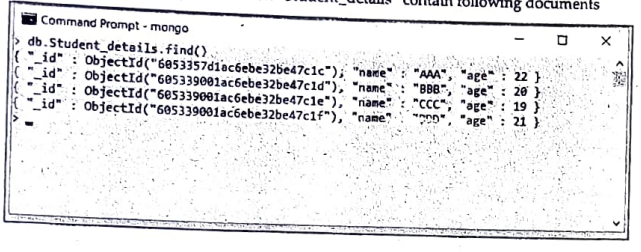
(8) Update Documents

For updating the document we have to provide some criteria based on which the document can be updated.

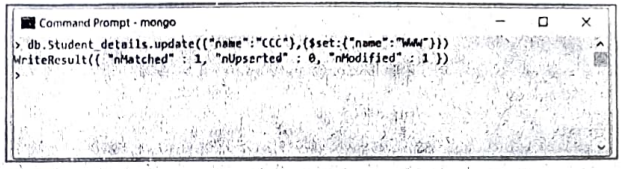
Syntax

```
db.collection_name.update(criteria,update_data)
```

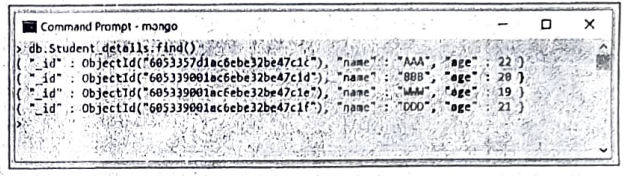
For example – Suppose the collection "Student_details" contain following documents



And we want to change the name "CCC" to "WWW", then the command can be issued as



This can be verified as



Thus the document gets updated.

By default the **update** command updates a single document. But we can update multiple documents as well. For that purpose we have to add {multi:true}

For example

```
db.Student_details.update({age:21},{$set:{age:22}},{multi:true})
```

(9) Sorting

We can use the **sort()** method for arranging the documents in ascending or descending order based on particular field of document.

Syntax

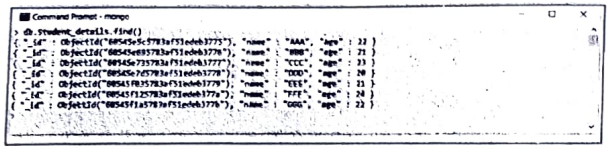
For displaying the documents in **ascending order** we should **pass value 1**

```
db.collection_name.find().sort({field_name:1})
```

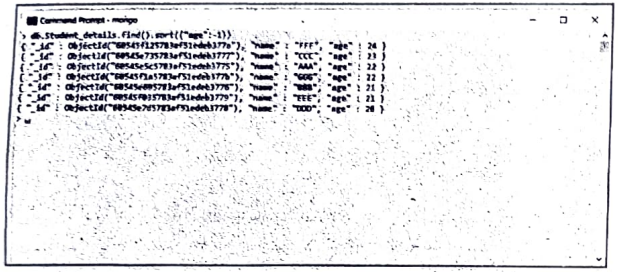
If we pass **-1** then the document will be displayed in the **descending order** of the field.

For example

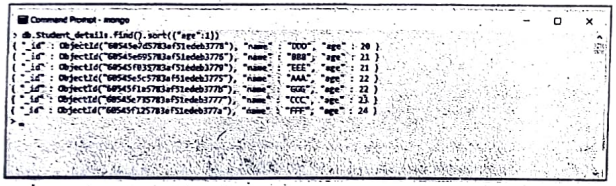
Suppose the collection contains following documents



Now to sort the data in descending order



If we want to display data in ascending order we issue following command



4.28 MongoDB-Node.JS Communication

For connecting Node.js with MongoDB we need **MongoClient**. This can be created using following code.

Connect.js

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
//create database
MongoClient.connect(url, function(err, db) {
if (err)
        throw err;
var dbo = db.db("studentDB");
    console.log("Connected with Database");
});
```

Program Explanation : In above code,

1) First of all, we have to import the module 'mongodb' using **require.** Thus we are creating MongoDB client.
2) Then specify the URL for MongoDB by means of hostname(localhost) and port number on which MongoDB is running (27017). This is a path at which we are going to create a database.
3) Then we are using **connect** method by passing the above URL. Inside this function a database object is created by a database name "studentDB".

Note that before execution of the above code, it is necessary to install the **mongodb** package using the following command at the current working directory.

