UNIT-3

Node.js: Getting Started with Node.js, Using Events, Listeners, Timers, and Callbacks in Node.js, Handling Data I/O in Node.js. Accessing the File System from Node.js, Implementing HTTP Services in Node.js.

Q) Differentiate React and Node

Feature	React	Node.js
Purpose	Front-end library	Back-end framework
Language	JavaScript	JavaScript (with Node.js runtime)
Architecture	Component-based	Modular
Execution Environment	Browser	Server
Dependency Management	npm	npm or Yarn
Development Ecosystem	Large and active community	Large and active community

Q) Explain the significance of NPM.

Node.js provides a package manager called **NPM (Node Package Manager)** which is a collection of all open-source JavaScript libraries available in this world. It is the world's largest software registry maintained by the Node.js team. It helps in installing any library or module into your machine.

This can be used to install, update, or uninstall any package through NPM.

1. **Installing**: npm install <package_name>[@<version>]

This will create a folder node_modules in the current directory and put all the packages related files inside it. Here @version is optional if you don't specify the version, the latest version of the module will be downloaded.

There are two modes of installation through NPM

i. **Global installation**: If we want to globally install any package or tool add -g to the command. On installing any package globally, that package gets added to the PATH so that we can run it from any location on the computer.

Syntax: npm install -g <package_name>
Eg.
npm install -g express

ii. **Local installation**: If we do not add -g to your command for installation, the modules get installed locally, within a

node_modules folder under the root directory. This is the default mode, as well.
Syntax: npm install <package_name>

Eg. npm install express

Best Practice: Start all projects with npm init. This will create a new package.json for you which allows you to add a bunch of metadata to help others working on the project have the same setup as you.

2. **Update:** We can also update the packages downloaded from the registry to keep the code more secure and stable. Any update for a global package can be done using the following command.

Syntax: npm update -g <package_name>
Eg. npm update express

3. **Uninstall:** uninstall the packages :

We can uninstall the package or module, which we downloaded using the following command.

Syntax: npm uninstall <package_name>[@<version>]
Eg. npm uninstall express@2.1.0

4. **Publishing a module**: It is also possible to publish the custom modules that we created to NPM so that we can make our modules to be available for others to download.

Steps to publish a custom module to NPM:

i. Create a public repository like Github to contain the code for the module.
ii. Create an account at https://npmjs.org/signup.
iii. Use the npm adduser command from a console prompt to add the user you created to the environment.
iv. Type in the username, password, and email that you used to create the account in step ii.
v. Modify the package.json file to include the new repository information and any keywords that you want made available in the registry search
vi. Publish the module using the following command from the application folder in the console: **npm publish**

The package will be published successfully. To use this module from npm, just use the "npm install mypackage" command from the command line and it will get installed.

vii. To remove a package from the registry make sure that you have added a user with rights to the module to the environment using npm adduser and then execute the following command: **npm unpublish <module_name>**

5. **Security:** To perform a quick security check know, we can make use of npm audit which generates a report on the dependencies of your application. This report consists of security threats to your application and can help you fix vulnerabilities by providing npm commands and recommendations for further troubleshooting.
   Syntax: npm audit

Q) What is package.json file? How to create it?

A Node project needs a configuration file named "package.json". It is a file that contains basic information about the project like the package name, version as well as more information like dependencies which specifies the additional packages required for the project.

To create a package.json file, open the Node command prompt and type the below command.

npm init

eg.

```
{
  "name": "my-package",
  "version": "1.0.0",
  "description": "A simple Node.js package for performing basic math operations.",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "node",
    "math",
    "package"
  ],
  "author": "Your Name",
  "license": "MIT"
}
```

Q) Explain about censorify module in node.js

The censored words are replaced with **** and that the new censored word gloomy is added to the censorify module instance censor.

Install the module censorify using: npm install censorify.

```
var censor = require("censorify");
console.log(censor.getCensoredWords());
console.log(censor.censor("Some very sad, bad and mad text."));
censor.addCensoredWord("gloomy");
console.log(censor.getCensoredWords());
console.log(censor.censor("A very gloomy day."))
```

Output:

```
D:\PVP\MWA\Lab\NodeDemo> node Censor.js
[ 'sad', 'bad', 'mad' ]
Some very ****, **** and **** text.
[ 'sad', 'bad', 'mad', 'gloomy' ]
A very **** day.
```

Q) Explain how to create modules in node.js with an example.

A module can contain functions, classes, objects, or any other piece of code that can be shared between different parts of an application.

Node.js has a built-in module system that allows you to create and use modules. A module can be defined in a separate file, and can be loaded into other parts of the application using the require() function.

Eg.

calc.js:

```
exports.add = (a, b) => {
    console.log("Add Result:", a + b);
  };

exports.subtract = (a, b) => {
    console.log("Sub Result:", a - b);
  };
```

demo.js:

```
const myCalculator = require("./calc");
myCalculator.add(1, 2);
myCalculator.subtract(3, 2);
```

Here, the functions are exported using exports object in calc.js and require() function is used in demo.js which loads the module calc.

Output:

```
D:\PVP\MWA\Lab\NodeDemo> node demo.js
Add Result: 3
Sub Result: 1
```

Q) Explain how to write data to console in Node.js

One of the most useful modules in Node.js during the development process is the console module. This module provides a lot of functionality when writing debug and information statements to the console. The console module allows you to control output to the console, implement time delta output, and write tracebacks and assertions to the console.

Function	Description

`log([data],[...])`	Writes data output to the console. The data variable can be a string or an object that can be resolved to a string. Additional parameters can also be sent. For example:
	`console.log("There are %d items", 5);`
	`>>There are 5 items`

`info([data],[...])`	Same as `console.log`.
`error([data], [...])`	Same as `console.log`; however, the output is also sent to `stderr`.
`warn([data],[...])`	Same as `console.error`.

`dir(obj)`	Writes out a string representation of a JavaScript object to the console. For example:
	`console.dir({name:"Brad", role:"Author"});`
	`>> { name: 'Brad', role: 'Author' }`

`time(label)`	Assigns a current timestamp with ms precision to the string `label`.

`timeEnd(label)`	Creates a delta between the current time and the timestamp assigned to `label` and outputs the results. For example:
	`console.time("FileWrite");`
	`f.write(data); //takes about 500ms`
	`console.timeEnd("FileWrite");`
	`>> FileWrite: 500ms`

`trace(label)`	Writes out a stack trace of the current position in code to `stderr`. For example:
	`module.trace("traceMark");`
	`>>Trace: traceMark`
	`  at Object.<anonymous> (C:\test.js:24:9)`
	`  at Module._compile (module.js:456:26)`
	`  at Object.Module._ext.js (module.js:474:10)`
	`  at Module.load (module.js:356:32)`
	`  at Function.Module._load (module.js:312:12)`
	`  at Function.Module.runMain(module.js:497:10)`
	`  at startup (node.js:119:16)`
	`  at node.js:901:3`

`assert(expression, [message])`	Writes the message and stack trace to the console if expression evaluates to `false`.

