Advanced Node.js

Software requirements

* Node.js (LTS)
* VS Code

Node.js:

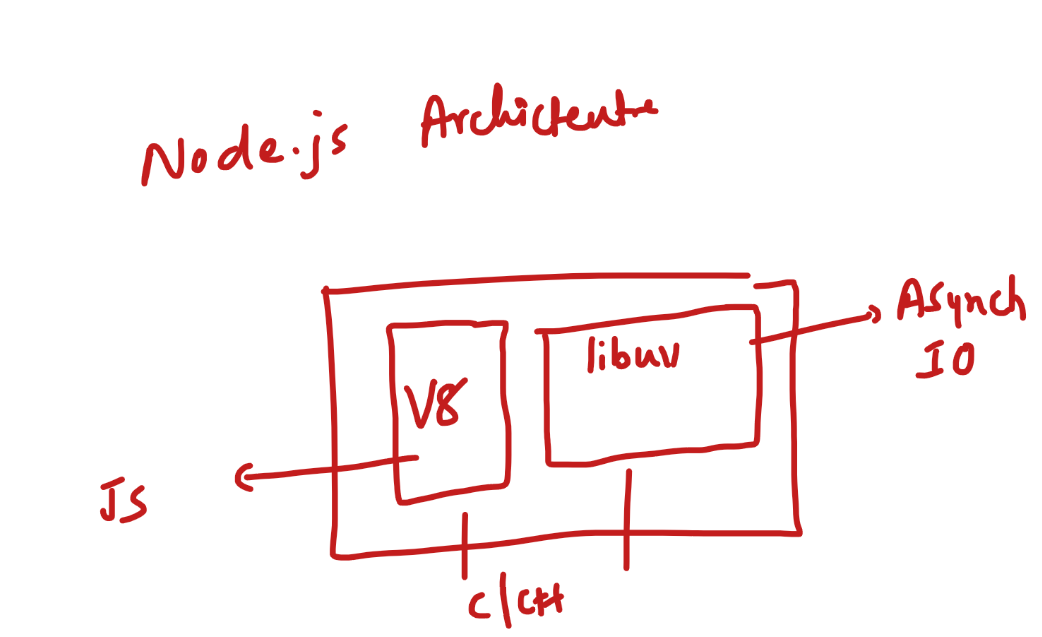
It is a runtime environment to run the javascript outside the browser

It helps the web developers who already have idea on Javascript to use it to implement various backend programs

1. Handling File Systems
2. Handling OS Resources
3. Accessing the Database
4. Networking

>> node file-name (or) node file-name.js

Node.js Architecture

V8 Engine: It takes care of running the Javascript code, V8 engine was part of Google Chrome

libuv: It takes care of Asynchronous IO operations

What are IO Operations

IO operations are those which performs CPU intensive tasks like:-

1. Accessing Network (TCP/UDP)
2. Accessing DB
3. Accessing files system
4. Timer

Synchronous vs Asynchronous

1. Synchronous: The logics are executed in sequence, where current statement depends on previous statement execution
2. Asynchronous: The logics are executed independently without blocking the primary program execution flow, to handle the results from the asynchronous operations we use some mechanisms in Javascript like callback functions