```
Prompt: > npm install mongodb
```

Review Question

1. *List and explain various steps for MongoDB- node.js communication.*

4.29 CRUD Operations using Node.JS

The CRUD operations are performed in collaboration with Node.js and MongoDB. The CRUD stands for Create, Read, Update and Delete operations.

In this section we will discuss how to perform these operations on the database with the help of simple demo applications.

Prerequisite : For performing these operations we need to have MongoDB installed in our machine. One must also install MongoDB Compass (It is a graphical tool) to verify the operations performed by node.js on MongoDB.

Example Code : In this example, we will create a Student database with two fields – name and city.

Creation of Database

Step 1 : We will create a database studentDB and collection name as Student_Info. For that purpose make a folder in your current working directory. I have created the folder by a name **StudentDBExample**.

Step 2 : Open the command prompt go to the path of **StudentDBExample**. And issue the commands

```
D:\NodeJSExamples\StudentDBEXample>npm init
```

And press enter key to accept the default values. By this **package.json** file gets created.

Step 3 : Then install the **mongodb** module using following command

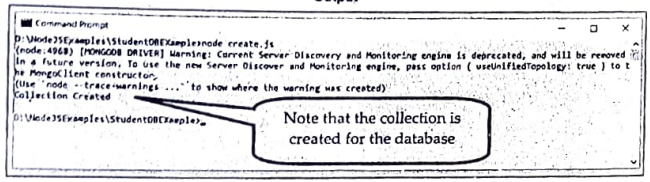
```
D:\NodeJSExamples\StudentDBEXample>npm install mongodb
```

Step 4 : Now create a node.js file for creating a database and collection. The code is as follows –

create.js

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
//create database
MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("studentDB");
 dbo.createCollection("Student_Info",function(err,res){
   if(err)
       throw err;
   console.log("Collection Created")
   });
   db.close();
});
```

Output



Program Explanation : In above code,

1) First of all, we have to import the module 'mongodb' using **require**. Thus we are creating MongoDB client.
2) Then specify the URL for MongoDB by means of hostname (localhost) and port number on which MongoDB is running (27017). This is a path at which we are going to create a database.
3) Then we are using **connect** method by passing the above URL. Inside this function a database object is created by a database name "studentDB".
4) Then using the method **createCollection** for database object we can create a collection inside the database "studentDB". The first parameter passed to this method is name of the collection. Note that here the collection name is "Student_Info".
5) The message "Collection Created" is displayed on the console using **console.log** method.

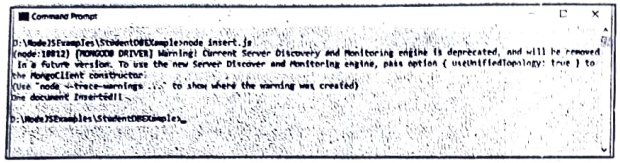
Insertion of Data

We can insert one document or multiple document at a time. First of all we will see the code for inserting one document inside the above created collection.

