Q1: Introduction to NodeJS

Introduction: Node.js is an open source and cross-platform runtime environment for executing JavaScript code outside of a browser. You need to remember that NodeJS is not a framework and it's not a programming language.
We often use Node.js for building back-end services like APIs like Web App or Mobile App. It's used in production by large companies such as Paypal, Uber, Netflix, Wallmart and so on

Q2: Features of NodeJS

- It's easy to get started and can be used for prototyping and agile development
- It provide fast and highly scalable services
- It uses JavaScript everywhere so it's easy for a JavaScript programmer to build back-end services using Node.js
- Source code more cleaner and consistent.
- Large ecosystem for open source library.
- It has Asynchronous or Non blocking nature.

Q3: Advantages of NodeJS

Easy Scalability: Developers prefer to use Node.js because it is easily scale the application in both horizontal and vertical direction. We can also add extra resources during the scalability of application.
Real time web apps: If you are building a web app you can also use PHP and it will take the same amount of time when you use Node.js, But if I am talking about building chat apps or gaming apps Node.js is much more preferable because of faster synchronization. Also, event loop avoid HTTP overload for Node.js development.
Fast Suite: NodeJs runs on the V8 engine developed by Google. Event loop in NodeJs handles all asynchronous operation so NodeJs acts like a fast suite and all the operations can be done quickly like reading or writing in the database, network connection or file system
Easy to learn and code: NodeJs is easy to learn and code because it uses JavaScript. If you are a front-end developer and have a good grasp on JavaScript you can easily learn and build the application on NodeJS
Advantage of Caching: It provides the caching of single module. Whenever there is any request for the first module, it gets cached in the application memory so you don't need to re-execute the code.
Data Streaming: In NodeJs HTTP request and response are considered as two separate events. They are data stream so when you process a file at the time of loading it will reduce the overall time and will make it faster when the data is presented in the form of transmissions. It also allows you to stream audio and video files at lightning speed.
Hosting: PaaS (Platform as a Service) and Heroku are the hosting platform for NodeJS application deployment which is easy to use without facing any issue.
Corporate Support: Most of the well known companies like Wallmart, Paypal, Mirosoft, yahoo are using NodeJS for building the applications. NodeJS uses JavaScript so most of the companies are combining front-end and backend Teams together into a single unit.

Q4: Application of NodeJS:

- Real Time Chats,
- Complex Single-Page applications,
- Real-time collaboration tools,
- Streaming apps
- JSON APIs based application

Q5: Difference between NodeJS and AngularJS

Angular JS	Node.JS
It is a structural framework for developing dynamic web apps.	It is a cross-platform run-time environment for applications written in JavaScript language.
It is entirely written in JavaScript.	It is written in C, C++ and JavaScript.
It is used to build single-page client-side applications.	It is used to build fast and scalable server side networking applications.
Ideal for developing highly active and interactive web apps.	Ideal for developing small size projects.
The developer only need to add the AngularJS file to use it in his application.	The developer need to install the NodeJS on his computer system.
AngularJS is a Web Framework.	NodeJS provides different Web Frameworks like Socket.io, Hapi.js, Meteor.js, Express.js, and Sails.js

Q6: What is Node.js? Where can you use it?

You can use Node.js in developing I/O intensive web applications like video streaming sites. You can also use it for developing: Real-time web applications, Network applications, General-purpose applications and Distributed systems.

Q7: How do you update NPM to a new version in Node.js?

You use the following commands to update NPM to a new version:
$ sudo npm install npm -g
/usr/bin/npm -> /usr/lib/node_modules/npm/bin/npm-cli.js
npm@2.7.1 /usr/lib/node_modules/npm

Q8: Why is Node.js Single-threaded?

Node.js is single-threaded for async processing. By doing async processing on a single-thread under typical web loads, more performance and scalability can be achieved as opposed to the typical thread-based implementation.

Q9: Explain callback in Node.js.

A callback function is called at the completion of a given task. This allows other code to be run in the meantime and prevents any blocking. Being an asynchronous platform, Node.js heavily relies on callback. All APIs of Node are written to support callbacks.

@10: What is callback hell in Node.js?

Callback hell is the result of heavily nested callbacks that make the code not only unreadable but also difficult to maintain. For example:

Q11: How do you prevent/fix callback hell?

The three ways to prevent/fix callback hell are: 1.Handle every single error
2.Keep your code shallow
3.Modularize – split the callbacks into smaller, independent functions that can be called with some parameters then joining them to achieve desired results.

Q12: Explain the role of REPL in Node.js.

As the name suggests, REPL (Read Eval print Loop) performs the tasks of – Read, Evaluate, Print and Loop. The REPL in Node.js is used to execute ad-hoc Javascript statements. The REPL shell allows entry to javascript directly into a shell prompt and evaluates the results. For the purpose of testing, debugging, or experimenting, REPL is very critical.

Q13: Name the types of API functions in Node.js.

There are two types of functions in Node.js.:
Blocking functions - In a blocking operation, all other code is blocked from executing until an I/O event that is being waited on occurs. Blocking functions execute synchronously For example:

```
const fs = require('fs');
const data = fs.readFileSync('/file.md');
console.log(data);
moreWork();
```

The second line of code blocks the execution of additional JavaScript until the entire file is read. moreWork () will only be called after Console.log
Non-blocking functions - In a non-blocking operation, multiple I/O calls can be performed without the execution of the program being halted. Non-blocking functions execute asynchronously. For example:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) =>
if (err) throw err;);
moreWork();
```

Since fs.readFile () is non-blocking, moreWork () does not have to wait for the file read to complete before being called. This allows for higher throughput.

Q14: Which is the first argument typically passed to a Node.js callback handler?

Typically, the first argument to any callback handler is an optional error object. The argument is null or undefined if there is no error. Error handling by a typical callback handler could be as follows:

```
function callback(err, results)
if(err)
```

Q15: What are the functionalities of NPM in Node.js?

Online repository for Node.js packages
Command line utility for installing packages, version management and dependency management of Node.js packages

Q16: What is the difference between Node.js and Ajax?

Node.js and Ajax (Asynchronous JavaScript and XML) are the advanced implementation of JavaScript. They all serve completely different purposes.

Ajax is primarily designed for dynamically updating a particular section of a page's content, without having to update the entire page.
Node.js is used for developing client-server applications.

Q17: Explain chaining in Node.js.

Chaining is a mechanism whereby the output of one stream is connected to another stream creating a chain of multiple stream operations.

Q18: What are "streams" in Node.js? Explain the different types of streams present in Node.js.

Streams are objects that allow reading of data from the source and writing of data to the destination as a continuous process.
There are four types of streams. 1.Readable to facilitate the reading operation
2.Writable to facilitate the writing operation
3.Duplex to facilitate both read and write operations
4.Transform is a form of Duplex stream that performs computations based on the available input

Q19: What are exit codes in Node.js? List some exit codes.

Exit codes are specific codes that are used to end a "process" (a global object used to represent a node process).
Examples of exit codes include:
Unused
Uncaught Fatal Exception
Fatal Error
Non-function Internal Exception Handler
Internal Exception handler Run-Time Failure
Internal JavaScript Evaluation Failure

Q20: What are Globals in Node.js?

Three keywords in Node.js constitute as Globals. These are:
Global – it represents the Global namespace object and acts as a container for all other global objects.
Process – It is one of the global objects but can turn a synchronous function into an async callback. It can be accessed from anywhere in the code and it primarily gives back information about the application or the environment.
Buffer – it is a class in Node.js to handle binary data.

Q21: Why is consistent style important and what tools can be used to assure it?

Consistent style helps team members modify projects easily without having to get used to a new style every time. Tools that can help include Standard and ESLint.

NodeIntroduction
advanced
modules
FileSystem
eventLoop
readHtmlFileByNode
basicNode
nodeProgrammingTechniques

A common task for a web server can be to open a file on the server and return the content to the client.how Node.js handles a file request:

Sends the task to the computers file system.
Ready to handle the next request.
When the file system has opened and read the file, the server returns the content to the client.
Node.js eliminates the waiting, and simply continues with the next request.

Node.js runs single-threaded, non-blocking, asynchronously programming, which is very memory efficient.

What Can Node.js Do?

Node.js can generate dynamic page content
Node.js can create, open, read, write, delete, and close files on the server
Node.js can collect form data
Node.js can add, delete, modify data in your database

What is a Node.js File?

Node.js files contain tasks that will be executed on certain events
A typical event is someone trying to access a port on the server
Node.js files must be initiated on the server before having any effect
Node.js files have extension ".js"

Module

it a set of functions you want to include in your application. To include a module, use the require() function with the name of the module:

Create Your Own Modules