Q) Explain Event Handling mechanism in Node.js

Node is used to build the back-end of web applications and provides an event-driven, non-blocking I/O model that makes it highly efficient for handling large amounts of data.

The Node.js event model does things differently from traditional event model. Instead of executing all the work for each request on individual threads, work is added to an event queue and then picked up by a single thread running an event loop. The event loop grabs the top item in the event queue, executes it, and then grabs the next item. When executing code that is no longer live or has blocking I/O, instead of calling the function directly, the function is added to the event queue along with a callback that is executed after the function completes. When all events on the Node.js event queue have been executed, the Node application terminates.

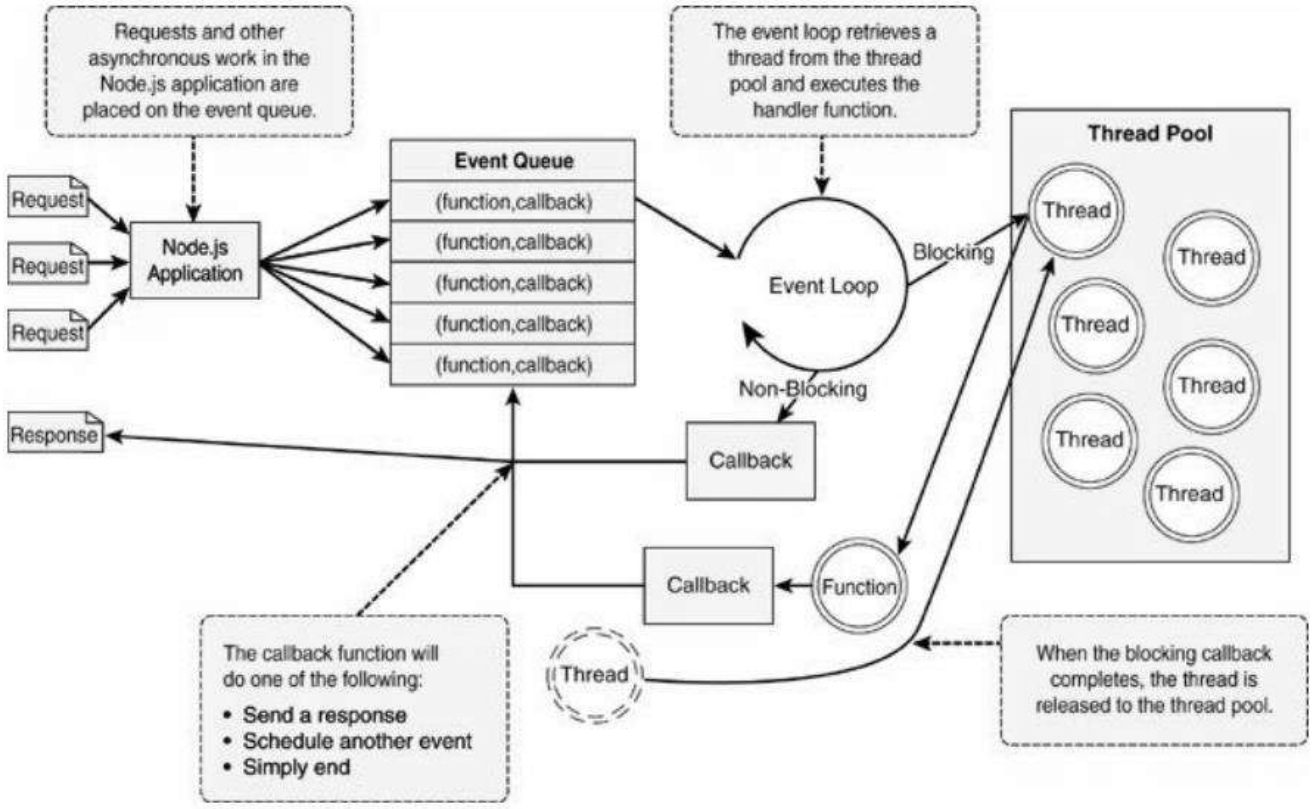

Fig. Event Handling in Node.js

We can then use the event model to schedule work on the event queue. In Node.js applications, work is scheduled on the event queue by passing a callback function using one of these methods:

- Make a call to one of the blocking I/O library calls such as writing to a file or connecting to a database.

- Add a built-in event listener to a built-in event such as an http.request or server.connection.
- Create own event emitters and add custom listeners to them.
- Use the process.nextTick option to schedule work to be picked up on the next cycle of the event loop.
- Use timers to schedule work to be done after a particular amount of time or at periodic intervals.

Eg. Create any custom event as an example.

Q) Explain how to schedule/add work to Event Queue using Timers.

There are three types of timers you can implement in Node.js: timeout, interval, and immediate.

i. Delaying Work with Timeouts

- Timeout timers are used to delay work for a specific amount of time. When that time expires, the callback function is executed and the timer goes away
- Timeout timers are created using the setTimeout(callback, delayMilliSeconds, [args]) method built into Node.js.

When you call setTimeout(), the callback function is executed after delayMilliSecondsexpires.

For example, the following executes myFunc() after 1 second:

setTimeout(myFunc, 1000);

The setTimeout() function returns a timer object ID. You can pass this ID to clearTimeout(timeoutId) at any time before the delayMilliSeconds expires to cancel the timeout function.

Eg. Implementing a series of timeouts at various intervals

Timer1.js

```
function simpleTimeout(consoleTimer){
    console.timeEnd(consoleTimer);
    }

    console.time("twoSecond");
    setTimeout(simpleTimeout, 2000, "twoSecond");
    console.time("oneSecond");
    setTimeout(simpleTimeout, 1000, "oneSecond");
    console.time("fiveSecond");
    setTimeout(simpleTimeout, 5000, "fiveSecond");
    console.time("50MilliSecond");
    setTimeout(simpleTimeout, 50, "50MilliSecond");
```

- The **console.time()** method starts a timer you can use to track how long an operation takes. You give each timer a unique name, and may have up to 10,000 timers running on a given page.
- When you call console.timeEnd() with the same name, the browser will output the time, in milliseconds, that elapsed since the timer was started.