Insert.js

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
//Insert
MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("studentDB");
 var mydata = { name: 'AAA', city: 'Pune' };
 dbo.collection("Student_Info").insertOne(mydata, function(err, res) {
 if(err)
    throw err;
       console.log("One document Inserted!");
       db.close();
    });
});
```

Output



Program Explanation : In above code,

6) First of all, we have to import the module 'mongodb' using **require**. Thus we are creating MongoDB client.
7) Then specify the URL for MongoDB by means of hostname(localhost) and port number on which MongoDB is running(27017). This is a path at which our **database exists.**
8) Then we are using **connect** method by passing the above URL. Inside this function a database object is created by a database name "studentDB". Thus now we can access to **studentDB**
9) Then a JSON object is in the form {name:value). We insert data in the collection in the form of document. This document is created in the form of JSON. Hence the **mydata** object is created with some values.
10) This **mydata** is inserted into the collection using **insertOne** method.

In the same manner we can insert multiple documents at a time. Following code illustrates it.

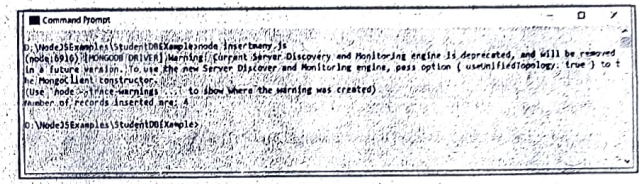
insertmany.js

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
//insert
MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("studentDB");
 var mydata = [
        { name: 'BBB', city: 'Mumbai' },
        { name: 'CCC', city: 'Chennai' },
        { name: 'DDD', city: 'Delhi' },
```



```
        { name: 'EEE', city: 'Ahmedabad' }
];
dbo.collection("Student_Info").insertMany(mydata,function(err,res){
if(err)
    throw err;
    console.log("Number of records inserted are: "+res.insertedCount);
    db.close();
    });
});
```

Output



Program Explanation : In above code

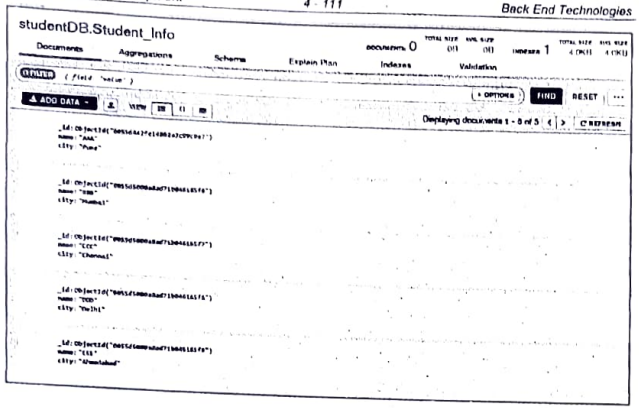
We have created array of values in JSON object as Follows –

```
var mydata = [
        { name: 'BBB', city: 'Mumbai' },
        { name: 'CCC', city: 'Chennai' },
        { name: 'DDD', city: 'Delhi' },
        { name: 'EEE', city: 'Ahmedabad' }
];
```

Then using the **insertMany** command we can insert all the above documents at a time

```
dbo.collection("Student_Info").insertMany(mydata,function(err,res)
```

This program execution can be verified in MongoDB compass.



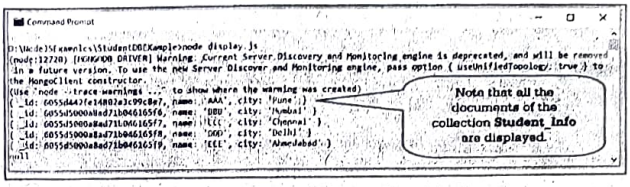
Read Data

We can read all the documents of the collection using **find** method. Following is a simple Node.js code that shows how to read the contents of the database

display.js

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
//Read
MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("studentDB");
 var cursor = dbo.collection("Student_Info").find({})
 cursor.each(function(err,doc){
    console.log(doc);
 db.close();
 });
});
```


Output



Program Explanation : In above code,

11) First of all, we have to import the module 'mongodb' using **require**. Thus we are creating MongoDB client.

12) Then specify the URL for MongoDB by means of hostname(localhost) and port number on which MongoDB is running(27017). This is a path at which our database exists.

13) Then we are using **connect** method by passing the above URL. Inside this function a database object is created by a database name "studentDB". Thus now we can access to **studentDB.**

14) Then for reading the contents present in the documents of the collection we use the find() method. The cursor is created for reading each record(document) of the collection.