```
var http = require('http');
exports.myDateTime = function () {
return Date();
};
```

Node.js HTTP Module

A set of functions you want to include in your application. Node.js has a set of built-in modules which you can use without any further installation. assert - Provides a set of assertion tests The assert module provides a way of testing expressions. If the expression evaluates to 0, or false, an assertion failure is being caused, and the program is terminated.

```
var assert = require('assert');
assert(5 > 7);
```

o/p= AssertionError: false == true

Method

1.assert() - Checks if a value is true. Same as assert.ok()
deepEqual() - Checks if two values are equal
deepStrictEqual() - Checks if two values are equal, using the strict equal operator (===)
2.buffer - To handle binary data
The buffers module provides a way of handling streams of binary data. The Buffer object is a global object in Node.js, and it is not necessary to import it using the require keyword.

```
var buf = Buffer.from('abc');
console.log(buf);
```

alloc() - Creates a Buffer object of the specified length
allocUnsafe() - Creates a non-zero-filled Buffer of the specified length
byteLength() - Returns the numbers of bytes in a specified object
compare() - Compares two Buffer objects
concat() - Concatenates an array of Buffer objects into one Buffer object
copy()- Copies the specified number of bytes of a Buffer object
entries() - Returns an iterator of "index" "byte" pairs of a Buffer object
equals() - Compares two Buffer objects, and returns true if it is a match, otherwise false
fill() - Fills a Buffer object with the specified values
from() - Creates a Buffer object from an object (string/array/buffer)
includes() - Checks if the Buffer object contains the specified value. Returns true if there is a match, otherwise false
indexOf() - Checks if the Buffer object contains the specified value. Returns the first occurrence, otherwise -1
keys() - Returns an array of keys in a Buffer object
length - Returns the length of a Buffer object, in bytes
poolSize- Sets or returns the number of bytes used for pooling
slice() - Slices a Buffer object into a new Buffer objects starting and ending at the specified positions
swap16()- Swaps the byte-order of a 16 bit Buffer object
toString()- Returns a string version of a Buffer object
toJSON()- Returns a JSON version of a Buffer object
values()- Returns an array of values in a Buffer object
write() - Writes a specified string to a Buffer object

child_process - To run a child processcluster

To split a single Node process into multiple processes The cluster module provides a way of creating child processes that runs simultaneously and share the same server port. Node.js runs single threaded programming, which is very memory efficient, but to take advantage of computers multi-core systems, the Cluster module allows you to easily create child processes that each runs on their own single thread, to handle the load.

```
var cluster = require('cluster');
if (cluster.isWorker) {
console.log('I am a worker');
}
else {
console.log('I am a master');
cluster.fork();
cluster.fork();
}
```

o/p - I am a master
I am a worker
I am a worker

disconnect

disconnect() Disconnects all workers
exitedAfterDisconnect- Returns true if a worker was exited after disconnect, or the kill method
fork() Creates a new worker, from a master
id - A unique id for a worker
isConnected - Returns true if the worker is connected to its master, otherwise false

isDead - Returns true if the worker's process is dead, otherwise false
isMaster - Returns true if the current process is master, otherwise false
isWorker - Returns true if the current process is worker, otherwise false
kill() - Kills the current worker
process - Returns the global Child Process
schedulingPolicy - Sets or gets the schedulingPolicy
send() - sends a message to a master or a worker
settings - Returns an object containing the clusters settings
setupMaster() - Changes the settings of a cluster
worker - Returns the current worker object
workers - Returns all workers of a master

crypto

crypto - To handle OpenSSL cryptographic functions The crypto module provides a way of handling encrypted data. (i) Encrypt the text 'abc'

```
var crypto = require('crypto');
var mykey = crypto.createCipher('aes-128-cbc', 'mypassword');
var mystr = mykey.update('abc', 'utf8', 'hex')
mystr += mykey.update.final('hex');
console.log(mystr);
```

o/p - 34feb914c099df25794bf9ccb85bea72
(ii) Decrypt back to 'abc'

```
var crypto = require('crypto');
var mykey = crypto.createDecipher('aes-128-cbc', 'mypassword');
var mystr = mykey.update('34feb914c099df25794bf9ccb85bea72', 'hex', 'utf8')
mystr += mykey.update.final('utf8');

console.log(mystr);
```

o/p - abc

constants

constants - Returns an object containing Crypto Constants
fips - Checks if a FIPS crypto provider is in use
createCipher() - Creates a Cipher object using the specific algorithm and password
createCipheriv() - Creates a Cipher object using the specific algorithm, password and initialization vector
createDecipher() - Creates a Decipher object using the specific algorithm and password
createDecipheriv() - Creates a Decipher object using the specific algorithm, password and initialization vector
createDiffieHellman() - Creates a DiffieHellman key exchange object
createECDH() - Creates an Elliptic Curve Diffie Hellmann key exchange object
createHash() - Creates a Hash object using the specified algorithm
createHmac() - Creates a Hmac object using the specified algorithm and key
createSign() - Creates a Sign object using the specified algorithm and key
createVerify() - Creates a Verify object using the specified algorithm
getCiphers - Returns an array of supported cipher algorithms
getCurves() - Returns an array of supported elliptic curves
getDiffieHellman() - Returns a predefined Diffie Hellman key exchange object
getHashes() - Returns an array of supported hash algorithms
pbkdf2() - Creates a Password Based Key Derivation Function 2 implementation
pbkdf2Sync() - Creates a synchronous Password Based Key Derivation Function 2 implementation
privateDecrypt() - Decrypts data using a private key
timingSafeEqual() - Compare two Buffers and returns true is they are equal, otherwise false
privateEncrypt() - Encrypts data using a private key
publicDecrypt() - Decrypts data using a public key
publicEncrypt() - Encrypts data using a public key
setEngine() - Sets the engine for some or all OpenSSL function

dgram

Provides implementation of UDP datagram sockets It can be used to send messages from one computer/server to another. (i)

```
var dgram = require('dgram');
var s = dgram.createSocket('udp4');
s.on('message', function(msg, rinfo)
console.log('I got this message: ' + msg););
s.bind(8080);
```

(ii) var dgram = require('dgram');