Output:

```
C:\Program Files\nodejs\node.exe .\Timer1.js
50MilliSecond: 50.341064453125 ms
50MilliSecond: 50.751ms
oneSecond: 1015.48388671875 ms
oneSecond: 1.016s
twoSecond: 2014.297119140625 ms
twoSecond: 2.014s
fiveSecond: 5008.2470703125 ms
fiveSecond: 5.009s
```

ii. Performing Periodic Work with Intervals

- Interval timers are used to perform work on a regular delayed interval. When the delay time expires, the callback function is executed and is then rescheduled for the delay interval again.
- Interval timers are created using the setInterval(callback,delayMilliSeconds, [args]) method built into Node.js.
- When you call setInterval(), the callback function is executed every interval after

delayMilliSeconds has expired. For example, the following executes myFunc() every second:

```
setInterval(myFunc, 1000);
```

Eg. Timer2.js

```
var x=0, y=0, z=0;
function displayValues(){
console.log("X=%d; Y=%d; Z=%d", x, y, z);
}

function updateX(){
x += 1;
}
function updateY(){
y += 1;
}
function updateZ(){
z += 1;
displayValues();
}
```

```
setInterval(updateX, 500);
setInterval(updateY, 1000);
setInterval(updateZ, 2000);
```

Output:

```
C:\Program Files\nodejs\node.exe .\Timer2.js
X=3; Y=1; Z=1
X=7; Y=3; Z=2
X=11; Y=5; Z=3
X=15; Y=7; Z=4
X=19; Y=9; Z=5
X=23; Y=11; Z=6
X=27; Y=13; Z=7
```

iii. Performing Immediate Work with an Immediate Timer

- Immediate timers are used to perform work on a function as soon as the I/O event callbacks are executed, but before any timeout or interval events are executed. This allows you to schedule work to be done after the current events in the event queue are completed.
- Immediate timers are created using the setImmediate(callback,[args])method built into Node.js. When you call setImmediate(), the callback function is placed on the event queue and popped off once for each iteration through the eventqueue loop after I/O events have a chance to be called

iv. Using nextTick to Schedule Work

- A useful method of scheduling work on the event queue is the process.nextTick(callback) function. This function schedules work to be run on the next cycle of the event loop. Unlike the setImmediate() method, nextTick() executes before the I/O events are fired.
- This can result in starvation of the I/O events, so Node.js limits the number of nextTick() events that can be executed each cycle through the event queue by the value of process.maxTickDepth, which defaults to 1000.

Eg. <u>Timer3.js</u>

```
var fs = require("fs");
fs.stat("nexttick.js", function(){
 console.log("nexttick.js Exists");
});

setImmediate(function(){
console.log("Immediate Timer 1 Executed");
});
```

```
setImmediate(function(){
 console.log("Immediate Timer 2 Executed");
});

process.nextTick(function(){
console.log("Next Tick 1 Executed");
});

process.nextTick(function(){
console.log("Next Tick 2 Executed");
});
```

Output:

```
C:\Program Files\nodejs\node.exe .\Timer3.js
Next Tick 1 Executed
Next Tick 2 Executed
Immediate Timer 1 Executed
Immediate Timer 2 Executed
nexttick.js Exists
```

v. Dereferencing Timers from the Event Loop

Often we do not want timer event callbacks to continue to be scheduled when they are the only events left in the event queue.
The unref() function available in the object returned by setInterval and setTimeout allows us to notify the event loop to not continue when these are the only events on the queue.

Eg.

```
myInterval = setInterval(myFunc);
myInterval.unref();
```

If for some reason later do not want the program to terminate if the interval function is the only event left on the queue, you can use the ref() function to rereference it: myInterval.ref();

Q) Explain how to create a custom event in Node.js

In Node.js, events are a core concept that allows applications to respond to different types of actions or changes that occur within the application. An event is essentially a signal that something has happened, such as a user clicking a button, a file being read or written, or a network connection being established.

Event listeners are functions that are registered to listen for and respond to specific events. When an event occurs, all registered event listeners for that event are executed in the order they were registered. Event listeners can be added or removed dynamically, and multiple event listeners can be registered for the same event.

Events are emitted using an EventEmitter object. This object is included in the events module. The emit(eventName, [args]) function triggers the eventName event and includes any arguments provided.

The following code snippet shows how to implement a simple event emitter:

```
var events = require('events');
var emitter = new events.EventEmitter();
emitter.emit("simpleEvent");
```

Adding Event Listeners to Objects

- **addListener(eventName, callback)**: Attaches the callback function to the object's listeners. Every time the eventName event is triggered, the callback function is placed in the event queue to be executed.
- **.on(eventName, callback)**: Same as .addListener().
- **.once(eventName, callback)**: Only the first time the eventName event is triggered, the callback function is placed in the event queue to be executed.

Removing Listeners from Objects:

- **.listeners(eventName):** Returns an array of listener functions attached to the eventName event.
- **.setMaxListeners(n):** Triggers a warning if more than n listeners are added to an EventEmitter object. The default is 10.
- **.removeListener(eventName, callback):** Removes the callback function from the eventName event of the EventEmitter object.

Eg. 1:

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

//Create an event handler:
var welcome = function () {
  console.log('Welcome to pvpsit');
}

var bye = function () {
  console.log('Good bye to pvpsit');
}

//Assign the eventhandler to an event:
eventEmitter.on('greet', welcome);
eventEmitter.on('greet', bye);

//Fire the 'greet' event:
eventEmitter.emit('greet');

var listener_count = eventEmitter.listenerCount('greet');
console.log(listener_count + " Listner(s) listening to greet event");
```

```
// Remove the binding of listner bye function
eventEmitter.removeListener('greet', bye);
console.log("listener bye removed..");
```

Output:

node Event_Demo.js

Welcome to pvpsit

Good bye to pvpsit

2 Listner(s) listening to greet event

listener bye removed..

Eg. 2.
Event1.js

```
var events = require('events');
 function Account() {
    this.balance = 0;
    events.EventEmitter.call(this);
    this.deposit = function(amount){
      this.balance += amount;
      this.emit('balanceChanged');
    };

    this.withdraw = function(amount){
      this.balance -= amount;
      this.emit('balanceChanged');
    };
 }

 Account.prototype.__proto__ = events.EventEmitter.prototype;
  function displayBalance(){
   console.log("Account balance: $%d", this.balance);
 }



 function checkOverdraw(){
    if (this.balance < 0){
        console.log("Account overdrawn!!!");
    }
 }

 function checkGoal(acc, goal){
  if (acc.balance > goal){
```

```
        console.log("Goal Achieved!!!");
    }
}
var account = new Account();
account.on("balanceChanged", displayBalance);
account.on("balanceChanged", checkOverdraw);
account.on("balanceChanged", function(){
            checkGoal(this, 1000);
});
account.deposit(220);
account.deposit(320);
account.deposit(600);
account.withdraw(1200);
```

Output:

```
C:\Program Files\nodejs\node.exe .\Event1.js
Account balance: $220
Account balance: $540
Account balance: $1140
Goal Achieved!!!
Account balance: $-60
Account overdrawn!!!
```

Q) What is a callback? Explain different types of callbacks with suitable examples.

Callback: A callback is a function or piece of code that is passed as an argument to another function, with the intention of being called at some point during the execution of that function or method. A callback function can be defined with or without parameters.

Eg.

