

Node.js

Node js is not a language

This is server environment

Node js can connect with database

code and syntax is very similar to javascript

Why do we use Node js

Node js is mostly used for API

so that we can connect to database through API in Mobile App and Web App.

Node is easy to understand who know javascript

Node is superfast for APIs

Javascript and Node are the same?

Javascript and Node js code syntax is same

If you know javascript you can easily understand Nodejs.

But both are not exactly same.

Introduction

- Node.js is an open-source and cross-platform JavaScript runtime environment.
- Node.js runs the V8 JavaScript engine, the core of Google Chrome.
- A Node.js app runs in a single process, without creating a new thread for every request.
- Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking.
- When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.
- This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

- Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code

Features of Node.js

- **Asynchronous and Event Driven** – All APIs of Node.js library are asynchronous, that is, non-blocking.
- It essentially means a Node.js based server never waits for an API to return data.
- The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call.
- **Very Fast** – Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.

- **Single Threaded but Highly Scalable** – Node.js uses a single threaded model with event looping.
- Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests.
- Node.js uses a single threaded program and the same program can provide service to a much larger number of requests than traditional servers like Apache HTTP Server.

- **No Buffering** – Node.js applications never buffer any data.
- These applications simply output the data in chunks.

Verify the installation

- Create a js file named main.js on your machine (Windows or Linux) having the following code.
- `console.log("Hello, World!")`
- Now execute main.js using Node.js interpreter to see the result:
- `$ node main.js`
- If everything is fine with your installation, it should produce the following result.
- Hello, World!

- Check the node version
- Node -v
- `Console.log("hi");`
- `Var x=10`
- `Console.log(x);`
- `Var y=20;`
- `Console.log(x+y);`

```
var a=20;
```

```
var b=30;
```

```
var c=40;
```

```
console.log(a+b+c);
```

```
var a=20;
```

```
let b=30;
```

```
const c=40;
```

```
console.log(a+b+c);
```

```
const arr=[2,3,5,1,6];
```

```
console.log(arr);
```

```
console.log(arr[0]);
```

Try1.js

```
export let x=10;
```

```
export let y=20;
```

try.js

```
import {x} from './try1'
```

```
const arr=[2,3,5,1,6];
```

```
console.log(arr);
```

SyntaxError: Cannot use import statement outside a module

Node js doesn't support export import statements

try1.js

```
module.exports={  
  x:10,  
  y:20  
}
```

try.js

```
const try1=require('./try1')  
const arr=[2,3,5,1,6];  
console.log(arr);  
console.log(try1.x);
```

Module in Node js

Core Modules:

Two types of Modules:

1 Global

modules we are using but no need of doing import like

2 Non Global

Compulsory we need to import these modules then only we can use it

```
const fs=require('fs')  
  
fs.writeFileSync("hello.txt","Good Day!!!");
```



```
var http = require("http");  
http.createServer(function (request, response) {  
    response.write('hello,TSEC');  
response.end();  
}).listen(8081);
```

```
var http = require("http");  
http.createServer(function (request, response) {  
    response.write("<h1>hello,TSEC</h1>");  
response.end();  
}).listen(8081);
```

Where to use Node.js?

- REPL

REPL

- REPL stands for Read Eval Print Loop and it represents a computer environment like a Windows console or Unix/Linux shell where a command is entered and the system responds with an output in an interactive mode.
- Node.js or **Node** comes bundled with a REPL environment.
- It performs the following tasks –
- **Read** – Reads user's input, parses the input into JavaScript data-structure, and stores in memory.
- **Eval** – Takes and evaluates the data structure.
- **Print** – Prints the result.
- **Loop** – Loops the above command until the user exits
- The REPL feature of Node is very useful in experimenting with Node.js codes and to debug JavaScript codes.