V8 engine & libuv both are implemented in C/C++ programs,

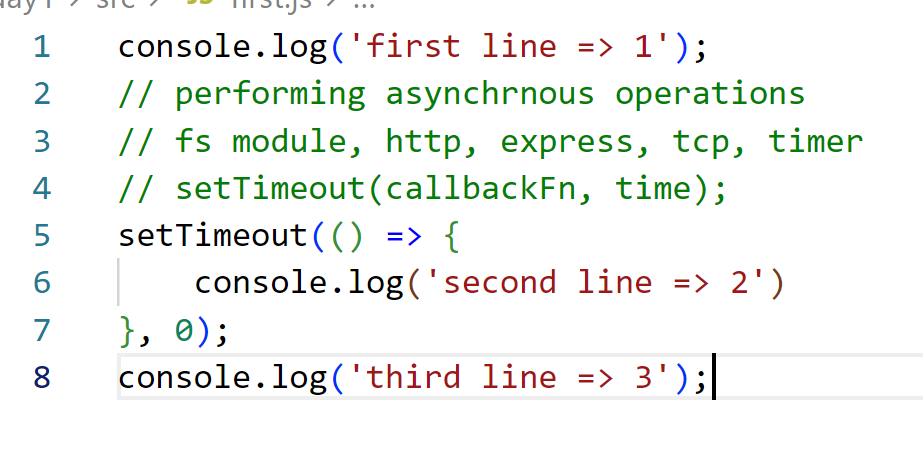
C++ addson: it provides a facility for a Javascript to run the C++ code by providing the API’s in Javascript

V8 engine will have callstack – which runs your javascript code

libuv will have event loop, event queue & other libraries to handle asynchronous operations.

Event loop checks whether the callstack is empty or not, if its empty it pushes the task present in the queue for execution.

first.js



Here the V8 runs the console.log(1) & console.log(3) first because setTimeout is handled by libuv, which adds the callback to the queue, since event loop waits for callstack to be empty, the callback in the queue is executed only after console.log(3) even though the timer is 0s.

Modules in Node.js

Reusable functionalities which you can export & import in your javascript, it could be variables, functions or classes

a.js

let module.exports.done = true;

b.js

let a = require(“./a.js”); // a.js module will be included

console.log(a.done);

Understanding how to include modules & require()

a.js

exports.done = false;

let b = require(“./b.js”);

b.js

let a = require(“./a.js”);

// some statements

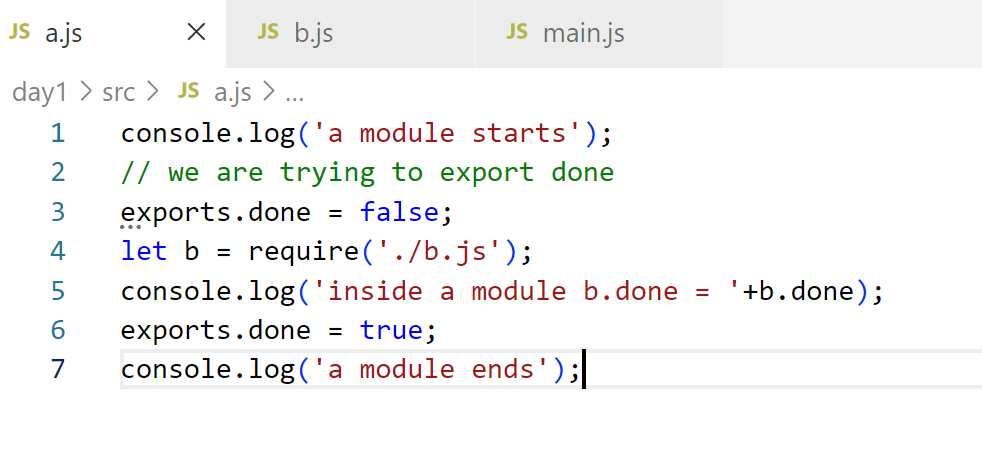
main.js

let a = require(“./a.js”);  
let b = require(“./b.js”);