```
function sum(a, b, callback) {
    let result = a + b;
    callback();
    console.log(result)
  }

  function logResult() {
    console.log('The sum is:');
  }

  sum(2, 3, logResult);
```

Output:

```
node callback0.js
The sum is:
5
```

Callback with parameters:

Eg. 1:

```
function add(a, b, callback) {
    let result = a + b;
    callback(result);
 }

 function logResult(sum) {
   console.log('The sum is %d',sum);
 }

 add(2, 3, logResult);
```

Output:

```
node callback1.0.js
The sum is 5
```

Eg. 2:

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

function CarShow() {
  events.EventEmitter.call(this);
  this.seeCar = function(make){
      this.emit('sawCar', make);
  };
}

CarShow.prototype.__proto__ = events.EventEmitter.prototype;

var show = new CarShow();
function logCar(make){
   console.log("Saw a " + make);
}
function logColorCar(make, color){
   console.log("Saw a %s %s", color, make);
}
show.on("sawCar", logCar);
show.on("sawCar", function(make){
var colors = ['red', 'blue', 'black'];
var color = colors[Math.floor(Math.random()*3)];
    logColorCar(make, color);
});
show.seeCar("Ferrari");
show.seeCar("Porsche");
show.seeCar("Bugatti");
show.seeCar("Lamborghini");
show.seeCar("Aston Martin");
```

Output:

```
node callback1.js
Saw a Ferrari
Saw a black Ferrari
Saw a Porsche
Saw a red Porsche
Saw a Bugatti
Saw a red Bugatti
Saw a Lamborghini
Saw a red Lamborghini
Saw a Aston Martin
Saw a red Aston Martin
```

Closure callback: In Node.js, a closure callback is a function that has access to variables in its parent function scope, even after the parent function has returned. This is achieved through closure, which allows a function to "remember" its lexical scope.

Eg. 1:

```
function counter() {
   let count = 0;

   const incrementCount = function() {
     count++;
     console.log(`Count is now ${count}`);
   };

   return incrementCount;
 }

 const callback = counter();
 callback(); // Output: 'Count is now 1'
 callback(); // Output: 'Count is now 2'
```

Output:

```
node callback2.0.js
Count is now 1
Count is now 2
```

Eg.2:

```
function logCar(logMsg, callback){
   process.nextTick(function() {
   callback(logMsg);
   });
 }
var cars = ["Ferrari", "Porsche", "Bugatti"];
for (var idx in cars){
   var message = "Saw a " + cars[idx];
```

```
    logCar(message, function(){
        console.log("Normal Callback: " + message);
    });
 }
for (var idx in cars){
    var message = "Saw a " + cars[idx];
    (function(msg){
       logCar(msg, function(){
       console.log("Closure Callback: " + msg);
    });
    })(message);
  }
```

Output:

```
node callback2.js
Normal Callback: Saw a Bugatti
Normal Callback: Saw a Bugatti
Normal Callback: Saw a Bugatti
Closure Callback: Saw a Ferrari
Closure Callback: Saw a Porsche
Closure Callback: Saw a Bugatti
```

The second loop also iterates over the cars array, but it uses a closure to pass the log message to the callback function.

Chained Callback: A chained callback is a series of callback functions that are executed one after another in a sequence. The output of one callback function is passed as input to the next callback function in the chain.

```
function add(a, b, callback) {
    let sum = a + b;
    callback(sum);
  }

  function square(num, callback) {
    let result = num * num;
    callback(result);
  }

  function logResult(result) {
    console.log(`The final result is ${result}`);
  }

  add(2, 3, function(sum) {
    square(sum, function(result) {
      logResult(result);
    });
  });
```

Output:
node callback3.0.js
The final result is 25

Eg. 2:

```
function logCar(car, callback){
    console.log("Saw a %s", car);
    if(cars.length){
    process.nextTick(function(){
    callback();
    });
    }
    }
   function logCars(cars){
    var car = cars.pop();
    logCar(car, function(){
    logCars(cars);
    });
    }
    var cars = ["Ferrari", "Porsche", "Bugatti",
    "Lamborghini", "Aston Martin"];
    logCars(cars);
```

Output:
node callback3.js
Saw a Aston Martin
Saw a Lamborghini
Saw a Bugatti
Saw a Porsche
Saw a Ferrari

Q) Explain Buffer class in Node.js with an example.

Node provides Buffer class which provides instances to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap. Buffer class is a global class that can be accessed in an application without importing the buffer module.

```
var buf1 = new Buffer(100);
var buf2 = new Buffer(100);

var buf3 = new Buffer(26);
for (var i = 0 ; i < 26 ; i++) {
 buf3[i] = i + 97;
}
console.log("buffer 3:"+ buf3.toString('ascii'));

len = buf1.write("Welcome to pvpsit");
console.log(" buf1 Octets written : "+ len);
console.log("buffer 1:"+ buf1.toString('utf-8'));
```

```
len = buf2.write("Welcome to it");
console.log("buf2 Octets written : "+ len);
console.log("buffer 2:"+ buf2.toString('utf-8'));

var buf6 = Buffer.concat([buf1,buf2]);
console.log("buffer after concatinating buf1 and buf2: " + buf6.toString());

var buf4 = new Buffer(26)
buf3.copy(buf4);
console.log("buffer4 content: " + buf4.toString());

var buf5 = buf4.slice(0,9);
console.log("buf4.slice(0,9): " + buf5.toString()+" and length is "+buf5.length);

var result = buf5.compare(buf4);
if(result < 0) {
 console.log(buf4 +" comes after " + buf5);
}else if(result == 0){
 console.log(buf4 +" is same as " + buf5);
}else {
 console.log(buf4 +" comes before " + buf5);
}

var json = buf5.toJSON(buf5);
console.log(json);
```