- Starting REPL REPL can be started by simply running node on shell/console without any arguments as follows.
- `$ node`
- You will see the REPL Command prompt `>` where you can type any Node.js command:
- `$ node`
- `>`

Variables

- You can make use variables to store values and print later like any conventional script.
- If **var** keyword is not used, then the value is stored in the variable and printed.
- Whereas if **var** keyword is used, then the value is stored but not printed.
- You can print variables using **console.log()**.

```
$ node
```

```
> x = 10
```

```
10
```

```
> var y = 10
```

```
undefined
```

```
> x + y
```

```
20
```

```
> console.log("Hello World")
```

```
Hello World
```

```
undefined
```

Multiline Expression

- Node REPL supports multiline expression similar to JavaScript

```
$ node  
> var x = 0  
undefined  
> do {  
... x++;  
... console.log("x: " + x);  
... } while ( x < 5 );  
x: 1  
x: 2  
x: 3  
x: 4  
x: 5
```


Underscore variable

- You can use underscore (`_`) to get the last result.

```
$ node
```

```
> var x = 10
```

```
undefined
```

```
> var y = 20
```

```
undefined
```

```
> x + y
```

```
30
```

```
> var sum = _
```

```
undefined
```

```
> console.log(sum)
```

```
30
```

```
undefined
```

REPL Commands

- **ctrl + c** – terminate the current command.
- **ctrl + c twice** – terminate the Node REPL.
- **ctrl + d** – terminate the Node REPL.
- **Up/Down Keys** – see command history and modify previous commands.
- **tab Keys** – list of current commands.
- **.help** – list of all commands.
- **.break** – exit from multiline expression.
- **.clear** – exit from multiline expression.
- **.save filename** – save the current Node REPL session to a file.
- **.load filename** – load file content in current Node REPL session.

nodemon

- When we are updating the program and saving the changes.
- After refreshing the browser the changes are reflected.
- Every time we need to restart the server.
- To save this step we can install
- **npm i nodemon -g**
- Now write nodemon progname.js
- Nodemon runs the nodejs project continuously.
- No need to execute the project everytime.

Event Loop

- Nodejs application is single threaded.
- It just contain only one main thread.
- If you perform long running operations on this main thread then it will block your UI.
- So to perform long operations asynchronously we use vent loop.

```
console.log('Event Loop')
```

- Internally node starts a node process and using that it simply prints the above statement.
- When the execution of script is over it terminates the process.
- Note: Node process executes the script and immediately stops once all the code is executed.

```
console.log('starting up')
```

```
setTimeout(()=>{  
    console.log("2 sec log")  
},2000);
```

```
setTimeout(()=>{  
    console.log("0 sec log")  
},0);
```

```
console.log("finishing up")
```

Event Emitter

- Node.js allows us to create and handle custom events easily by using events module.
- Event module includes EventEmitter class which can be used to raise and handle custom events.
- The on() method requires name of the event to handle and callback function which is called when an event is raised.
- The emit() function raises the specified event.
- First parameter is name of the event as a string and then arguments.
- An event can be emitted with zero or more arguments.
- You can specify any name for a custom event in the emit() function.


```
const express =require('express');
const EventEmitter=require('events');
const app=express()
const event= new EventEmitter();
event.on("count API",()=>{
    count++;
    console.log("Event Called", count)
})
app.get ("/",( req, res)=>{
    res.send("api called")
    event.emit("count API")
});
```

```
app.get ("/search",( req, res)=>{
    res.send("search api called")
});
app.get ("/update",( req, res)=>{
    res.send("update api called")
});

app.listen(5000)
```

```
const EventEmitter= require('events')
const myEmitter= new EventEmitter()
```

```
myEmitter.on('Test',()=>{
  console.log("Test was fired")
})
myEmitter.on('Test',()=>{
  console.log("Test was fired")
})
myEmitter.on('Test',()=>{
  console.log("Test was fired")
})
myEmitter.emit('Test')
```

```
var event=require('events')
var eventemitter=new event.EventEmitter();
eventemitter.on("speak",function(name){
  console.log(name, "is speaking")
})
eventemitter.emit("speak","John")
```

How to read a file:

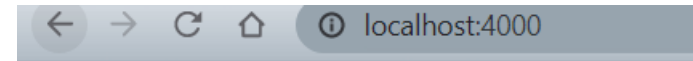
```
var http=require('http');
var fs= require('fs')
http.createServer((req,res)=>{

    fs.readFile('client.html',function(err,data){
        res.writeHead(200,{ 'content-type': 'text/html' })
        res.write(data)
        return res.end()
    })
}).listen(4000);
```

Client.html

```
<html>
  <head>
    <title>Sample Page</title>
  </head>

  <body>
    Hello World!
    Good morning
  </body>
</html>
```



Hello World! Good morning

Open a file

- `fs.open(path, flags[, mode], callback)`
- **path** – This is the string having file name including path.
- **flags** – Flags indicate the behavior of the file to be opened.
- **mode** – It sets the file mode (permission and sticky bits), but only if the file was created. It defaults to 0666, readable and writeable.
- **callback** – This is the callback function which gets two arguments (err, fd).