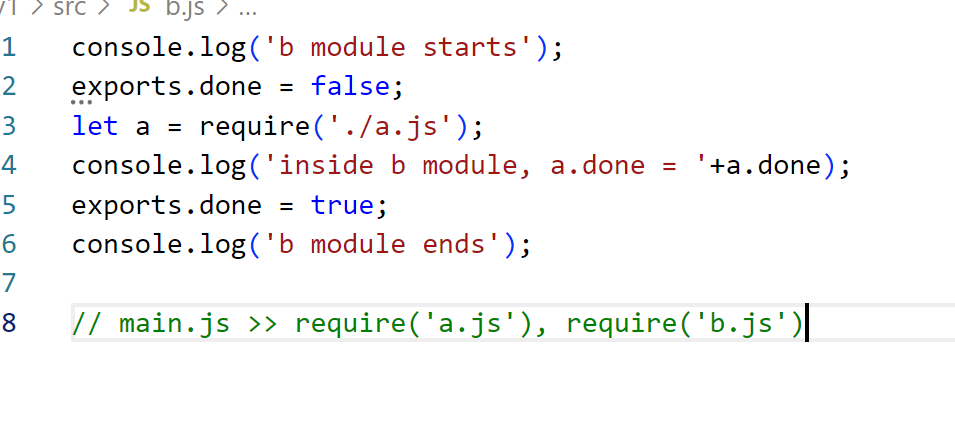
console.log(a.done);