O/P:

```
PS D:\PVP\MWA\Lab\Files> node Event_Demo.js
Welcome to pvpsit
Good bye to pvpsit
2 Listner(s) listening to greet event
listener bye removed..
PS D:\PVP\MWA\Lab\Files>
 *  History restored

buffer 3:abcdefghijklmnopqrstuvwxyz
 buf1 Octets written : 17
buffer 1:Welcome to pvpsit
buf2 Octets written : 13
buffer 2:Welcome to it
buffer after concatinating buf1 and buf2: Welcome to pvpsitWelcome to it
buffer4 content: abcdefghijklmnopqrstuvwxyz
buf4.slice(0,9): abcdefghi and length is 9
abcdefghijklmnopqrstuvwxyz comes after abcdefghi
{
  type: 'Buffer',
  data: [
     97,  98,  99, 100,
```

```
    101, 102, 103, 104,
    105
  ]
}
```

Writing to Buffers Syntax Following is the syntax of the method to write into a Node Buffer: buf.write(string[, offset][, length][, encoding])
Parameters Here is the description of the parameters used: • string - This is the string data to be written to buffer.
• offset - This is the index of the buffer to start writing at. Default value is 0.
• length - This is the number of bytes to write. Defaults to buffer.length.
• encoding - Encoding to use. 'utf8' is the default encoding.
Return Value This method returns the number of octets written. If there is not enough space in the buffer to fit the entire string, it will write a part of the string.

Reading from Buffers Syntax Following is the syntax of the method to read data from a Node Buffer: buf.toString([encoding][, start][, end])
Parameters Here is the description of the parameters used:
• encoding - Encoding to use. 'utf8' is the default encoding.
• start - Beginning index to start reading, defaults to 0.
• end - End index to end reading, defaults is complete buffer.
Return Value This method decodes and returns a string from buffer data encoded using the specified character set encoding.
Convert Bufferto JSON Syntax Following is the syntax of the method to convert a Node Buffer into JSON object:
buf.toJSON()
Return Value This method returns a JSON-representation of the Buffer instance.

Concatenate Buffers Syntax Following is the syntax of the method to concatenate Node buffers to a single Node Buffer:
Buffer.concat(list[, totalLength])
Parameters Here is the description of the parameters used:
• list - Array List of Buffer objects to be concatenated.
• totalLength - This is the total length of the buffers when concatenated.
Return Value This method returns a Buffer instance.

Compare Buffers Syntax Following is the syntax of the method to compare two Node buffers: buf.compare(otherBuffer);
Parameters Here is the description of the parameters used:
• otherBuffer - This is the other buffer which will be compared with buf.
Return Value Returns a number indicating whether it comes before or after or is the same as the otherBuffer in sort order.

Copy Buffer Syntax Following is the syntax of the method to copy a node buffer: buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])
Parameters Here is the description of the parameters used:

• targetBuffer - Buffer object where buffer will be copied.
• targetStart - Number, Optional, Default: 0
• sourceStart - Number, Optional, Default: 0
• sourceEnd - Number, Optional, Default: buffer.length
Return Value No return value.

Slice Buffer Syntax Following is the syntax of the method to get a sub-buffer of a node buffer: buf.slice([start][, end])
Parameters Here is the description of the parameters used:
• start - Number, Optional, Default: 0
• end - Number, Optional, Default: buffer.length
Return Value Returns a new buffer which references the same memory as the old one

Buffer Length Syntax Following is the syntax of the method to get a size of a node buffer in bytes: buf.length;
Return Value Returns the size of a buffer in bytes.

Q) What are streams? Explain different types of streams with suitable examples.

Streams are objects that let you read data from a source or write data to a destination in continuous fashion. In Node.js, there are four types of streams:
• Readable - Stream which is used for read operation.
• Writable - Stream which is used for write operation.
• Duplex - Stream which can be used for both read and write operation.

Each type of Stream is an EventEmitter instance and throws several events at different instance of times. For example, some of the commonly used events are:
• data - This event is fired when there is data is available to read.
• end - This event is fired when there is no more data to read.
• error - This event is fired when there is any error receiving or writing data.
• finish - This event is fired when all the data has been flushed to underlying system.

Eg. Read Stream

```
var fs = require('fs');
 var options = { encoding: 'utf8', flag: 'r' };
 var fileReadStream = fs.createReadStream("grains.txt", options);
 fileReadStream.on('data', function(chunk) {
 console.log('Grains: %s', chunk);
 console.log('Read %d bytes of data.', chunk.length);
 });
 fileReadStream.on("close", function(){
 console.log("File Closed.");
});
```

Eg. Write Stream

```
var fs = require('fs');
var grains = ['wheat', 'rice', 'oats'];

var options = { encoding: 'utf8', flag: 'w' };
var fileWriteStream = fs.createWriteStream("grains.txt", options);

fileWriteStream.on("close", function(){
    console.log("File Closed.");
});

while (grains.length){
    var data = grains.pop() + " ";
    fileWriteStream.write(data);
    console.log("Wrote: %s", data);
}
fileWriteStream.end();

var options = { encoding: 'utf8', flag: 'r' };
var fileReadStream = fs.createReadStream("grains.txt", options);

fileReadStream.on('data', function(chunk) {
    console.log('Grains: %s', chunk);
    console.log('Read %d bytes of data.', chunk.length);
});
```

Eg. Duplex Stream:

```
var stream = require('stream');
var util = require('util');
util.inherits(Duplexer, stream.Duplex);
function Duplexer(opt) {
 stream.Duplex.call(this, opt);
 this.data = [];
 }
Duplexer.prototype._read = function readItem(size) {
var chunk = this.data.shift();
 if (chunk == "stop"){
 this.push(null);
 } else {
 if(chunk){
 this.push(chunk);
 } else {
 setTimeout(readItem.bind(this), 500, size);
 }
 }
 };
 Duplexer.prototype._write = function(data, encoding, callback) {
 this.data.push(data);
 callback();
```

```
};
var d = new Duplexer();
d.on('data', function(chunk){
   console.log('read: ', chunk.toString());
});
d.on('end', function(){
console.log('Message Complete');
});
d.write("I think, ");
d.write("therefore ");
d.write("I am.");
d.write("Rene Descartes");
d.write("stop");
```

Piping is a mechanism where we provide the output of one stream as the input to another stream. It is normally used to get data from one stream and to pass the output of that stream to another stream. There is no limit on piping operations.
Eg. To copy data from one file to another:

```
var fs = require("fs");
// Create a readable stream
var readerStream = fs.createReadStream('count.txt');
// Create a writable stream
var writerStream = fs.createWriteStream('c.txt');
// Pipe the read and write operations
// read input.txt and write data to output.txt
readerStream.pipe(writerStream);
console.log("Program Ended");
```

Chaining is a mechanism to connect the output of one stream to another stream and create a chain of multiple stream operations. It is normally used with piping operations.
Eg. Zlib for compressing a file

```
var fs = require("fs");
var zlib = require('zlib');
// Compress the file input.txt to input.txt.gz
fs.createReadStream('c.txt')
 .pipe(zlib.createGzip())
 .pipe(fs.createWriteStream('c.txt.gz'));

console.log("File Compressed.");

fs.createReadStream('c.txt.gz')
 .pipe(zlib.createGunzip())
 .pipe(fs.createWriteStream('c.txt'));

console.log("File Decompressed.");
```

Q) Write a program in Node.js to count no.of lines, words and characters in a given file.