1	r Open file for reading. An exception occurs if the file does not exist.
2	r+ Open file for reading and writing. An exception occurs if the file does not exist.
3	rs Open file for reading in synchronous mode.
4	rs+ Open file for reading and writing, asking the OS to open it synchronously.
5	w Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
6	wx Like 'w' but fails if the path exists. The file is opened in exclusive mode. Exclusive mode ensures that the files are newly created.

7	w+ Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
8	wX+ Like 'w+' but fails if path exists.
9	a Open file for appending. The file is created if it does not exist.
10	ax Like 'a' but fails if the path exists.
11	a+ Open file for reading and appending. The file is created if it does not exist.
12	ax+ Like 'a+' but fails if the the path exists.

Get the file information

- `fs.stat(path, callback)`
- **path** – This is the string having file name including path.
- **callback** – This is the callback function which gets two arguments (`err`, `stats`) where **stats** is an object of `fs.Stats` type
- 1. `stats.isFile()`
- Returns true if file type of a simple file.
- 2. `stats.isDirectory()`
- Returns true if file type of a directory.
- 3. `stats.isBlockDevice()`
- Returns true if file type of a block device.


```
var fs = require("fs");
console.log("Going to get file info!");
fs.stat('input.txt', function (err, stats)
{
    if (err) {
        return console.error(err);
    }
    console.log(stats);
    console.log("Got file info
successfully!");
```

```
// Check file type
    console.log("isFile ? " +
stats.isFile());
    console.log("isDirectory ? " +
stats.isDirectory());
});
```

Writing a File

- `fs.writeFile(filename, data[, options], callback)`
- This method will over-write the file if the file already exists.
- **path** – This is the string having the file name including path.
- **data** – This is the String or Buffer to be written into the file.
- **options** – The third parameter is an object which will hold {encoding, mode, flag}.
- By default. encoding is utf8, mode is octal value 0666. and flag is 'w'
- **callback** – This is the callback function which gets a single parameter `err` that returns an error in case of any writing error.

- `var fs = require("fs");`
- `console.log("Going to write into existing file");`
- `fs.writeFile('input.txt', 'Hello! We are at the end of node module!!!!!!!!!!!!!!', function(err) {`
- `if (err) {`
- `return console.error(err);`
- `}`
- `console.log("Data written successfully!");`
- `console.log("Let's read newly written data");`
- `fs.readFile('input.txt', function (err, data) {`
- `if (err) {`
- `return console.error(err);`
- `}`
- `console.log("Asynchronous read: " + data.toString());`
- `});`
- `});`

Reading a File:

- `fs.read(fd, buffer, offset, length, position, callback)`
- This method will use file descriptor to read the file.
- **fd** – This is the file descriptor returned by `fs.open()`.
- **buffer** – This is the buffer that the data will be written to.
- **offset** – This is the offset in the buffer to start writing at.
- **length** – This is an integer specifying the number of bytes to read.
- **position** – This is an integer specifying where to begin reading from in the file. If position is null, data will be read from the current file position.
- **callback** – This is the callback function which gets the three arguments, (err, bytesRead, buffer).

- `var fs = require("fs");`
- `var buf = new Buffer(1024);`
- `console.log("Going to open an existing file");`
- `fs.open('input.txt', 'r+', function(err, fd) {`
- `if (err) {`
- `return console.error(err);`
- `}`
- `console.log("File opened successfully!");`
- `console.log("Going to read the file");`
- `fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){`
- `if (err){`
- `console.log(err);`
- `}`
- `console.log(bytes + " bytes read");`
- `// Print only read bytes to avoid junk.`
- `if(bytes > 0){`
- `console.log(buf.slice(0, bytes).toString());`
- `}`
- `});`
- `});`

Closing a File

- `fs.close(fd, callback)`
- **fd** – This is the file descriptor returned by file `fs.open()` method.
- **callback** – This is the callback function. No arguments other than a possible exception are given to the completion callback.