cyclic require

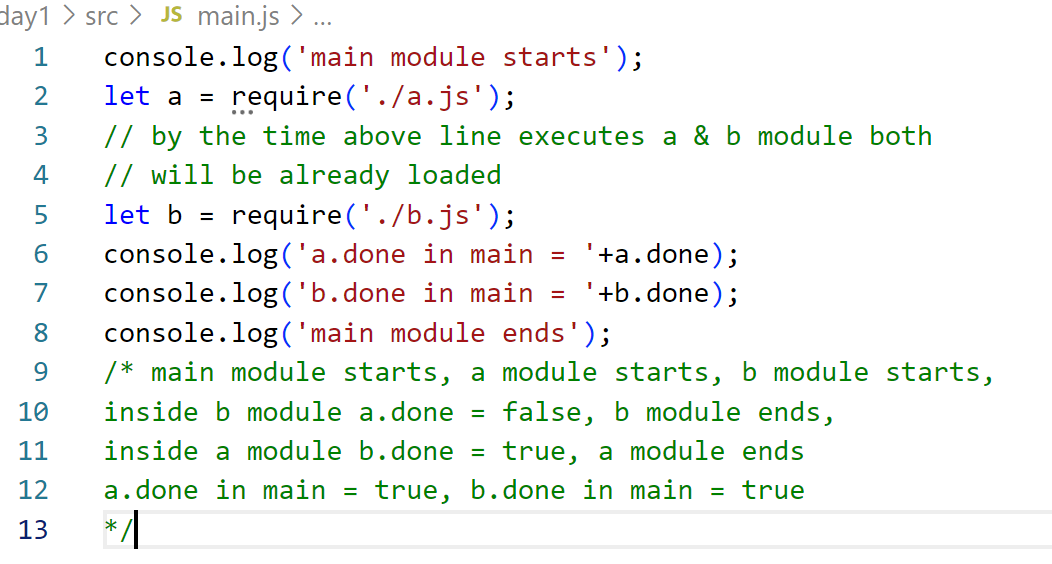
a.js



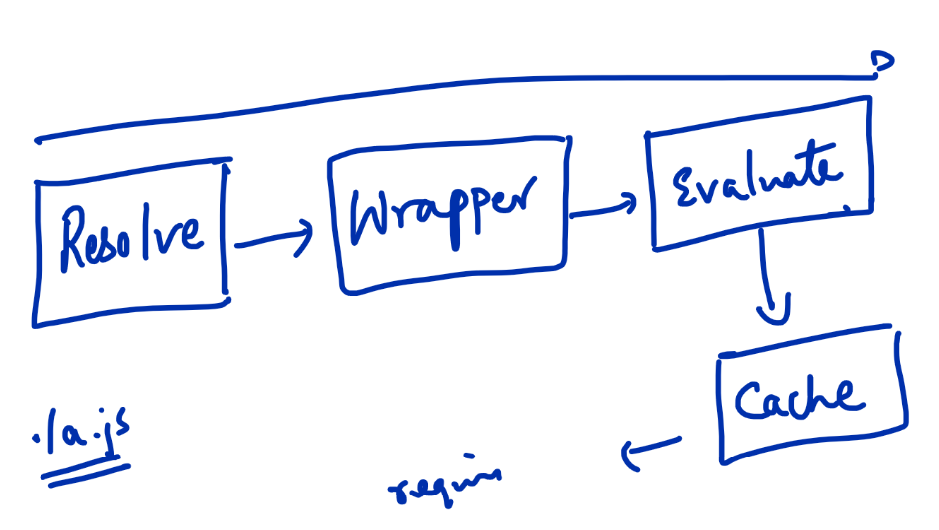
b.js



main.js



How the require works



How to use exports for functions & classes

circle.js

module.exports.area = function(r) {  
 // write some code  
}

module.exports.Circle = class {  
 //  
}

In Javascript you have constructor keyword

class Employee {   
 constructor(id, name, salary) {   
 this.id = id; // initializes id property for employee object  
 this.name = name; // initializes name property for employee

display() {   
 console.log(this.id); // console.log(id) doesn’t work

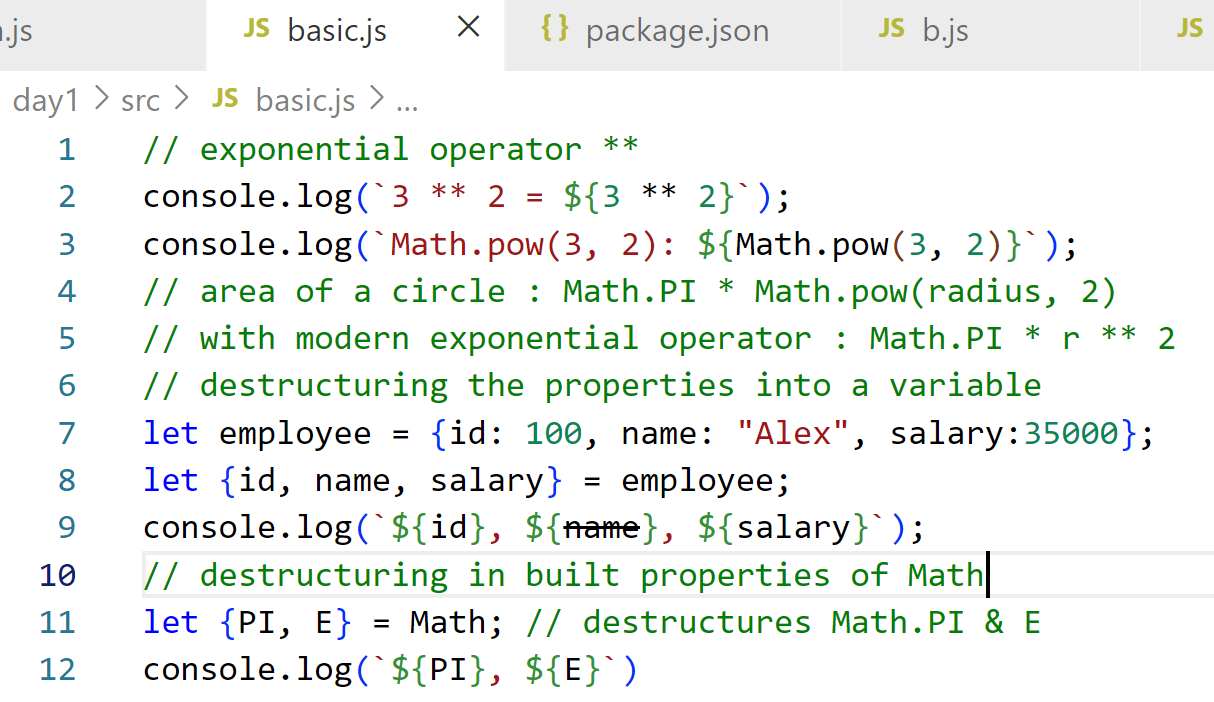
console.log(‘id = ‘+this.id+’, name = ‘+this.name+’, salary = ‘+this.salary);