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

// Function to count lines, words, and characters in a file
const countLinesWordsChars = (file) => {
  let lines = 0;
  let words = 0;
  let chars = 0;

  // Create a readable stream from the file
  const stream = fs.createReadStream(file, { encoding: 'utf8' });

  // Listen for 'data' event, which is emitted whenever data is read from the
stream
  stream.on('data', (data) => {
    // Count lines by counting the number of newline characters
    lines += data.split('\n').length;

    // Count words by splitting the data by whitespace characters and
filtering out empty strings
    words += data.split(/\s+/).filter(Boolean).length;

    // Count characters by adding the length of the data
    chars += data.length;
  });

  // Listen for 'end' event, which is emitted when the end of the stream is
reached
  stream.on('end', () => {
    console.log(`Number of lines: ${lines}`);
    console.log(`Number of words: ${words}`);
    console.log(`Number of characters: ${chars}`);
  });

  // Listen for 'error' event, which is emitted when an error occurs
  stream.on('error', (err) => {
    console.error(`Error reading file: ${err}`);
  });
};

// Call the function with the file name as argument
countLinesWordsChars('count.txt');
```

Q) Write a program in Node.js to count no.of vowels, consonants, digits and special characters in a given file.

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

function countChars(filename) {
  const vowels = 'aeiouAEIOU';
  let vowelCount = 0;
  let consonantCount = 0;
  let digitCount = 0;
  let specialCharCount = 0;

  const stream = fs.createReadStream(filename, { encoding: 'utf8' });

  stream.on('data', (data) => {
    for (const char of data) {
      if (char.match(/[a-zA-Z]/)) {
        if (vowels.includes(char)) {
          vowelCount++;
        } else {
          consonantCount++;
        }
      } else if (char.match(/\d/)) {
        digitCount++;
      } else if (char.match(/\S/)) {
        specialCharCount++;
      }
    }
  });

  stream.on('end', () => {
    console.log(`Vowels: ${vowelCount}`);
    console.log(`Consonants: ${consonantCount}`);
    console.log(`Digits: ${digitCount}`);
    console.log(`Special Characters: ${specialCharCount}`);
  });

  stream.on('error', (err) => {
    console.error(`Error reading file: ${err}`);
  });
}

//reading file from user
const file = process.argv[2];
if (file) {
  countChars(file);
} else {
  console.error('Please provide a file name as an argument.');
}
```

Q) Explain about Node File I/O with suitable examples.

The Node File System (fs) module can be imported using the following syntax:
var fs = require("fs")
Synchronous vs Asynchronous
Every method in the fs module has synchronous as well as asynchronous forms.
Asynchronous methods take the last parameter as the completion function callback and the first parameter of the callback function as error.
It is better to use an asynchronous method instead of a synchronous method, as the former never blocks a program during its execution, whereas the second one does.

Eg. Asynchronous read and write:

```
var fs = require('fs');
 var fruitBowl = ['apple', 'orange', 'banana', 'grapes'];
 function writeFruit(fd){
 if (fruitBowl.length){
 var fruit = fruitBowl.pop() + " ";
 fs.write(fd, fruit, null, null, function(err, bytes){
 if (err){
 console.log("File Write Failed.");
 } else {
 console.log("Wrote: %s %dbytes", fruit, bytes);
 writeFruit(fd);
 }
 });
 } else {
 fs.close(fd);
 }
 }
 fs.open('fruit.txt', 'w', function(err, fd){
 writeFruit(fd);
});

function readFruit(fd, fruits){
    var buf = new Buffer(5);
    buf.fill();
    fs.read(fd, buf, 0, 5, null, function(err, bytes, data){
    if ( bytes > 0) {
    console.log("read %dbytes", bytes);
    fruits += data;
    readFruit(fd, fruits);
    } else {
    fs.close(fd);
    console.log ("Fruits: %s", fruits);
    }
    });
    }
```

```
    fs.open('fruit.txt', 'r', function(err, fd){
    readFruit(fd, "");
   });
```

Eg. Synchronous Read and Write

```
var fs = require('fs');
 var veggieTray = ['carrots', 'celery', 'olives'];
 fd = fs.openSync('veggie.txt', 'w');
 while (veggieTray.length){
 veggie = veggieTray.pop() + " ";
 var bytes = fs.writeSync(fd, veggie, null, null);
 console.log("Wrote %s %dbytes", veggie, bytes);
 }
fs.closeSync(fd);

fd = fs.openSync('veggie.txt', 'r');
 var veggies = "";
 do {
 var buf = new Buffer(5);
 buf.fill();
 var bytes = fs.readSync(fd, buf, null, 5);
 console.log("read %dbytes", bytes);
 veggies += buf.toString();
 } while (bytes > 0);
 fs.closeSync(fd);
console.log("Veg g (to get output shown) ies: " + veggies);
```

Table 6.1 **Flags that define how files are opened**

Mode	Description
`r`	Open file for reading. An exception occurs if the file does not exist.
`r+`	Open file for reading and writing. An exception occurs if the file does not exist.
`rs`	Open file for reading in synchronous mode. This is not the same as forcing `fs.openSync()`. When used, the OS bypasses the local file system cache. Useful on NFS mounts because it lets you skip the potentially stale local cache. You should only use this flag if necessary because it can have a negative impact on performance.
`rs+`	Same as `rs` except the file is open file for reading and writing.
`w`	Open file for writing. The file is created if it does not exist or truncated if it does exist.
`wx`	Same as `w` but fails if the path exists.
`w+`	Open file for reading and writing. The file is created if it does not exist or truncated if it exists.
`wx+`	Same as `w+` but fails if path exists.
`a`	Open file for appending. The file is created if it does not exist.
`ax`	Same as `a` but fails if the path exists.
`a+`	Open file for reading and appending. The file is created if it does not exist.
`ax+`	Same as `a+` but fails if the path exists.