- `var fs = require("fs");`
- `var buf = new Buffer(1024);`
- `console.log("Going to open an existing file");`
- `fs.open('input.txt', 'r+', function(err, fd) {`
- `if (err) {`
- `return console.error(err); }`
- `console.log("File opened successfully!");`
- `console.log("Going to read the file");`
- `fs.read(fd, buf, 0, buf.length, 0, function(err, bytes) {`
- `if (err) {`
- `console.log(err); }`
- `// Print only read bytes to avoid junk.`
- `if(bytes > 0) {`
- `console.log(buf.slice(0, bytes).toString());}`
- `// Close the opened file.`
- `fs.close(fd, function(err) {`
- `if (err) { console.log(err); }`
- `console.log("File closed successfully.");`
- `});`
- `});`
- `});`

Delete a File

- `fs.unlink(path, callback)`
- **path** – This is the file name including path.
- **callback** – This is the callback function. No arguments other than a possible exception are given to the completion callback.

- `var fs = require("fs");`
- `console.log("Going to delete an existing file");`
- `fs.unlink('input.txt', function(err) {`
- `if (err) {`
- `return console.error(err);`
- `}`
- `console.log("File deleted successfully!");`
- `});`

Node Js Net Module

- Node.js **net** module is used to create both servers and clients.
- This module provides an asynchronous network wrapper and it can be imported using the following syntax.
- `Var net= require('net');`
- Node.js has a 'net' module which provides an asynchronous network API for creating stream-based TCP/IPC servers and clients.

```
server.listen(9000, () => { console.log('opened server on port: ', 9000); });
```

- To create a TCP/IPC based server, we use the `createServer` method.
- `Var server=net.createServer();`
- The server object is of type `net.server`.
- the method needed is 'listen' which starts the server for listening to connections in async, firing the 'listening' event.
- `Server.listen(9000,()=>{ console.log('opened server on port',9000); })`

- To find out on which address a server is running, we can use the `address()` method on the `net.Server` instance.
- If we need to log the port on which the server is running, then we can get this info as well without hardcoding.
- `Server.listen(9000,()=>{ console.log('opened server on %j', server.address().port)`
- The first parameter of `listen` is the port in which the server starts listening, and a callback which gets called once it has started listening.
- A few of the common errors raised are:
 - `ERR_SERVER_ALREADY_LISTEN` – server is already listening and hasn't been closed.
 - `EADDRINUSE` – another server is already listening on the given port/handle/path.
- Whenever an error happens, an 'error' event is raised. We can hook to it and capture the errors accordingly.

- `server.on('error', (e) => {`
- `if (e.code === 'EADDRINUSE') {`
- `console.log('Address in use, retrying...');`
- `setTimeout(() => {`
- `server.close();`
- `server.listen(PORT, HOST);`
- `}, 1000);`
- `}`
- `});`

- Whenever a client connects to this server then a 'connection' event is raised and in the callback we can get hold of the client object for communicating data.
- `server.on("connection", (socket) => {`
- `console.log("new client connection is made");`
- `});`

- The second parameter is actually a callback which has the reference to the connection object, and the client object is of type 'net. Socket'.
- To get the details like address and port, we can rely on remoteAddress, and remotePort properties respectively.

- `server.on("connection", (socket) => {`
- `console.log("Client connection details - ", socket.remoteAddress +
 ":" + socket.remotePort);`
- `});`

- if there is any data being sent by client, we can capture that data on the server by subscribing to 'data' event on the client socket object
- Some of the most commonly used events on 'net.Socket' are data, error and close.
- data is for listening to any data sent,
- error when there is an error happens and
- close event is raised when a connection is closed which happens once.

Server.js

```
const net=require('net')
const server = net.createServer();
server.on("connection",(socket)=>{
    console.log("new client connection is
made",socket.remoteAddress+": "+socket.remotePort);
    socket.on("data",(data)=>{
        console.log(data.toString());
    })
    socket.once("close",()=>{
        console.log("Client connection is closed");
    })
    socket.on("error",(err)=>{
        console.log("error")
    })
    socket.write('server:Hello connection successfully made<br>')
})
```

```
server.on("error",(err)=>{
  if(err=== 'EADDRINUSE')
  {
    console.log('Address in use')
    setTimeout(()=>{
      server.close()
      server.listen(PORT,HOST)
    },1000)
  }
  else{
    console.log('server failed')}
})
server.listen(9000,()=>{
  console.log('opened server on %j',server.address().port)
})
```

```
const net = require('net');
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
const client = net.createConnection({ port: 9000 }, () => {
  console.log('CLIENT: I connected to the server.');
  client.write('CLIENT: Hello this is client!');
});
client.on('data', (data) => {
  console.log(data.toString());
  //client.end();
});
client.on('end', () => {
  console.log('CLIENT: I disconnected from the server.');
})
rl.on('line', (input) => {
  client.write(`CLIENT: ${input}`);
});
```