console.log(`id = ${this.id}, name = ${this.name}`)  
 }  
 }   
}

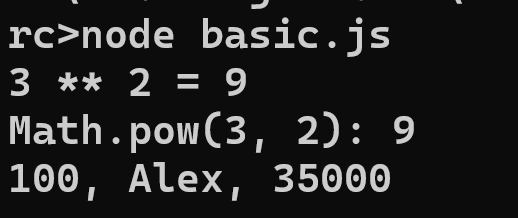
In many case back tick is useful

let id = 100;   
let name = “Alex”;  
let url = `http://ip:port:/${id}/${name}`

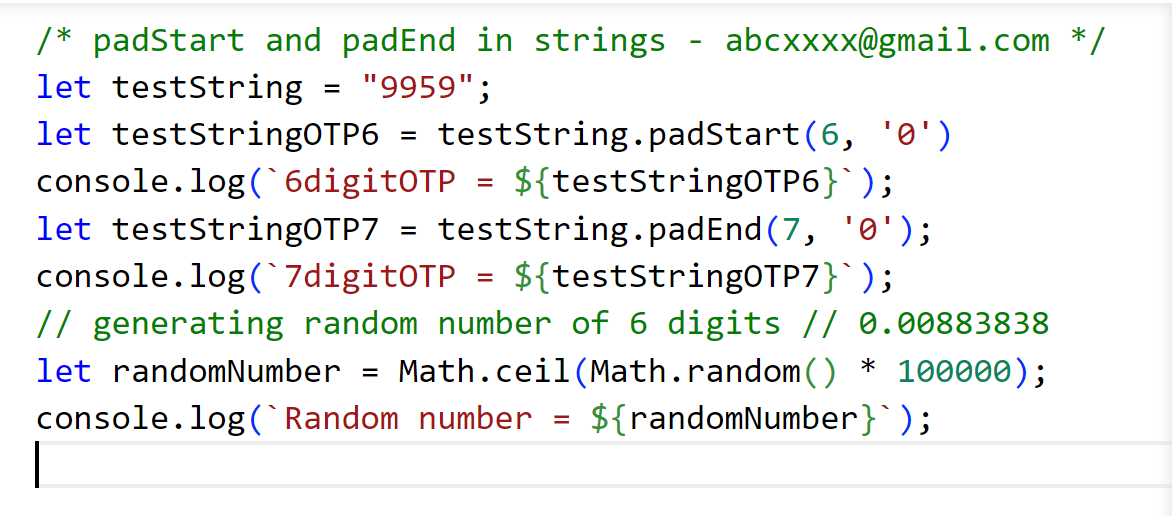
Understanding the basics



Output:



padStart & padEnd: This is used on strings when you want to add extra characters either in the beginning or at the end



Array methods:

filter, map & sort: These methods have an internal iteration

filter: It filters each element based on some condition and returns a new array

let filteredItems = array.filter(item => item % 2 == 0);

map: It transforms each element into another element & stores in a new array

sort: It sorts the elements and returns a new array

Callbacks, Promises & Async/await.

Callbacks: These are the functions which are supplied as an argument to a function which is going to execute later

Drawbacks of callbacks

If you keep nesting the callbacks it becomes difficult to understand the code.

Callbacks written in some places like interacting with SQL/No Sql databases

mongodb.connection(url, ( err, client ) => {   
 // run some queries using the client

client.query(“inserting data”, (err, result) => {   
 // here the nesting starts to grow when there dependent results we need obtain  
 });  
}) ;

Earlier people used to write callbacks when they want to interact with the apis

ex: XMLHttpRequest was the object people used to call the APIs,

Promises: It is introduced to simplify writing nested callbacks, it will have 2 states

1. Pending state
2. Settled State – Resolved/Success & Rejected/Failed

To settle the promise state we need to call some functions like then(callbackFn) or catch(callbackFn)

callbackFn of then() & catch() are called when promise is settled, the then() invokes its callbackFn, when the promise is resolved/successful, the catch() invokes its callbackFn, when the promise is rejected/failed.