```
var cursor = dbo.collection("Student_Info").find({})
```

Then using cursor.each function, each document is displayed within a callback function.

Updating Data

We can change one or more fields of the document using **updateOne** method.

update.js

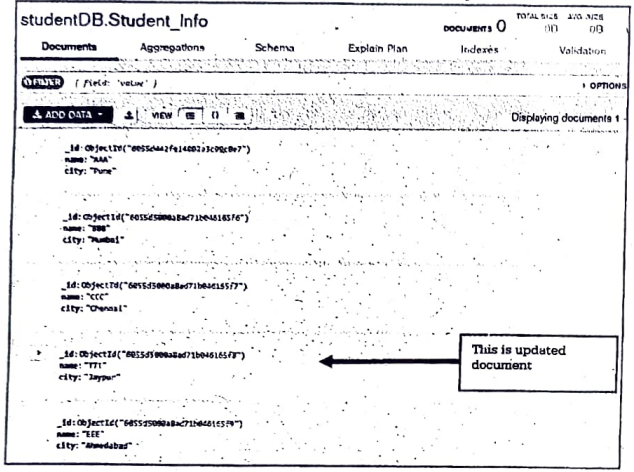
```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
//Update
MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("studentDB");
 var mydata = { name: 'DDD' };
 var newvalues = { $set: {name: 'FFF', city: 'Jaypur' } };
```

```
dbo.collection('Student_Info').updateOne(mydata,newdata,function(err,res){
 if(err)
    throw err;
      console.log("One document Updated!!");
      db.close();
    });
});
```

Output

```
Command Prompt

D:\NodeJSExamples\StudentDBExample>node update.js
(node:12976) [MONGODB DRIVER] Warning: Current Server Discovery and Monitoring engine is deprecated, and will be removed
 in a future version. To use the new Server Discover and Monitoring engine, pass option { useUnifiedTopology: true } to
the MongoClient constructor.
(Use `node --trace-warnings ...` to show where the warning was created)
One document Updated!!

D:\NodeJSExamples\StudentDBExample>
```

Again this updation can be verified by opening MongoDB compass



Program Explanation : In above code,

We are updating one document using **updateOne** method. This method require two parameter first one is the existing data, and second one is the new data which you want to put in place of the existing one. This can be done with the help of following lines of code

```
var mydata = { name: "DDD" }  //existing record with name = "DDD"
var newdata = {$set: {name:"TTT",city:'Jaypur' }} //replace it by a new data
```

Then **updateOne** method is called by passing these two parameters.

Deleting Data

This is the operation in which we are simply deleting the desired document. Here to delete a single record we have used **deleteOne** method. The code is as follows –

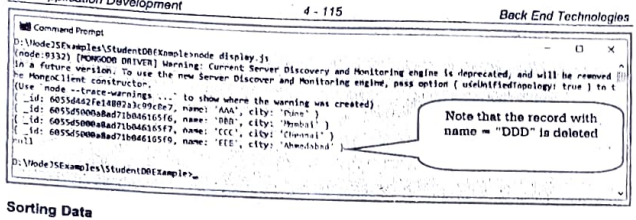
delete.js

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
//Delete
MongoClient.connect(url, function(err, db) {
 if(err) throw err;
 var dbo = db.db("studentDB");
 var mydata = { name: "TTT" };
 dbo.collection('Student_Info').deleteOne(mydata,function(err,res){
 if(err)
    throw err;
      console.log("One document Deleted!!");
      db.close();
    });
});
```

Output

```
Command Prompt

D:\NodeJSExamples\StudentDBExample>node delete.js
(node:11852) [MONGODB DRIVER] Warning: Current Server Discovery and Monitoring engine is deprecated, and will be removed
 in a future version. To use the new Server Discover and Monitoring engine, pass option { useUnifiedTopology: true } to
the MongoClient constructor.
(Use `node --trace-warnings ...` to show where the warning was created)
One document Deleted!!

D:\NodeJSExamples\StudentDBExample>
```

The above deletion operation can be verified by displaying the contents of the collection



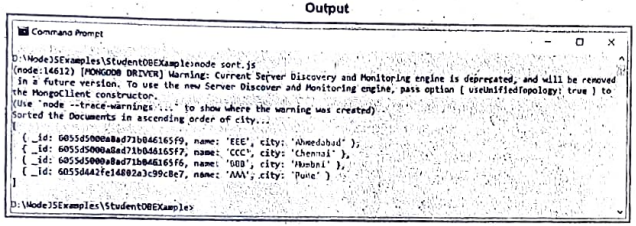
Sorting Data

Data can be arranged in sorted order in the collection. For displaying the sorted ordered data we use the method **sort**. Following code illustrates how to display the data in ascending order of the filed named **city**.