PROBLEMS OUTPUT TERMINAL JUPYTER DEBUG CONSOLE

```
PS C:\Users\admin\Documents\tenode\node\hello> node net1.js
opened server on 9000
new client connection is made ::ffff:127.0.0.1:58069
CLIENT: Hello this is client!
CLIENT: hi
CLIENT: How r u doing?
█
```

```
PS C:\Users\admin\Documents\tenode\node\hello> node cli.js
CLIENT: I connected to the server.
server:Hello connection successfully made<br>
> hi
How r u doing?
█
```

Buffers

- A buffer is a space in memory (typically RAM) that stores binary data.
- In Node.js, we can access these spaces of memory with the built-in Buffer class.
- Buffers store a sequence of integers, similar to an array in JavaScript.
- Unlike arrays, you cannot change the size of a buffer once it is created.

- when you read from a file with `fs.readFile()`, the data returned to the callback or Promise is a buffer object.
- Additionally, when HTTP requests are made in Node.js, they return data streams that are temporarily stored in an internal buffer when the client cannot process the stream all at once.
- Buffers are useful when you're interacting with binary data, usually at lower networking levels.
- They also equip you with the ability to do fine-grained data manipulation in Node.js.

Creating a buffer

- Node Buffer can be constructed in a variety of ways.
- Method 1 :syntax to create an uninitiated Buffer of 10 octets –
 - `var buf = new Buffer(10);`
- Method 2: syntax to create a Buffer from a given array –
 - `var buf = new Buffer([10, 20, 30, 40, 50]);`
- Method 3: syntax to create a Buffer from a given string and optionally encoding type –
 - `var buf = new Buffer("New Buffer", "utf-8");`

Writing to a buffer

- syntax of the method to write into a Node Buffer –
- `buf.write(string[, offset][, length][, encoding])`
- Parameters:
- `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 : 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.

Writing to a buffer

```
buf = new Buffer(256);  
len = buf.write("Hello TSEC");  
  
console.log("Octets written : "+ len);
```

Reading from a buffer

- `buf.toString([encoding][, start][, end])`
- Parameters:
- `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.

- The buffer object comes with the `toString()` and the `toJSON()` methods, which return the entire contents of a buffer in two different formats.
- the `toString()` method converts the bytes of the buffer into a string and returns it to the user.
- `hiBuf.toString();`

Streams

- Stream:
- Streams are **objects that allows developers to read/write data to and from a source in a continuous manner.**
- There are four main types of streams in Node.js; readable, writable, duplex and transform.
- Each stream is an EventEmitter instance that emits different events at several intervals.
- Node.js streams are based on Eventemitters.
- Stream.on['data']
- Stream.on['readable']
- Stream.on['close']
- Stream.on['end']
- Stream.on['finish']
- Stream.destroy[]

- The readable stream is a stream that is used for read operations.
- The writable stream is a stream used for write operations.
- A duplex stream is a stream that performs both read and write operations.
- A transform stream is a stream that uses its input to compute an output.
- The streams throw several events since they are EventEmitter instances.
- These events are used to track and monitor the stream.

Reading Streams

```
var fs=require('fs')
var readStream=fs.createReadStream('output.txt','utf-8')
var data='';
readStream.on('data',function(chunk){
    console.log('-----')
    data+=chunk;
    // console.log(chunk);
})
readStream.on('end',function(chunk){
    console.log(data)
    console.log('-----End-----')
})
```

Writing to a stream

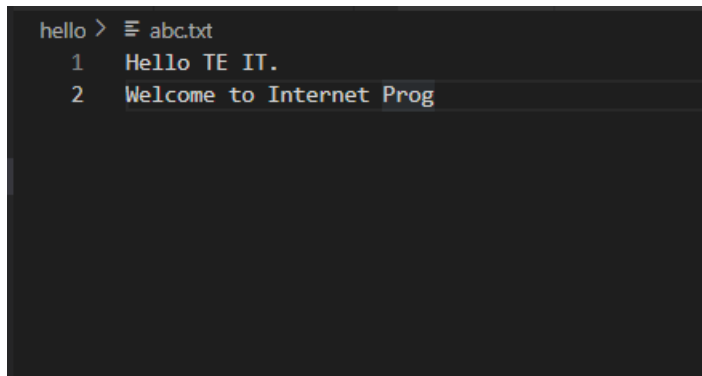
- The procedure is similar to the one of reading.
- The only difference is that this time we are supposed to create a write stream instead.