```
var s = dgram.createSocket('udp4');
s.send(Buffer.from('abc'), 8080, 'localhost');
```

When initiating the second file, the first Command window will now look like this:
o/p - I got this message: abc

dns

To do DNS lookups and name resolution functions
var dns = require('dns');
var w3 = dns.lookup('www.w3schools.com', function (err, addresses, family)
console.log(addresses););**o/p -** 192.229.133.221

getServers

getServers() - Returns an array containing all IP addresses belonging to the current server
lookup() - Looks up a hostname. A callback function contains information about the hostname, including it's IP address
lookupService() - Looks up a address and port. A callback function contains information about the address, such as the hostname
resolve() - Returns an array of record types belonging to the specified hostname
resolve4() - Looks up an IPv4 address. The callback function includes an array of IPv4 addresses
resolve6() - Looks up an IPv6 address. The callback function includes an array of IPv6 addresses
resolveCname() - Looks up CNAME records for the specified hostname. The callback function includes an array of available domains for the hostname
resolveMx() - Looks up mail exchange records for the specified hostname.
resolveNaptr() - Looks up regular expression based records for the specified hostname.
resolveNs() - Looks up name server records for the specified hostname.
resolveSoa() - Looks up a start of authority record for the specified hostname.
resolveSrv() - Looks up service records for the specified hostname.
resolvePtr() - Looks up pointer records for the specified hostname.
resolveTxt() - Looks up text query records for the specified hostname.v**reverse() -** Reverses an IP address into an array of hostnames
setServers() - Sets the IP addresses of the servers

events

events - To handle events
In Node.js, all events are an instance of the EventEmitter object
In Node.js, all events are an instance of the EventEmitter object
o/p - A scream is detected!
addListener() - Adds the specified listener
defaultMaxListeners - Sets the maximum number of listeners allowed for one event. Default is 10
emit() - Call all the listeners registered with the specified name
eventNames() - Returns an array containing all registered events
getMaxListeners() - Returns the maximum number of listeners allowed for one event
listenerCount() - Returns the number of listeners with the specified name
listeners() - Returns an array of listeners with the specified name
on() - Adds the specified listener
once() - Adds the specified listener once. When the specified listener has been executed, the listener is removed
prependListener() - Adds the specified listener as the first event with the specified name
prependOnceListener() - Adds the specified listener as the first event with the specified name, once. When the specified listener has been executed, the listener is removed
removeAllListeners() - Removes all listeners with the specified name, or ALL listeners if no name is specified
removeListener() - Removes the specified listener with the specified name
setMaxListeners() - Sets the maximum number of listeners allowed for one event. Default is 10

fs - To handle the file system

var fs = require('fs');
fs.readFile('demofile.txt', 'utf8', function(err, data)
if (err) throw err; console.log(data););
o/p - Hello! Welcome to demofile.txt
access() - Checks if a user has access to this file or directory
accessSync() - Same as access(), but synchronous instead of asynchronous
appendFile() - Appends data to a file
appendFileSync() - Same as appendFile(), but synchronous instead of asynchronous
chmod() - Changes the mode of a file
chmodSync() - Same as chmod(), but synchronous instead of asynchronous
chown() - Changes the owner of a file
chownSync() - Same as chown(), but synchronous instead of asynchronous
close() - Closes a file
closeSync() - Same as close(), but synchronous instead of asynchronous
constants - Returns an object containing constant values for the file system
createReadStream() - Returns a new stream object
createWriteStream() - Returns a new writeable stream object
exists() - Deprecated. Checks if a file or folder exists
existsSync() -Same as exists(), but synchronous instead of asynchronous. This method is NOT deprecated

fchmod() - Changes the mode of a file
fchmodSync() - Same as fchmod(), but synchronous instead of asynchronous
fchown() - Changes the owner of a file
fchownSync() - Same as fchown(), but synchronous instead of asynchronous
fdatasync() - Syncronizes a file with the one stored on the computer
fdatasyncSync() - Same as fdatasync(), but synchronous instead of asynchronous
fstat() - Returns the status of a file
fstatSync() - Same as fstat(), but synchronous instead of asynchronous
fsync() - Syncronizes a file with the one stored on the computer
fsyncSync() - Same as fsync(), but synchronous instead of asynchronous
ftruncated() - Truncates a file
ftruncatedSync() - Same as ftruncated(), but synchronous instead of asynchronous
futimes() - Change the timestamp of a file
futimesSync() - Same as futimes(), but synchronous instead of asynchronous
lchmod() - Changes the mode of a file, for Mac OS X
lchmodSync() - Same as lchmod(), but synchronous instead of asynchronous
lchown() - Changes the owner of a file, for Mac OS X
lchownSync() - Same as lchown(), but synchronous instead of asynchronous
link() - Makes an addition name for a file. Both the old and the new name may be used
linksync() - Same as link(), but synchronous instead of asynchronous
lstat() - Returns the status of a file
lstatSync() - Same as lstat(), but synchronous instead of asynchronous
mkdir() - Makes a new directory
mkdirSync() - Same as mkdir(), but synchronous instead of asynchronous
mkdtemp() - Makes a new temporary directory
mkdtempSync() - Same as mktemp(), but synchronous instead of asynchronous
open() - Opens a file
openSync() - Same as open(), but synchronous instead of asynchronous
read() - Reads the content of a file
readdir() - Reads the content of a directory
readdirSync() - Same as readdir(), but synchronous instead of asynchronous
readFile() - Reads the content of a file
readFileSync() - Same as readFile(), but synchronous instead of asynchronous
readlink() - Reads the value of a link
readlinkSync() - Same as readlink(), but synchronous instead of asynchronous
realpath() - Returns the absolute pathname
realpathSync() - Same as realpath(), but synchronous instead of asynchronous
rename() - Renames a file
renameSync() - Same as rename(), but synchronous instead of asynchronous
rmdir() - Removes a directory
rmdirSync() - Same as rmdir(), but synchronous instead of asynchronous
stat() - Returns the status of a file
statSync() - Same as stat(), but synchronous instead of asynchronous
symlink() - Makes a symbolic name for a file
truncate() - Truncates a file
truncateSync() - Same as truncate(), but synchronous instead of asynchronous
unlink() - Removes a link
unlinkSync() - Same as unlink(), but synchronous instead of asynchronous
utimes() - Change the timestamp of a file
utimesSync() - Same as utimes(), but synchronous instead of asynchronous
watch() - Watch for changes of a filename or directoryname
watchFile() - Watch for changes of a filename
write() - Writes buffer to a file
write() - Writes data to a file
writeFile() -Writes data to a file
writeFileSync() -Same as writeFile(), but synchronous instead of asynchronous
writeSync() -Same as write(), but synchronous instead of asynchronous
writeSync() - Same as write(), but synchronous instead of asynchronous

http

http - To make Node.js act as an HTTP server.
The HTTP module provides a way of making Node.js transfer data over HTTP.
var http = require('http');.

http.createServer(function (req, res) .
res.writeHead(200, {'Content-Type': 'text/plain'}); res.write('Hello World!'); res.end();).listen(8080);.
o/p - Hello World!

createClient()- Deprecated. Creates a HTTP client
createServer()- Creates an HTTP server
get()- Sets the method to GET, and returns an object containing the users request
globalAgent- Returns the HTTP Agent
request()- Returns an object containing the users request

https

To make Node.js act as an HTTPS server.
The HTTPS module provides a way of making Node.js transfer data over HTTP TLS/SSL protocol, which is the secure HTTP protocol.
var https = require('https');
https.createServer(function (req, res)
res.writeHead(200, {'Content-Type': 'text/plain'}); res.write('Hello World!'); res.end();).listen(8080);
o/p - Hello World!**createServer() -** Creates an HTTPS server
get() - Sets the method to GET, and returns an object containing the users request
globalAgent - Returns the HTTPS Agent
request - Makes a request to a secure web server

(i) Node.js HTTP Module

var http = require('http');
http.createServer(function (req, res)
res.writeHead(200, {'Content-Type': 'text/html'}); //Return the url part of the request object: res.write(req.url);
res.end();).listen(8080);
o/p - /summer
(ii) Split the Query String There are built-in modules to easily split the query string into readable parts, such as the URL module.
var http = require('http');
var url = require('url');
http.createServer(function (req, res)
res.writeHead(200, {'Content-Type': 'text/html'}); /*Use the url module to turn the querystring into an object:*/ var q = url.parse(req.url, true).query; /*Return the year and month from the query object:*/ var txt = q.year + " " + q.month;
res.end(txt);).listen(8080);
o/p - 2017 July

net

net - To create servers and clients
var net = require('net');
connect() - Creates a new connection to the server, and returns a new Socket
createConnection() - Creates a new connection to the server, and returns a new Socket
createServer() - Creates a new server
isIP - Checks if the specified value is an IP address
isIPv4 - Checks if the specified value is an IPv4 address
isIPv6 - Checks if the specified value is an IPv6 address
os - Provides information about the operation system

querystring

To handle URL query strings
The Query String module provides a way of parsing the URL query string.
var querystring = require('querystring');
var q = querystring.parse('year=2017&month=february');
console.log(q.year);
o/p - 2018

escape() - Returns an escaped querystring
parse() - Parses the querystring and returns an object
stringify() - Stringifies an object, and returns a query string
unescape() - Returns an unescaped query string

readline

To handle readable streams one line at the time The Readline module provides a way of reading a datastream, one line at a time.

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

var myInterface = readline.createInterface({
input: fs.createReadStream('demofile1.html')