sort.js

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
//sort in ascending order
MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("studentDB");
 var mydata = { city: 1};
 dbo.collection("Student_Info").find().sort(mydata).toArray(function(err,res){
 if(err)
    throw err;
       console.log("Sorted the Documents in ascending order of city...");
   console.log(res);
       db.close();
    });
});
```

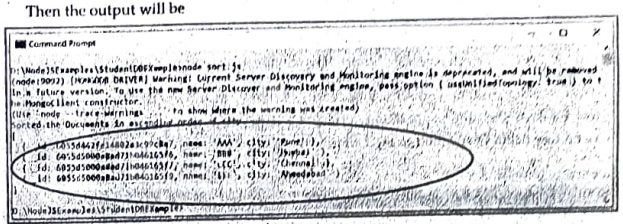
Output



Program Explanation : In above code to sort in ascending order we use **sort**. The key field based on which the document is to be sorted is **city**. We have passed 1 along with it for displaying the city names in ascending order. If we pass -1 then the cities are displayed in descending order. Simply change the above code as

```
var mydata = { city: -1};
```

Then the output will be



Review Questions

1. *Explain insertion and deletion database operation using NodeJs.*
2. *Write a program to display the sorted data in the database using NodeJs.*
3. *What is CRUD ? Explain the CRUD operations in node.js.*

4.30 Mongoose ODM for Middleware

- ODM stands for Object Data Modelling.
- Mongoose is a MongoDB Object Data Modelling (ODM) tool or library designed to work in an asynchronous environment with node.js.
- It allows us to connect to the MongoDB database.
- Besides the data modelling in Node.JS Mongoose also provides a layer of CRUD features on top of MongoDB.
- It manages relationships between data, provides schema validation and is used to translate between objects in code and the representation of those objects in MongoDB.

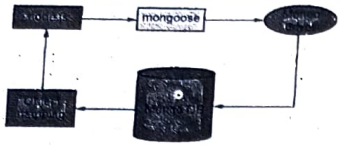


Fig. 4.30.1 Communication between node.js and MongoDB using mongoose

Following are the steps to be followed for using mongoose :

Step 1 : For installing mongoose ,issue the following command at the command prompt

```
$ npm install mongoose
```

Step 2 : For using the mongoose in our node.js app we write the code

```
const mongoose = require("mongoose")
```

Step 3 : Connect to Database : Now we should connect to MongoDB

```
mongoose.connect("mongodb://localhost:27017/studDB");
```

Note that the database is **studDB** to which we get connected.

Step 4 : Create a Schema : We can create a schema as follows –

```
const studSchema = new mongoose.Schema({
  rollno: Number,
  name: String
});
```

Step 5 : Create Collection : Now that we have a schema, we can create a collection. This schema will be the blueprint of the documents into that collection. We will use **mongoose.model( )** method in order to create the **students** collection.

```
const Student = mongoose.model("Student", studentSchema);
```

Step 6 : Create Document : We can create a new document as follows

```
const student = new Student({
  rollno: 101,
  name: 'Ankita'
});
```

Step 7 : We can save this document as

```
student.save();
```


Review Question

1. *Write short note on - Mongoose ODM.*

4.31 Advanced MongoDB

Following are the features of advanced MongoDB.

4.31.1 Indexing

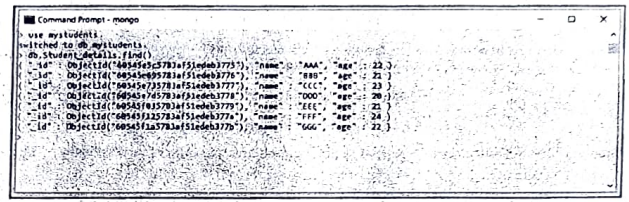
For efficient execution of queries the indexing is used. If we use the query without indexes then the execution of that query will be very slow.

Definition of Index : Index is a special data structure that store small part of collection's data in such a way that we can use it in querying.

The index store the values of index fields outside the table or collection and keep track of their location in the disk.

Demonstration :

In order to create an index, we will use the database named **mystudents**. The collection **Student_details** will be used for creating index.



1) Index creation

Syntax for creating index is