Newer APIs in Javascript use promise to perform asynchronous operations.

ex: fetch(), axios libraries, mongodb libraries

Async/Await: It is used to make asynchronous operation synchronous, i.e., when you are dependent on the result of the previous asynchronous operation

async function testAsync() {  
 let res = await fetch(url); // blocked until promise is settled  
 let value = await res.json(); //blocked until promise is settled  
}

All three programs in an html file

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <div>

        <button onclick = 'fetchData()'>CallApi</button>

        <br />

        <button onclick = 'test()'>PromiseButton</button>

        <br />

        <button onclick = 'testAsync()'>AsyncAwaitButton</button>

    </div>

    <div id = 'users'></div>

    <script type = 'text/javascript'>

        async function testAsync() {

            let url = 'https://jsonplaceholder.typicode.com/users';

            let res = await fetch(url);  // waits promise to settle

            let value = await res.json();// waits promise to settle

            document.querySelector('#users')

                .textContent = JSON.stringify(value);

        }

        function test() {

            let url = 'https://jsonplaceholder.typicode.com/users';

            fetch(url)

                .then(response => response.json())

                .then(value => {

                    let ele = document.querySelector('#users');

                    ele.textContent = JSON.stringify(value);

                });

        }

        function fetchData() {

         let url = 'https://jsonplaceholder.typicode.com/users';

         let xhr = new XMLHttpRequest();

         xhr.open('GET', url, true);

         xhr.send(); // sends the HTTP GET Request

         // onreadystatechange callback is executed aysnc

         // readyState 1 to 4

         xhr.onreadystatechange = () => {

            if(xhr.readyState == 4) {

                // responseText will have resonseContent

                let response = xhr.responseText;

                let ele = document

                        .querySelector('#users');

                ele.textContent = response;