```
});

var lineno = 0;
myInterface.on('line', function (line) {
lineno++;
console.log('Line number ' + lineno + ': ' + line);
});

o/p - Line number 1: '<'!DOCTYPE html>
Line number 2: '<'html>
Line number 3: '<'title>Demo - Open file'<'/title>
Line number 4: '<'body>
Line number 5: '<'h1>My Header'<'/h1>
Line number 6: '<'p>My paragraph'<'/p>
Line number 7: '<'/body>
Line number 8: '<'/html>
```

clearLine() - Clears the current line of the specified stream
clearScreenDown() - Clears the specified stream from the current cursor down position
createInterface() - Creates an Interface object
cursorTo() - Moves the cursor to the specified position
emitKeypressEvents() - Fires keypress events for the specified stream
moveCursor() - Moves the cursor to a new position, relative to the current position

stream

To handle streaming data
The Stream module provides a way of handling streaming data.
There are two types of streams: readable and writeable.
An example of a readable stream is the response object you get when working with the http.createServer() method.

```
var http = require('http');
http.createServer(function (req, res)
res.writeHead(200, {'Content-Type': 'text/html'}); res.write('Hello World!'); res.end();).listen(8080);
```

o/p - Hello World!
isPaused() - Returns true if the state of the readable stream is paused, otherwise false
pause() - Pauses the readable stream
pipe() - Turns the readable stream into the specified writable stream
read() - Returns a specified part of the readable stream
resume() - Resumes a paused stream
setEncoding() - Sets the character encoding of the readable stream
unpipe() - Stops turning a readable stream into a writable stream, caused by the pipe() method
unshift() - Pushes some specified data back into the internal buffer
wrap() - Helps reading streams made by older Node.js versions
string_decoder - To decode buffer objects into strings

timers

To execute a function after a given number of milliseconds
The Timers module provides a way scheduling functions to be called later at a given time.
The Timer object is a global object in Node.js, and it is not necessary to import it using the require keyword.

```
var myInt = setInterval(function ()
console.log("Hello");, 500);
```

o/p - Hello Hello**clearImmediate() -** Cancels an Immediate object
clearInterval() - Cancels an Interval object
clearTimeout() - Cancels a Timeout object
ref() - Makes the Timeout object active. Will only have an effect if the Timeout.unref() method has been called to make the Timeout object inactive.
setImmediate() - Executes a given function immediately.
setInterval() - Executes a given function at every given milliseconds
setTimeout() - Executes a given function after a given time (in milliseconds)
unref() - Stops the Timeout object from remaining active.

tls

To implement TLS and SSL protocols
The TLS module provides a way of implementing TLS (Transport Layer Security) and SSL (Secure Socket Layer).

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

connect() - Returns a Socket object
createSecureContext() - Creates an object containing security details
createServer() - Creates a Server object

getCiphers() - Returns an array containing the supported SSL ciphers
tty - Provides classes used by a text terminal

url

The URL module provides a way of parsing the URL string.
var http = require('http');
var url = require('url');
http.createServer(function (req, res)
res.writeHead(200, {'Content-Type': 'text/plain'}); var q = url.parse(req.url, true); res.write(q.href); res.end();).listen(8080);
o/p - default.htm**url.format() -** Returns a formatted URL string
url.parse() - Returns a URL object
url.resolve() - Resolves a URL

util

The Util module provides access to some utility functions.
var util = require('util');
var txt = 'Congratulate %s on his %dth birthday!';
var result = util.format(txt, 'Linus', 6);

console.log(result);
o/p - Congratulate Linus on his 6th birthday!
debuglog() - Writes debug messages to the error object
deprecate() - Marks the specified function as deprecated
format() -Formats the specified string, using the specified arguments
inherits() - Inherits methods from one function into another
inspect() - Inspects the specified object and returns the object as a string
v8 - To access information about V8

vm

The VM module provides a way of executing JavaScript on a virtual machine, almost like eval() in JavaScript.
var vm = require('vm');
var myObj = name: 'John', age: 38;
vm.createContext(myObj);
vm.runInContext('age += 1;', myObj);
console.log(myObj);
o/p - name: 'John', age: 39

zlib

The Zlib module provides a way of zip(compress) and unzip(decompress) files.
ex Compress a file (demofile.txt) into a gzip file (mygzipfile.txt.gz):

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

var gzip = zlib.createGzip();
var r = fs.createReadStream('./demofile.txt');
var w = fs.createWriteStream('./mygzipfile.txt.gz');
r.pipe(gzip).pipe(w);
createDeflate() - Creates a Deflate object
createDeflateRaw() - Creates a DeflateRaw object
createGunzip() - Creates a Gunzip object
createGzip() - Creates a Gzip object
createInflate() - Creates a Inflate object
createInflateRaw() - Creates a InflateRaw object
createUnzip() - Creates a Unzip object
deflate() - Compress a string or buffer, using Deflate

NodeIntroduction
advanced
modules
FileSystem
eventLoop
readHtmlFileByNode
basicNode
nodeProgrammingTechniques

File System

The Node.js file system module allows you to work with the file system on your computer.
Common use for the File System module:
Read files
Create files
Update files
Delete files
Rename files

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res)
fs.readFile('demofile1.html', function(err, data) { res.writeHead(200, {'Content-Type': 'text/html'}); res.write(data); return
res.end(); });).listen(8080);
```

o/p - My Header
My paragraph.