```
db.<collection>.createIndex({KEY:1})
```

The key determines the field on the basis of which the index is created. After the colon the direction of the key(1 or -1) is used to indicate ascending or descending order.

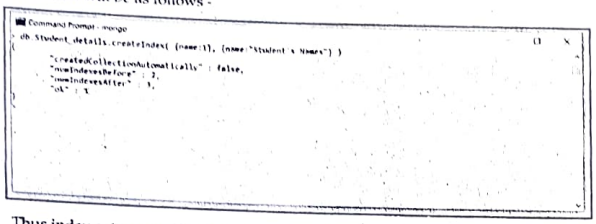
The MongoDB will generate index names by concatenating the indexed keys with the direction of each key with underscore as separator. For example if the index is on the field **name** and the order as 1 then the index will be created as **name_1.**

We can also use name option to define custom index name while creating the index.

For example -

```
> db.Student_details.createIndex( {name:1}, {name:"Student's Names"} )
```

The result will be as follows -



Thus index gets created on the collection **Student_details.**

2) Find Index

We can find all the available indexes in the MongoDB by using **getIndexes** method.

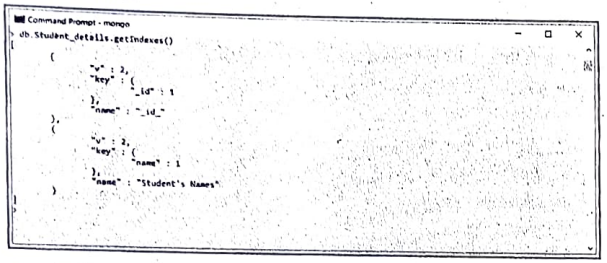
Syntax

```
db.<collection>.getIndexes()
```

For example –

```
>db.Student_details.getIndexes()
```

The result will be



Note that the output contains the default_id index and user created index.

3) Drop Index

To delete an index we use **dropIndex** method.

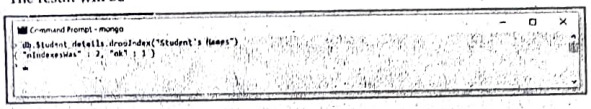
Syntax

```
db.<collection>.dropIndex(Index Name)
```

For example

```
db.Student_details.dropIndex("Student's Names")
```

The result will be



4) Compound Index

We can create index on multiple fields in MongoDB document. Foe example

```
db.Student_details.createIndex( {name:1, age:-1} )
```

It will sort by name in ascending order and age in descending order. If the names are same then the descending order of age can be noticed.

4.31.2 Aggregation

Aggregation is an operation used to process the data that results the computed results. The aggregation groups the data from multiple documents and operate on grouped data to get the combined result. The MongoDB aggregation is equivalent to count(*) and with group by in sql. MongoDB supports the concept of aggregation framework. The typical pipeline of aggregation framework is as follows -

Fig. 4.31.1 Aggregation framework

1) **$match() stage** - filters those documents we need to work with, those that fit our needs
2) **$group() stage** - does the aggregation job
3) **$sort() stage** - sorts the resulting documents the way we require (ascending or descending)

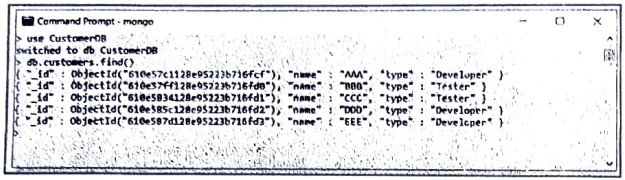
The input of the pipeline can be one or several collections. The pipeline then performs successive transformations on the data and the result is obtained.

Syntax for aggregate operation