- `const fileSystem = require("fs");`
- `var data = "Sample text";`
- `const writeStream = fileSystem.createWriteStream("output.txt");`
- `writeStream.write(data, "UTF8");`
- `writeStream.end()`
- `writeStream.on("finish", () => {`
 - `console.log("Finished writing");`
- `});`
- `writeStream.on("error", (error) => {`
 - `console.log(error.stack);`
- `});`

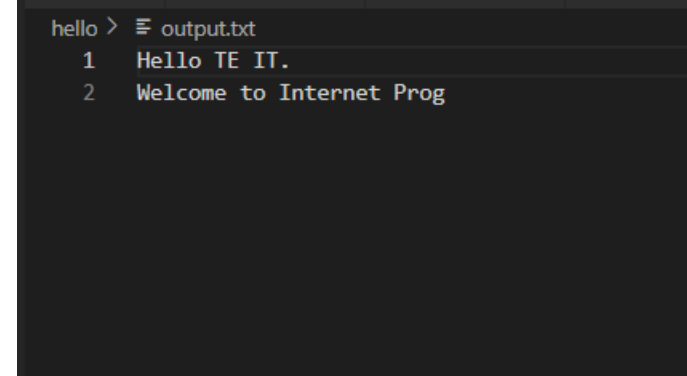
Piping the streams

- 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

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



A terminal window with a dark background. The prompt is 'hello >'. To its right is a file icon and the text 'abc.txt'. Below this, there are two lines of text: '1 Hello TE IT.' and '2 Welcome to Internet Prog'.



A terminal window with a dark background. The prompt is 'hello >'. To its right is a file icon and the text 'output.txt'. Below this, there are two lines of text: '1 Hello TE IT.' and '2 Welcome to Internet Prog'.

Chaining the streams

- 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.

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

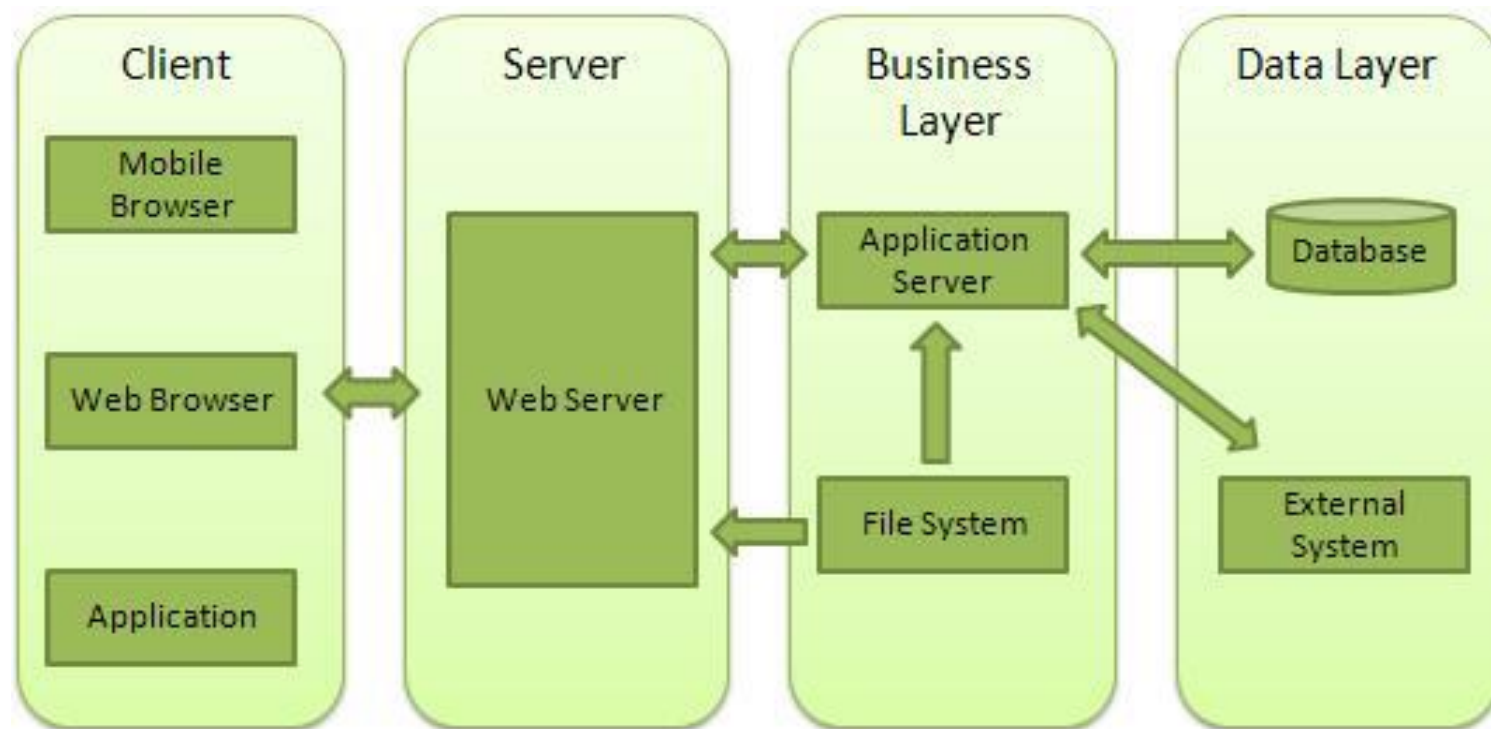
- `var fs = require("fs");`
- `var zlib = require('zlib');`
- `// Decompress the file input.txt.gz to input.txt`
- `fs.createReadStream('input.txt.gz')`
- `.pipe(zlib.createGunzip())`
- `.pipe(fs.createWriteStream('input.txt'));`
-
- `console.log("File Decompressed.");`