(i)Create Files

The File System module has methods for creating new files:

(a)fs.appendFile()

method appends specified content to a file. If the file does not exist, the file will be created:

```
var fs = require('fs');
fs.appendFile('mynewfile1.txt', 'Hello content!', function (err)
if (err) throw err; console.log('Saved!'););
```

o/p-Saved!

(b)fs.open()

method takes a "flag" as the second argument, if the flag is "w" for "writing", the specified file is opened for writing. If the file does not exist, an empty file is created: Create a new, empty file using the open() method:

```
var fs = require('fs');
fs.open('mynewfile2.txt', 'w', function (err, file)
if (err) throw err; console.log('Saved!'););
```

o/p - Saved!

(c)fs.writeFile()

method replaces the specified file and content if it exists. If the file does not exist, a new file, containing the specified content, will be created:

```
var fs = require('fs');
fs.writeFile('mynewfile3.txt', 'Hello content!', function (err)
if (err) throw err; console.log('Saved!'););
```

o/p - Saved!

Update Files

The File System module has methods for updating files:

(a)fs.appendFile()

method appends the specified content at the end of the specified file: ex Append "This is my text." to the end of the file "mynewfile1.txt":

```
var fs = require('fs');
fs.appendFile('mynewfile1.txt', ' This is my text.', function (err)
if (err) throw err; console.log('Updated!'););
```

o/p - Updated!

(b)fs.writeFile()method replaces the specified file and content:

```
var fs = require('fs');
fs.writeFile('mynewfile3.txt', 'This is my text.', function (err)
if (err) throw err; console.log('Replaced!'););
```

o/p - Replaced!

(iii)Delete Files

To delete a file with the File System module, use the fs.unlink() method.
The fs.unlink() method deletes the specified file:

```
var fs = require('fs');
fs.unlink('mynewfile2.txt', function (err)
if (err) throw err; console.log('File deleted!'););
```

o/p - File deleted!**(iv)Rename Files**

To rename a file with the File System module, use the fs.rename() method.
The fs.rename() method renames the specified file:

```
var fs = require('fs');
fs.rename('mynewfile1.txt', 'myrenamedfile.txt', function (err)
if (err) throw err; console.log('File Renamed!'););
```

o/p - File Renamed!

Node.js URL Module

The URL module splits up a web address into readable parts. Parse an address with the url.parse() method, and it will return a URL object with each part of the address as properties: var url = require('url'); var adr = 'http://localhost:8080/default.htm?year=2017&month=february'; var q = url.parse(adr, true); console.log(q.host); console.log(q.pathname); console.log(q.search); var qdata = q.query; console.log(qdata.month); o/p - localhost:8080 /default ? year=2017&month=february february --Create a Node.js file that opens the requested file and returns the content to the client. If anything goes wrong, throw a 404 error:

```
var http = require('http');
var url = require('url');
var fs = require('fs');
http.createServer(function (req, res) {
```

```
var q = url.parse(req.url, true);
var filename = "." + q.pathname;
fs.readFile(filename, function(err, data) {
if (err) {
res.writeHead(404, {'Content-Type': 'text/html'});
return res.end("404 Not Found");
}
res.writeHead(200, {'Content-Type': 'text/html'});
res.write(data);
return res.end();
});
}).listen(8080);
```

o/p - Summer
I love the sun!
Winter
I love the snow!

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.

NPM creates a folder named "node_modules", where the package will be placed. All packages you install in the future will be placed in this folder.
ex Create a Node.js file that will convert the output "Hello World!" into upper-case letters:

Events

(i)Objects
in Node.js can fire events, like the readStream object fires events when opening and closing a file:

```
var fs = require('fs');
var readStream = fs.createReadStream('./demofile.txt');
readStream.on('open', function ()
console.log('The file is open'););
```

o/p - The file is open

(ii)Events Module
Node.js has a built-in module, called "Events", where you can create-, fire-, and listen for- your own events.
To include the built-in Events module use the require() method. In addition, all event properties and methods are an instance of an EventEmitter object. To be able to access these properties and methods, create an EventEmitter object:

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

(iii)The EventEmitter Object
You can assign event handlers to your own events with the EventEmitter object.
In the example below we have created a function that will be executed when a "scream" event is fired.
To fire an event, use the emit() method.

```
var events = require('events');
var eventEmitter = new events.EventEmitter();
var myEventHandler = function ()
console.log('I hear a scream!');
eventEmitter.on('scream', myEventHandler);
eventEmitter.emit('scream');
```

o/p - I hear a scream!

Upload Files

Send an Email

NodeIntroduction
advanced
modules
FileSystem
eventLoop
readHtmlFileByNode

dontBlockEventLoop

```
const n = 10;
function asyncAvg(n, avgCB) {

// Save ongoing sum in JS closure.
```

basicNode
nodeProgrammingTechniques

```
var sum = 0;
function help(i, cb) {
sum += i;
if (i == n) {
cb(sum);
return;
}

// "Asynchronous recursion".
// Schedule next operation asynchronously.
setImmediate(help.bind(null, i+1, cb));
}

// Start the helper, with CB to call avgCB.
help(1, function(sum){
var avg = sum/n;
avgCB(avg);
});
}

asyncAvg(n, function(avg){
console.log('avg of 1-n: ' + avg);
});
```

event file

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

function someAsyncOperation(callback) {
fs.readFile('text.txt', callback);
}

const timeoutScheduled = Date.now();

setTimeout(() => {
const delay = Date.now() - timeoutScheduled;

console.log(delayms have passed since I was scheduled);
}, 100);

someAsyncOperation(() => {
const startCallback = Date.now();

while (Date.now() - startCallback < 10) {
}
});
```

setImmediate callbacks are fired off the event loop, once per iteration in the order that they were queued. So on the first iteration of the event loop, callback A is fired. Then on the second iteration of the event loop, callback B is fired, then on the third iteration of the event loop callback C is fired, etc. This prevents the event loop from being blocked and allows other I/O or timer callbacks to be called in the mean time (as is the case of the 0ms timeout, which is fired on the 1st or 2nd loop iteration). nextTick callbacks, however, are always fired immediately after the current code is done executing and BEFORE going back to the event loop. In the nextTick example, we end up executing all the nextTick callbacks before ever returning to the event loop. Since setTimeout's callback will be called from the event loop, the text 'TIMEOUT FIRED' will not be output until we're done with every nextTick callback.

nextTick

```
process.nextTick(function A() {
process.nextTick(function B() {
console.log('1');

process.nextTick(function D() { console.log('2'); });
process.nextTick(function E() { console.log('3'); });
});

process.nextTick(function C() {
console.log('4');

process.nextTick(function F() { console.log('5'); });
process.nextTick(function G() { console.log('6'); });
});
});
```

```
setTimeout(function timeout() {
console.log('TIMEOUT FIRED');
}, 0)
```

process.nextTick