```
db.collection.aggregate([operator:{expression}, ...])
```

Demo Example

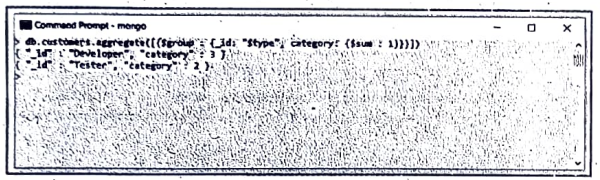
For demonstration purpose, I have created a database named **CustomerDB** inside which there is a collection document named **customers.** Some data is already inserted into it. The contents of **customers** document are as shown below -



Now issue the following command

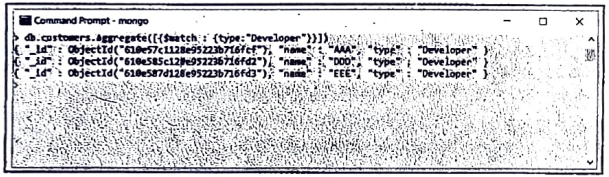
```
> db.customers.aggregate([{$group : {_id: "$type", category: {$sum: 1}}}])
```

The result will be –



The above command will display total categories of the customers - In our database there are two types of customers - "Developer" and "Tester". There are 3 developers and 2 testers in the collection document. The aggregate function is applied on the $group

Similarly if we want to find only **Developers** from the collection document **customers** then we use $match for aggregate function. The demonstration is as follows -



Expressions used by Aggregate function

Expression	Description
$sum	Summates the defined values from all the documents in a collection
$avg	Calculates the average values from all the documents in a collection
$min	Return the minimum of all values of documents in a collection
$max	Return the maximum of all values of documents in a collection
$addToSet	Inserts values to an array but no duplicates in the resulting document
$push	Inserts values to an array in the resulting document
$first	Returns the first document from the source document
$last	Returns the last document from the source document

4.31.3 Map Reduce

Map reduce is a data processing programming model that helps in performing operations on large data sets and produce aggregate results.

Map reduce is used for large volume of data. The syntax for map reduce is

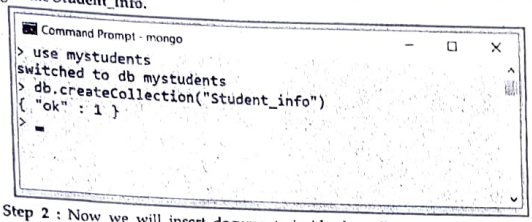
```
>db.collection.mapReduce(
function() {emit(key,value);},  <- map function
function(key,values) {return reduceFunction} ,  <- reduce function
 out: collection,  <- the collection in which the result of mapReduce can be stored
 query: document,
 sort: document,
 limit: number
})
```

Where

1) **map Function :** It uses emit() function in which it takes two parameters key and value key. Here the key is on which we make groups(such as group by name, or age) and the second parameter is on which aggregation is performed like avg(), sum() is calculated on.
2) **reduce Function :** This is a function in which we perform aggregate functions like avg(), sum()
3) **out :** It will specify the collection name where the result will be stored.
4) **query :** We will pass the query to filter the resultset.
5) **sort :** It specifies the optional sort criteria.
6) **limit :** It specifies the optional maximum number of documents to be returned.

Demo Example :

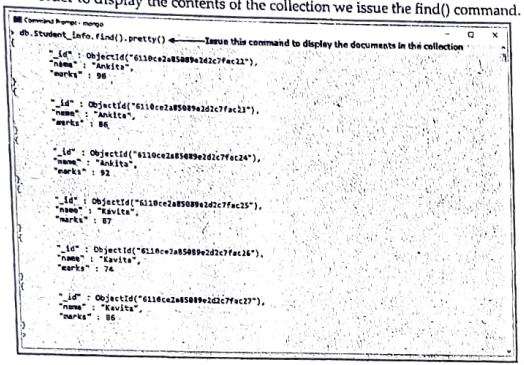
Step 1 : Create a collection inside the database **mystudents**. The collection is created using name **Student_info.**



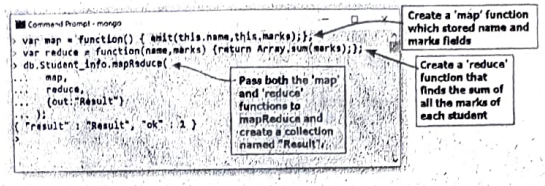
Step 2 : Now we will insert documents inside the collection **Student_info** using following command