Q) Write a program in Node.js to access file system
A. accessing file statistics

```
var fs = require('fs');
 fs.stat('file_stats.js', function (err, stats) {
 if (!err){
 console.log('stats: ' + JSON.stringify(stats, null, ' '));
 console.log(stats.isFile() ? "Is a File" : "Is not a File");
 console.log(stats.isDirectory() ? "Is a Folder" : "Is not a Folder");
 console.log(stats.isSocket() ? "Is a Socket" : "Is not a Socket");
 console.log(stats.isDirectory() ? "Is a Directory" : "Is not a Directory");
 stats.isBlockDevice();
 stats.isCharacterDevice();
 stats.isSymbolicLink(); //only lstat
 stats.isFIFO();

 }
});
```

Table 6.2 **Attributes and methods of `Stats` objects for file system entries**

Attribute/Method	Description
`isFile()`	Returns `true` if the entry is a file
`isDirectory()`	Returns `true` if the entry is a directory
`isSocket()`	Returns `true` if the entry is a socket
`dev`	Specifies the device ID on which the file is located
`mode`	Specifies the access mode of the file
`size`	Specifies the number of bytes in the file
`blksize`	Specifies the block size used to store the file in bytes
`blocks`	Specifies the number of blocks the file is taking on disk
`atime`	Specifies the time the file was last accessed
`mtime`	Specifies the time the file was last modified
`ctime`	Specifies the time the file was created

b. **To list files/directories in a given directory:**

```
var fs = require('fs');
 var Path = require('path');
 function WalkDirs(dirPath){
    console.log(dirPath);
    fs.readdir(dirPath, function(err, entries){
    for (var idx in entries){
        var fullPath = Path.join(dirPath, entries[idx]);
        (function(fullPath){
            fs.stat(fullPath, function (err, stats){
            if (stats.isFile()){
                console.log(fullPath);
            } else if (stats.isDirectory()){
                WalkDirs(fullPath);
            }
        });
        })(fullPath);
    }
    });
 }

WalkDirs("../Files");
```

C. To create a directory:

```
let fs = require('fs')
fs.mkdir("../Files/folderA", function(err){
    console.log(err ? "Directory not created" : "Directory created.");
});
```

D. **Listing Files:**
fs.readdir(path, callback)
fs.readdirSync(path)
e. Deleting Files:
fs.unlink(path, callback)
fs.unlinkSync(path)
eg.

```
fs.unlink("new.txt", function(err){
console.log(err ? "File Delete Failed" : "File Deleted");
 });
```

F. **Truncating Files:**
To truncate a file, use one the following fs calls and pass in the number of bytes you want the file to contain when the truncation completes:
fs.truncate(path, len, callback)
fs.truncateSync(path, len)
The truncateSync(path) returns true or false based on whether the file is successfully truncated. The asynchronous truncate() call passes an error value to the callback function if an error is encountered when truncating the file.
Eg.

```
fs.truncate("new.txt", function(err){
console.log(err ? "File Truncate Failed" : "File Truncated");
});
```

G. **Making and Removing Directories:**
fs.mkdir(path, [mode], callback)
fs.mkdirSync(path, [mode])
The mkdirSync(path) returns true or false based on whether the directory is successfully created. The asynchronous mkdir() call passes an error value to the callback function if an error is encountered when creating the directory.
Eg.

```
let fs = require('fs')
fs.mkdir("../Files/folderA", function(err){
    console.log(err ? "Directory not created" : "Directory created.");
});
```

Output:

```
node CreateDir.js
Directory created.
```

H. **Delete Directories:**
fs.rmdir(path, callback)
fs.rmdirSync(path)
eg.

```
let fs = require('fs')
fs.rmdir("../Files/folderA", function(err){
    console.log(err ? "Directory not deleted": "Directory deleted.");
});
```

I. Renaming Files and Directories:

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

The oldPath specifies the existing file or directory path, and the newPath specifies the new name. The renameSync(path) returns true or false based on whether the file or directory is successfully renamed. The asynchronous rename() call passes an error value to the callback function if an error is encountered when renaming the file or directory.

Eg.
fs.rename("old.txt", "new.txt", function(err){ console.log(err ? "Rename Failed" : "File Renamed"); });

fs.rename("testDir", "renamedDir", function(err){ console.log(err ? "Rename Failed" : "Folder Renamed"); });

J. Watching for File Changes:
the fs module provides a useful tool to watch a file and execute a callback function when the file changes. This can be useful if you want to trigger events to occur when a file is modified, but do not want to continually poll from your application directly. This does incur some overhead in the underlying OS, so you should use watches sparingly.

fs.watchFile(path, [options], callback)

When a file change occurs, the callback function is executed and passes a current and previous Stats object.

Eg.
fs.watchFile("log.txt", {persistent:true, interval:5000}, function (curr, prev) { console.log("log.txt modified at: " + curr.mtime); console.log("Previous modification was: " + prev.mtime);
});

Q) Explain Events and Methods available on HTTP ClientRequest and ServerResponse objects.

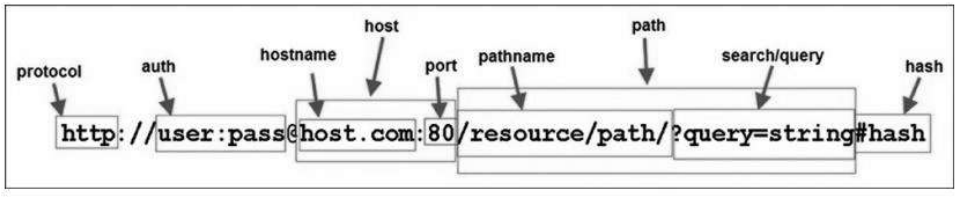

Figure 7.1 Basic components that can be included in a URL

Table 7.1 **Properties of the `URL` object**

Property	Description
`href`	This is the full URL string that was originally parsed.
`protocol`	The `request` protocol lowercased.
`host`	The full `host` portion of the URL including `port` information lowercased.
`auth`	The `authentication` information portion of a URL.
`hostname`	The `hostname` portion of the host lowercased.
`port`	The `port` number portion of the host.
`pathname`	The `path` portion of the URL including the initial slash if present.
`search`	The `query` string portion of the URL including the leading question mark.
`path`	The full `path` including the `pathname` and `search`.
`query`	This is either the `parameter` portion of the query string or a parsed object containing the `query` string parameters and values if the `parseQueryString` is set to `true`.
`hash`	The `hash` portion of the URL including the pound sign (#).