```
function fn(name){
return f;

function f(){
var n = name;
console.log("Next TICK "+n+", ");
}
}

function myTimeout(time,msg){
setTimeout(function(){
console.log("TIMEOUT "+msg);
},time);
}

process.nextTick(fn("ONE"));
myTimeout(0,"AFTER-ONE");
// set timeout to execute in 0 seconds for all

process.nextTick(fn("TWO"));
myTimeout(0,"AFTER-TWO");
process.nextTick(fn("THREE"));
myTimeout(0,"AFTER-THREE");
process.nextTick(fn("FOUR"));
```

when you use process.nextTick you basically ensure that the function that you pass as a parameter will be called immediately in the next tick ie. start of next event loop.So that's why all your function in next tick executes before your timer ie. setTimeout next tick doesn't mean next second it means next loop of the nodejs eventloop. Also next tick ensures that the function you are trying to call is executed asynchronously. And next tick has higher priority than your timers, I/O operations etc queued in the eventloop for execution. You should use nextTick when you want to ensure that your code is executed in next event loop instead of after a specified time. nextTick is more efficient than timers and when you want to ensure the function you call is executed asynchronously.

setImmediate

```
setImmediate(function A() {
setImmediate(function B() {
console.log('1');

setImmediate(function D() { console.log('2'); });
setImmediate(function E() { console.log('3'); });
});

setImmediate(function C() {
console.log('4');
setImmediate(function F() { console.log('5'); });
setImmediate(function G() { console.log('6'); });
});
});

setTimeout(function timeout() {
console.log('TIMEOUT FIRED');
}, 0)
```

setTimeout

```
const start = Date.now();
setTimeout(function(){
console.log(Date.now() - start);
}, 500);

for(let i=0; i<5000000; ++i){}
```

The idea of non-blocking is that the loop iterations are quick. So to iterate for each tick should take short enough a time that the setTimeout will be accurate to within reasonable precision (off by maybe 100 ms or so). In theory though you're right. If I write an application and block the tick, then setTimeouts will be delayed. So to answer you're question, who can assure setTimeouts execute on time? You, by writing non-blocking code, can control the degree of accuracy up to almost any reasonable degree of accuracy. As long as javascript is "single-threaded" in terms of code execution (excluding web-workers and the like), that will always happen. The single-threaded nature is a huge simplification

in most cases, but requires the non-blocking idiom to be successful. Try this code out either in your browser or in node, and you'll see that there is no guarantee of accuracy, on the contrary, the setTimeout will be very late:

domain_modules

```
'use strict';

const domain = require('domain');
const EE = require('events');
const fs = require('fs');
const net = require('net');
const util = require('util');
const print = process._rawDebug;

const pipeList = [];
const FILENAME = '/tmp/tmp.tmp';
const PIPENAME = '/tmp/node-domain-example-';
const FILESIZE = 1024;
let uid = 0;

// Setting up temporary resources
const buf = Buffer.alloc(FILESIZE);
for (let i = 0; i < buf.length; i++)
buf[i] = ((Math.random() * 1e3) % 78) + 48; // Basic ASCII
fs.writeFileSync(FILENAME, buf);

function ConnectionResource(c) {
EE.call(this);
this._connection = c;
this._alive = true;
this._domain = domain.create();
this._id = Math.random().toString(32).substr(2).substr(0, 8) + (++uid);

this._domain.add(c);
this._domain.on('error', () => {
this._alive = false;
});
}
util.inherits(ConnectionResource, EE);

ConnectionResource.prototype.end = function end(chunk) {
this._alive = false;
this._connection.end(chunk);
this.emit('end');
};

ConnectionResource.prototype.isAlive = function isAlive() {
return this._alive;
};

ConnectionResource.prototype.id = function id() {
return this._id;
};

ConnectionResource.prototype.write = function write(chunk) {
this.emit('data', chunk);
return this._connection.write(chunk);
};

// Example begin
net.createServer((c) => {
const cr = new ConnectionResource(c);

const d1 = domain.create();
fs.open(FILENAME, 'r', d1.intercept((fd) => {
streamInParts(fd, cr, 0);
}));

pipeData(cr);

c.on('close', () => cr.end());
```

```
}).listen(8080);

function streamInParts(fd, cr, pos) {
const d2 = domain.create();
const alive = true;
d2.on('error', (er) => {
print('d2 error:', er.message);
cr.end();
});
fs.read(fd, Buffer.alloc(10), 0, 10, pos, d2.intercept((bRead, buf) => {
if (!cr.isAlive()) {
return fs.close(fd);
}
if (cr._connection.bytesWritten < FILESIZE) {
// Documentation says callback is optional, but doesn't mention that if the write fails an exception will be thrown.
const goodtogo = cr.write(buf);
if (goodtogo) {
setTimeout(() => streamInParts(fd, cr, pos + bRead), 1000);
}
else {
cr._connection.once('drain', () => streamInParts(fd, cr, pos + bRead));
}
return;
}
cr.end(buf);
fs.close(fd);
}));
}

function pipeData(cr) {
const pname = PIPENAME + cr.id();
const ps = net.createServer();
const d3 = domain.create();
const connectionList = [];
d3.on('error', (er) => {
print('d3 error:', er.message);
cr.end();
});
d3.add(ps);
ps.on('connection', (conn) => {
connectionList.push(conn);
conn.on('data', () => {}); // don't care about incoming data.
conn.on('close', () => {
connectionList.splice(connectionList.indexOf(conn), 1);
});
});

cr.on('data', (chunk) => {
for (let i = 0; i < connectionList.length; i++) {
connectionList[i].write(chunk);
}
});

cr.on('end', () => {
for (let i = 0; i < connectionList.length; i++) {
connectionList[i].end();
}

ps.close();
});
pipeList.push(pname);
ps.listen(pname);
}

process.on('SIGINT', () => process.exit());
process.on('exit', () => {
try {
for (let i = 0; i < pipeList.length; i++) {
fs.unlinkSync(pipeList[i]);
}
```

```
fs.unlinkSync(FILENAME);
} catch (e) { }
});
```

httpStatusCode

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

http.createServer((request, response) => {
request.on('error', (err) => {
console.error(err);
response.statusCode = 400;
response.end();
});

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

if (request.method === 'POST' && request.url === '/echo') {
request.pipe(response);
}
else {
response.statusCode = 404;
response.end();
}
}).listen(8080);
```

NodeIntroduction
advanced
modules
FileSystem
eventLoop
readHtmlFileByNode
basicNode
nodeProgrammingTechniques

index.js

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

function onRequest(request, response) {
response.writeHead(200, {'Content-Type': 'text/html'});
fs.readFile('./index.html', null, function(error, data) {
if (error) {
response.writeHead(404);
response.write('File not found!');
}
else {
response.write(data);
}
response.end();
});
}

http.createServer(onRequest).listen(8000);
```

text.html

```
'<'html>
'<'body>
'<'h1>My Header'<'/h1>
'<'p>My paragraph.'<'/p>
'<'/body>
'<'/html>
```

modules
modules.js

```
exports.myDateTime = function () {
return Date();
};
```

server.js

```
var http=require('http');
var dt = require('./app');
```

```
function onRequest(request, response){
response.writeHead(200, {'Content-Type' : 'text/plain'});
response.write("The date and time are currently: " + dt.myDateTime());
response.end();
}

http.createServer(onRequest).listen(8000);
```

Routing

routeHtml.html

```
'<!doctype html>
'<html lang="en">
'<head>
'<meta charset="UTF-8">
'<title>Test Doc'</title>
'</head>
'<body>
'<h1>It works!'</h1>
'</body>
'</html>
```

routeHtml2.html

```
'<!doctype html>
'<html lang="en">
'<head>
'<meta charset="UTF-8">
'<title>Login'</title>
'</head>
'<body>
'<h1>The Login Page'</h1>
'</body>
'</html>
```

appRoute.js

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

function renderHTML(path, response) {
fs.readFile(path, null, function(error, data) {
if (error) {
response.writeHead(404);
response.write('File not found!');
}
else {
response.write(data);
}
response.end();
});
}

module.exports = {
handleRequest: function(request, response) {
response.writeHead(200, {'Content-Type': 'text/html'});

var path = url.parse(request.url).pathname;
switch (path) {
case '/':
renderHTML('./index.html', response);
break;
case '/login':
renderHTML('./login.html', response);
break;
default:
response.writeHead(404);
response.write('Route not defined');
response.end();
}
```

```
}
};
```

routeServer.js

```
var http = require('http');
var app = require('./app');

http.createServer(app.handleRequest).listen(8000);
```


DIRTy (data-intensive real-time) applications

Because Node itself is very lightweight on I/O, it's good at shuffling or proxying data from one pipe to another. It allows a server to hold a number of connections open while handling many requests and keeping a small memory footprint. It's designed to be responsive, like the browser.
example of a DIRTy application written with Node is Browserling
Node was built from the ground up to have an event-driven and asynchronous model.