```
db.Student_info.insertMany([
{name:"Ankita", marks:96},
{name:"Ankita", marks:86},
{name:"Ankita", marks:92},
{name:"Kavita", marks:87},
{name:"Kavita", marks:74},
{name:"Kavita", marks:86 }
]);
```

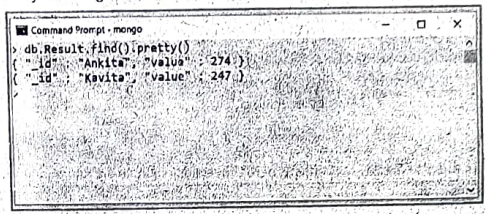
Now in order to display the contents of the collection we issue the find() command.



Step 3 : Now we will apply mapReduce function



Step 4 : The output of the **mapReduce** can be seen with help of **find()** command. It is illustrated by following screenshot -



Advantages of mapReduce

1) MapReduce allows the developer to store complex result in separate collection.
2) MapReduce provides the tools to create incremental aggregation over large collections.
3) It is flexible.

4.31.4 Replication

Replication is process of making data available across multiple data servers

The replication is mainly used for security purpose. In case of sever failure and hardware failure the replication is used to restore the data.

The replication in MongoDB is carried out with the help of **replica sets.**

The replica sets are combination of various MongoDB instances having single primary node and multiple secondary nodes. The secondary node automatically copies the changes made to primary in order to maintain the data across all servers. Refer Fig. 4.31.2.

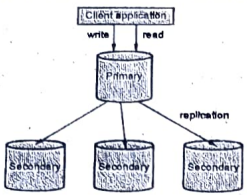


Fig. 4.31.2 Concept of replication

If the primary node gets failed then the secondary node will take primary node's role to provide continuous availability of data. In this case the primary node selection is made by the process called **replica set elections**. In this process the most suitable secondary node will be selected as new primary node.

Benefits of Replication

1) Replication is used for data availability.
2) It is helpful for handling the situations such as hardware failure and server crash.
3) It enhances the performance as data is available across multiple machines and servers.

4.31.5 Sharding

- Sharding is a concept in MongoDB, which splits large data sets into small data sets across multiple MongoDB instances.
- It is not replication of data, but amassing different data from different machines.
- Sharding allows horizontal scaling of data stored in multiple shards or multiple collections. Logically all the shards work as one collection.
- Sharding works by creating a **cluster of MongoDB** instances which is made up of three components.
  - o **Shard :** It is a mongodb instance that contains the sharded data. The combination of multiple shards create a complete data set.

  - o **Router :** This is a mongodb instance which is basically responsible for directing the commands sent by client to appropriate server.
  - o **Config server :** This is mongodb instance which holds the information about various mongodb instances which hold the shard data.

Review Questions

1. *List and explain the features of advanced MongoDB.*
2. *What is replication ? Enlist the advantages of replication in database systems.*
3. *What is the purpose of mapReduce. Explain it with suitable example.*
4. *Explain the concept of sharding.*

4.32 Multiple Choice Questions

Q.1 Node.js is ____ language.

[a] server-side programming [b] client-side programming
[c] database programming [d] none of these

Q.2 The Node.js is _____ by default.

[a] synchronous [b] asynchronous
[c] threaded [d] all of these

Q.3 Node.js is written in _____.

[a] C [b] C++
[c] JavaScript [d] All of the above

Q.4 The extension for NodeJS files is _____.

[a] .node [b] .java
[c] .js [d] .html

Q.5 What is default scope in Node.js application ?

[a] local [b] public
[c] private [d] global

Q.6 npm stands for _____.

[a] node project manager [b] node package manager
[c] new project manager [d] nodejs project manger

Q.7 The command used to check the version of installed nodeJS _____.

[a] node -version [b] npm - -version
[c] npm getVersion [d] None of these