The http.ClientRequest Object:
The ClientRequest object is created internally when you call http.request() when building the HTTP client.
To implement a ClientRequest object, you use a call to http.request() using the following syntax: http.request(options, callback)

Table 7.2 **Options that can be specified when creating a `ClientRequest`**

Property	Description
`host`	The domain name or IP address of the server to issue the request Defaults to `localhost`.
`hostname`	Same as `host` but preferred over host to support `url.parse(`
`port`	Port of remote server. Defaults to 80.
`localAddress`	Local interface to bind for network connections.
`socketPath`	Unix Domain Socket (use one of `host:port` or `socketPath`
`method`	A string specifying the HTTP request method. For example, `GET` `POST`, `CONNECT`, `OPTIONS`, etc. Defaults to `GET`.
`path`	A string specifying the requested resource path. Defaults to `/`. Th should also include the query string if any. For example: `/book.html?chapter=12`
`headers`	An object containing request headers. For example: `{ 'content-length': '750', 'content-type': 'text/plain`
`auth`	Basic authentication in the form of `user:password` used to compute an `Authorization` header.
`agent`	Defines the `Agent` behavior. When an `Agent` is used, request defaults to `Connection:keep-alive`. Possible values are: `undefined` (default): Uses global `Agent`. `Agent` object: Uses specific `Agent` object. `false`: Disables `Agent` behavior.

Table 7.3 **Events available on ClientRequest objects**

Property	Description
`response`	Emitted when a response to this request is received from the server. The callback handler receives back an `IncomingMessage` object as the only parameter.
`socket`	Emitted after a socket is assigned to this request.
`connect`	Emitted every time a server responds to a request that was initiated with a `CONNECT` method. If this event is not handled by the client, then the connection will be closed.
`upgrade`	Emitted when the server responds to a request that includes an Update request in the headers.
`continue`	Emitted when the server sends a `100 Continue HTTP` response instructing the client to send the request body.

Table 7.4 **Methods available on `ClientRequest` objects**

Method	Description
`write(chunk, [encoding])`	Writes a chunk, `Buffer` or `String` object, of body data into the request. This allows you to stream data into the `Writable` stream of the `ClientRequest` object. If you stream the body data, you should include the `{'Transfer-Encoding', 'chunked'}` header option when you create the request. The encoding parameter defaults to `utf8`.
`end([data], [encoding])`	Writes the optional data out to the request body and then flushes the `Writable` stream and terminates the request.
`abort()`	Aborts the current request.
`setTimeout(timeout, [callback])`	Sets the socket timeout for the request.
`setNoDelay ([noDelay])`	Disables the Nagle algorithm, which buffers data before sending it. The `noDelay` argument is a Boolean that is `true` for immediate writes and `false` for buffered writes.
`setSocketKeepAlive ([enable], [initialDelay])`	Enables and disables the `keep-alive` functionality on the client request. The `enable` parameter defaults to `false`, which is disabled. The `initialDelay` parameter specifies the delay between the last data packet and the first `keep-alive` request.

The http.ServerResponse Object :

The ServerResponse object is created by the HTTP server internally when a request event is received. It is passed to the request event handler as the second argument. You use the ServerRequest object to formulate and send a response to the client. The ServerResponse implements a Writable stream, so it provides all the functionality of a Writable stream object. For example, you can use the write() method to write to it as well as pipe a Readable stream into it to write data back to the client.

Table 7.5 **Events available on `ServerResponse` objects**

Property	Description
`close`	Emitted when the connection to the client is closed prior to sending the `response.end()` to finish and flush the response.
`headersSent`	A Boolean that is `true` if headers have been sent; otherwise, `false`. This is read only.
`sendDate`	A Boolean that, when set to `true`, the `Date` header is automatically generated and sent as part of the response.
`statusCode`	Allows you to specify the response status code without having to explicitly write the headers. For example: `response.statusCode = 500;`

Table 7.6 **Methods available on `ServerResponse` objects**

Method	Description
`writeContinue()`	Sends an `HTTP/1.1 100 Continue` message to the client requesting that the body data be sent.

`writeHead(statusCode, [reasonPhrase], [headers])`	Writes a response header to the request. The `statusCode` parameter is the three-digit HTTP response status code, for example, `200`, `401`, `500`. The optional `reasonPhrase` is a string denoting the reason for the `statusCode`. The `headers` are the response headers object, for example:
	`response.writeHead(200, 'Success', {`
	`'Content-Length': body.length,`
	`'Content-Type': 'text/plain' });`

`setTimeout(msecs, callback)`	Sets the socket timeout for the client connection in milliseconds along with a `callback` function to be executed if the timeout occurs.
`setHeader(name, value)`	Sets the value of a specific header where `name` is the HTTP header name and `value` is the header value.
`getHeader(name)`	Gets the value of an HTTP header that has been set in the response.
`removeHeader(name)`	Removes an HTTP header that has been set in the response.
`write(chunk, [encoding])`	Writes a `chunk`, `Buffer` or `String` object, of data out to the response `Writable` stream. This only writes data to the body portion of the response. The default encoding is `utf8`. This returns `true` if the data is written successfully or `false` if the data is written to user memory. If it returns `false`, then a `drain` event is emitted by the `Writable` stream when the buffer is free again.
`addTrailers(headers)`	Adds HTTP trailing headers to the end of the response.
`end([data], [encoding])`	Writes the optional data out to the response body and then flushes the `Writable` stream and finalizes the response.

Q) Implement HTTP Services in Node.js to read user name from user and greet the user as the response.

index.html:

```
<!DOCTYPE html>
<html>
<head>
  <title>Greeting Form</title>
</head>
<body>
  <form action="/greet" method="POST">
    <label for="name">Enter your name:</label>
    <input type="text" id="name" name="name">
    <button type="submit">Submit</button>
  </form>
</body>
</html>
```

index.js

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

const server = http.createServer((req, res) => {
  if (req.method === 'GET' && req.url === '/') {
    // Read the HTML file
    fs.readFile(path.join(__dirname, 'index.html'), 'utf8', (err, data) => {
      if (err) {
        res.statusCode = 500;
        res.end('Internal Server Error');
      } else {
        res.setHeader('Content-Type', 'text/html');
        res.end(data);
      }
    });
  } else if (req.method === 'POST' && req.url === '/greet') {
    let data = '';

    // Collect the data from the request
    req.on('data', chunk => {
      data += chunk;
    });

    // Process the collected data
    req.on('end', () => {
      const name = new URLSearchParams(data).get('name');
      const greeting = `Hello, ${name}!`;

      res.setHeader('Content-Type', 'text/plain');
```

```
      res.statusCode = 200;
      res.end(greeting);
    });
  } else {
    res.statusCode = 404;
    res.end('Not Found');
  }
});

// Start the server on port 3000
server.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

O/P:

node index.js

Server listening on port 3000