Node tries to keep consistency between the browser and the server by reimple□menting common host objects, such as these:
□ Timer API (for example, setTimeout)
□ Console API (for example, console.log)
Node also includes a core set of modules for many types of network and file I/O. These include modules for HTTP, TLS, HTTPS, filesystem (POSIX), Datagram (UDP), and NET (TCP). The core is intentionally small, low-level, and uncomplicated, includ□ing just the building blocks for I/O-based applications. Third-party modules build upon these blocks to offer greater abstractions for common problems.

Node is a platform for JavaScript applications,

Streaming data
streams can be thought of as data distributed over time. By bringing data in chunk by chunk, the developer is given the ability to handle that data as it comes in instead of waiting for it all to arrive before acting.

```
var stream = fs.createReadStream('./resource.json')

Data event fires when a new chunk is ready
stream.on('data', function (chunk)
)

stream.on('end', function ()
)
```

A data event is fired whenever a new chunk of data is ready, and an end event is fired when all the chunks have been loaded. A chunk can vary in size, depending on the type of data. This low-level access to the read stream allows you to efficiently deal with data as it's read instead of waiting for it all to buffer in memory.

Node also provides writable streams that you can write chunks of data to. One of those is the response (res) object when a request happens on an HTTP server.

Building a multiroom chat application

Application overview

allows users to chat online with each other by entering messages into a simple form, A message, once entered, is sent to all other users in the same chat room. When starting the application, a user is automatically assigned a guest name, but they can change it by entering a command, Chat commands are prefaced with a slash. (/)Similarly, a user can enter a command to create a new chat room (or join it if it already exists). When joining or creating a room, the new room name will be shown in the horizontal bar at the top of the chat application. The room will also be included in the list of available rooms to the right of the chat message area. The application shows how Node can simultaneously serve conventional HTTP data (like static files) and real-time data (chat messages).

Application requirements and initial setup
To serve static files, you'll use Node's built-in http module. But when serving files via HTTP, it's usually not enough to just send the contents of a file; you also should include the type of file being sent. This is done by setting the Content-Type HTTP header with the proper MIME type for the file.

web socket.io
To handle chat-related messaging, you could poll the server with Ajax. But to make this application as responsive as possible,

you'll avoid using traditional Ajax as a means to send messages. Ajax uses HTTP as a transport mechanism, and HTTP wasn't designed for real-time communication. When a message is sent using HTTP, a new TCP/IP connection must be used. Opening and closing connections takes time, and the size of the data transfer is larger because HTTP headers are sent on every request. Instead of employing a solution reliant on HTTP, this application will prefer Web□Socket, which was designed as a bidirec□tional lightweight communications protocol to support real-time communication.
Node can easily handle simultaneously serving HTTP and WebSocket using a single TCP/IP port,

Serving HTTP and WebSocket
Although this application will avoid the use of Ajax for sending and receiving chat messages, it will still use HTTP to deliver the HTML, CSS, and client-side JavaScript needed to set things up in the user's browser.

application needs to be capable of doing three basic things:
□ Serving static files to the user's web browser
□ Handling chat-related messaging on the server
□ Handling chat-related messaging in the user's web browser

Variable declarationsvar cache = ;*cache object is where the contents of cached files are stored*

SENDING FILE DATA AND ERROR RESPONSES
Next you need to add three helper functions used for serving static HTTP files. The first will handle the sending of 404 errors when a file is requested that doesn't exist. Add the following helper function to server.js:

```
function send404(response) {
response.writeHead(404, {'Content-Type': 'text/plain'});
response.write('Error 404: resource not found.');
response.end();
}
```

The second helper function serves file data. The function first writes the appropriate HTTP headers and then sends the contents of the file. Add the following code to server.js:

Accessing memory storage (RAM) is faster than accessing the filesystem. Because of this, it's common for Node applications to cache frequently used data in memory. Our chat application will cache static files to memory, only reading them from disk the first time they're accessed. The next helper determines whether or not a file is cached and, if so, serves it. If a file isn't cached, it's read from disk and served. If the file doesn't exist, an HTTP 404 error is returned as a response. Add this helper function to server.js.

CREATING THE HTTP SERVER

For the HTTP server, an anonymous function is provided as an argument to create□Server, acting as a callback that defines how each HTTP request should be handled. The callback function accepts two arguments: request and response. When the call□back executes, the HTTP server will populate these arguments with objects that, respectively, allow you to work out the details of the request and send back a response.

```
//Create HTTP server, using anonymous function to define per-request behavior
var server = http.createServer(function(request, response) {

var filePath = false;
if (request.url == '/') {

//Determine HTML file to be served by default
filePath = 'public/index.html';
} else {

//Translate URL path to relative file path
filePath = 'public' + request.url;
var absPath = './' + filePath;

//Serve static file serveStatic(response, cache, absPath);
});
}
```

STARTING THE HTTP SERVER
```
server.listen(3000, function() );
```

Handling chat-related messaging using Socket.IO

now, second—handling communication between the browser and server. Modern browsers are capable of using WebSocket to handle communication between the browser and the server.

Socket.IO, out of the box, provides virtual channels, so instead of broadcasting every message to every connected user, you can broadcast only to those who have subscribed to a specific channel.

Event emitters
An event emitter is associated with a conceptual resource of some kind and can send and receive messages to and from the resource. The resource could be a connection to a remote server or something more abstract, like a game character.

Setting up the Socket.IO server
The first line loads functionality from a custom Node module that supplies logic to handle Socket.IO-based server-side chat functionality.
The next line starts the Socket.IO server functionality, providing it with an already defined HTTP server so it can share the same TCP/IP port:
var chatServer = require('./lib/chat_server'); chatServer.listen(server);

need to create a new file, chat_server.js, inside the lib directory. Start this file by adding the following variable declarations. These declarations allow the use of Socket.IO and initialize a number of variables that define chat state:

```
var socketio = require('socket.io');
var io;
var guestNumber = 1;
var nickNames = ;
var namesUsed = [];
var currentRoom = ;
```

ESTABLISHING CONNECTION LOGIC

Starting up a Socket.IO server

```
exports.listen = function(server) {

//Start Socket.IO server, allowing it to piggyback on existing HTTP server
io = socketio.listen(server);
io.set('log level', 1);

//Define how each user connection will be handled
io.sockets.on('connection', function (socket) {

//Assign user a guest name when they connect
guestNumber = assignGuestName(socket, guestNumber,nickNames, namesUsed);

//Place user in Lobby room when they connect
joinRoom(socket, 'Lobby');
handleMessageBroadcasting(socket, nickNames);

//Handle user messages, name□change attempts, and room creation/changes
handleNameChangeAttempts(socket, nickNames, namesUsed);
handleRoomJoining(socket);

//Provide user with list of occupied rooms on request
socket.on('rooms', function() {
socket.emit('rooms', io.sockets.manager.rooms);
});

//Define cleanup logic for when user disconnects
handleClientDisconnection(socket, nickNames, namesUsed);
});
};
```
With the connection handling established, you now need to add the individual helper functions that will handle the application's needs.