Web Module

- A Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients.
- Web servers usually deliver html documents along with images, style sheets, and scripts.
- Most of the web servers support server-side scripts, using scripting languages or redirecting the task to an application server which retrieves data from a database and performs complex logic and then sends a result to the HTTP client through the Web server.
- Apache web server is one of the most commonly used web servers.

Web Application Architecture

- A web Application is usually divided into 4 layers.
- **Client** – This layer consists of web browsers, mobile browsers or applications which can make HTTP requests to the web server.
- **Server** – This layer has the Web server which can intercept the requests made by the clients and pass them the response.
- **Business** – This layer contains the application server which is utilized by the web server to do the required processing. This layer interacts with the data layer via the database or some external programs.
- **Data** – This layer contains the databases or any other source of data.



Creating a web server using node

- Node.js provides an **http** module which can be used to create an HTTP client or a server.

- `var http = require('http');`
- `var fs = require('fs');`
- `var url = require('url');`
- `http.createServer(function (request, response) {`
- `// Parse the request containing file name`
- `var pathname = url.parse(request.url).pathname;`
- `// Print the name of the file for which request is made.`
- `console.log("Request for " + pathname + " received.");`
- `// Read the requested file content from file system`
- `fs.readFile(pathname.substr(1), function (err, data) {`
- `if (err) {`
- `console.log(err);`
- `// HTTP Status: 404 : NOT FOUND`
- `// Content Type: text/plain`
- `response.writeHead(404, {'Content-Type': 'text/html'});`
- `} else {`
- `//Page found`
- `// HTTP Status: 200 : OK`
- `// Content Type: text/plain`
- `response.writeHead(200, {'Content-Type': 'text/html'});`
- `// Write the content of the file to response body`
- `response.write(data.toString());`
- `}`
- `// Send the response body`
- `response.end();`
- `});`
- `}).listen(8081);`
- `// Console will print the message`

Creating a web client using node

- A web client can be created using **http** module.

- `var http = require('http');`
- `// Options to be used by request`
- `var options = {`
- `host: 'localhost',`
- `port: '8081',`
- `path: '/index.htm' };`
- `// Callback function is used to deal with response`
- `var callback = function(response) {`
- `// Continuously update stream with data`
- `var body = '';`
- `response.on('data', function(data) {`
- `body += data; });`
- `response.on('end', function() {`
- `// Data received completely.`
- `console.log(body);`
- `}); }`
- `// Make a request to the server`
- `var req = http.request(options, callback);`
- `req.end();`