Handling application scenarios and events

The chat application needs to handle the following types of scenarios and events:
□ Guest name assignment
□ Room-change requests
□ Name-change requests
□ Sending chat messages

 Room creation
 User disconnection

Assigning a guest name

```
function assignGuestName(socket, guestNumber, nickNames, namesUsed) {
var name = 'Guest' + guestNumber;
nickNames[socket.id] = name;

socket.emit('nameResult', {
success: true,
name: name
});

namesUsed.push(name);
return guestNumber + 1;
}
```

JOINING ROOMS

```
function joinRoom(socket, room) {
socket.join(room);
currentRoom[socket.id] = room;
socket.emit('joinResult', {room: room});

socket.broadcast.to(room).emit('message', {
text: nickNames[socket.id] + ' has joined ' + room + '.' });

var usersInRoom = io.sockets.clients(room);
if (usersInRoom.length > 1) {
var usersInRoomSummary = 'Users currently in ' + room + ': ';
for (var index in usersInRoom) {
var userSocketId = usersInRoom[index].id;
if (userSocketId != socket.id) {
if (index > 0) {
usersInRoomSummary += ', ';
}
usersInRoomSummary += nickNames[userSocketId];
}
}

usersInRoomSummary += '.';
socket.emit('message', {text: usersInRoomSummary});
}
```

handle name

```
function handleNameChangeAttempts(socket, nickNames, namesUsed) {
socket.on('nameAttempt', function(name) {
if (name.indexOf('Guest') == 0) {
socket.emit('nameResult', {
success: false,
message: 'Names cannot begin with "Guest".'
});
}
else {
if (namesUsed.indexOf(name) == -1) {
var previousName = nickNames[socket.id];

var previousNameIndex = namesUsed.indexOf(previousName);
namesUsed.push(name);
nickNames[socket.id] = name;
delete namesUsed[previousNameIndex];

socket.emit('nameResult', {
success: true,
name: name
});

socket.broadcast.to(currentRoom[socket.id]).emit('message', {
text: previousName + ' is now known as ' + name + '.' });
}
else {
socket.emit('nameResult', {
success: false,
message: 'That name is already in use.'
```

});
}
}
});
*}***SENDING CHAT MESSAGES**
function handleMessageBroadcasting(socket) {
socket.on('message', function (message) {
socket.broadcast.to(message.room).emit('message', {
text: nickNames[socket.id] + ': ' + message.text
});
});
} **HANDLING USER DISCONNECTIONS**
function handleClientDisconnection(socket) {
socket.on('disconnect', function() {
var nameIndex = namesUsed.indexOf(nickNames[socket.id]);
delete namesUsed[nameIndex];
delete nickNames[socket.id];
});
*}***SENDING CHAT MESSAGES**
function handleMessageBroadcasting(socket) {
socket.on('message', function (message) {
socket.broadcast.to(message.room).emit('message', {
text: nickNames[socket.id] + ': ' + message.text
});
});
}

Using client-side JavaScript for the application's user interface

it's time to add the client-side JavaScript needed to communicate with the server. Client-side JavaScript is needed to handle the following functionality:
Sending a user's messages and name/room change requests to the server
 Displaying other users' messages and the list of available rooms

NodeIntroduction
advanced
modules
FileSystem
eventLoop
readHtmlFileByNode
basicNode
nodeProgrammingTechniques

The Node convention for asynchronous callbacks

Most Node built-in modules use callbacks with two arguments: the first argument is for an error, should one occur, and the second argument is for the results.
Here's a typical example of this common function signature:

var fs = require('fs');
fs.readFile('./titles.json', function(er, data)
if (er) throw er;);

Handling repeating events with event emitters
Event emitters fire events and include the ability to handle those events when trig gered. Some important Node API components, such as HTTP servers, TCP servers, and streams, are implemented as event emitters. You can also create your own. As we mentioned earlier, events are handled through the use of listeners. A listener is the association of an event with a callback function that gets triggered each time the event occurs. For example, a TCP socket in Node has an event called data that's trig gered whenever new data is available on the socket:
socket.on('data', handleData);
The socket is an event emitter to which you can then add a listener, using the on method, to respond to data events. These data events are emitted whenever new data is available on the socket.

var net = require('net');
var server = net.createServer(function(socket)
{ socket.on('data', function(data) {
socket.write(data);
}) });

server.listen(8888);
RESPONDING TO AN EVENT THAT SHOULD ONLY OCCUR ONCE
The code in the following listing, using the once method, modifies the previous echo server example to only echo the first chunk of data sent to it.
var net = require('net');
var server = net.createServer(function(socket)
{ socket.once ('data', function(data) {
socket.write(data);
}) });

server.listen(8888);

CREATING EVENT EMITTERS: A PUB/SUB EXAMPLE

The following code defines a channel event emitter with a single listener that responds to someone joining the channel. Note that you use on to add a listener to an event emitter:

var EventEmitter = require('events').EventEmitter;
var channel = new EventEmitter();

channel.on('join', function()
{ console.log("Welcome!") });

This join callback, however, won't ever be called, because you haven't emitted any events yet. You could add a line to the listing that would trigger an event using the emit function:

channel.emit('join');

A simple publish/subscribe system using an event emitter

var events = require('events');
var net = require('net');
var channel = new events.EventEmitter();
channel.clients = {};
channel.subscriptions = {};

//Add a listener for the join event that stores a user's client object, allowing the application to send data back to the user.
channel.on('join', function(id, client) {
this.clients[id] = client;
this.subscriptions[id] = function(senderId, message) {

//Ignore data if it's been directly broadcast by the user. if (id != senderId) {
this.clients[id].write(message);
}
}

//Add a listener, specific to the current user, for the broadcast event. this.on('broadcast', this.subscriptions[id]);
});

var server = net.createServer(function (client) {
var id = client.remoteAddress + ':' + client.remotePort;
client.on('connect', function() {

// Emit a join event when a user connects to the server, specifying the user ID and client object. channel.emit('join', id, client);
});
client.on('data', function(data) {
data = data.toString();

//Emit a channel broadcast event, specifying the user ID and message, when any user sends data. channel.emit('broadcast', id, data);
});
});

server.listen(8888);

NodeIntroduction
advanced
modules
FileSystem
eventLoop
readHtmlFileByNode
basicNode
nodeProgrammingTechniques

Listing currently installed packages

ls, la and ll are aliases of list command. la and ll commands shows extended information like description and repository.

npm list --json

json - Shows information in json format
long - Shows extended information
parseable - Shows parseable list instead of tree
global - Shows globally installed packages
depth - Maximum display depth of dependency tree
dev/development - Shows devDependencies
prod/production - Shows dependencies

Updating npm and packages

npm install -g npm@latest

If you want to check for updated versions you can do:
npm outdated

In order to update a specific package:
npm update package name