            }

         }

        }

    </script>

</body>

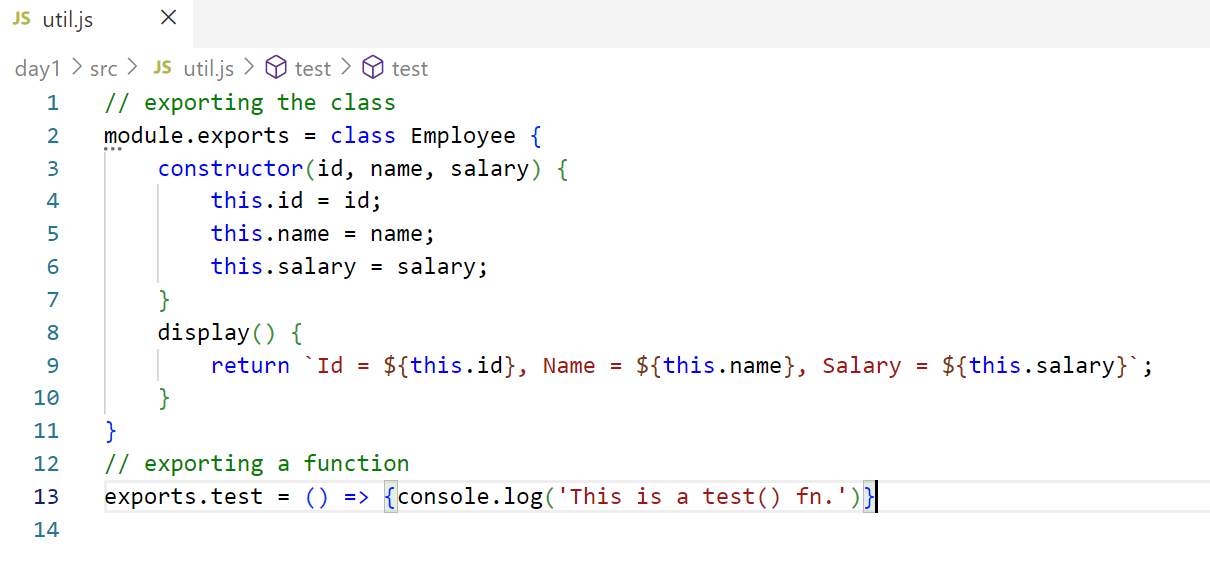
</html>

How to use the classes & functions as a module & include

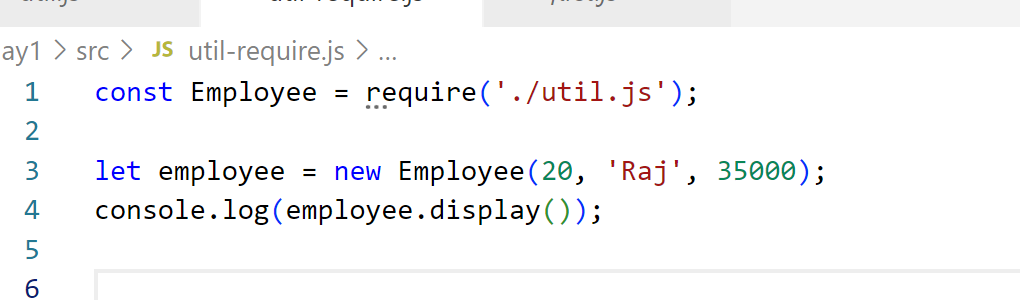
module.exports = class Employee {   
 // constructors & functions goes here  
}

exports.area = function() {  
 // area code goes here  
}

util.js



util-require.js



Modern syntax to import & export

export class Employee { } // import {Employee} from ‘./a.js’

export function test() { } //import {Employee,test} from ‘./a.js’;

Note: This modular type doesn’t work by default, we need to update package.json with “type”:”module”

util.js

// exporting the class

export class Employee {

    constructor(id, name, salary) {

        this.id = id;

        this.name = name;

        this.salary = salary;

    }

    display() {

        return `Id = ${this.id}, Name = ${this.name}, Salary = ${this.salary}`;

    }

}

// exporting a function

export function test() {

    console.log('This is a test() fn.')

}

// exporting an object

export const message = {status:200, result:"HELLO"};

// exporting an array

const users = [

    {id:1, name:"Virat", phone:9999},

    {id:2, name:"Rohit", phone:8888}

];

util-imports.js

import {Employee, message, test, users} from './util.js';

// use their names as it is to work with it

let emp = new Employee(123, 'Raj', 90000);

console.log(emp.display());

// using arrays

let arr = users;

arr.forEach(item => console.log(item));

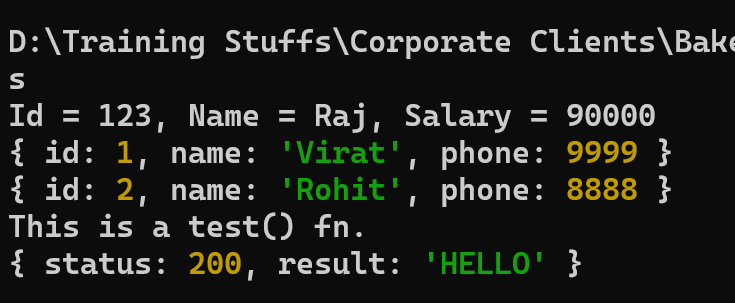
// calling test

test();

// using message without assignment

console.log(message);

Output:



Types of exports & imports

1. Named exports & imports: It must be imported with the same name
2. Default exports & imports : It can be imported with any name, a file can have maximum one default exports

// default export

export default class App {   
 //connect(), store(), retrieve(), update();  
}

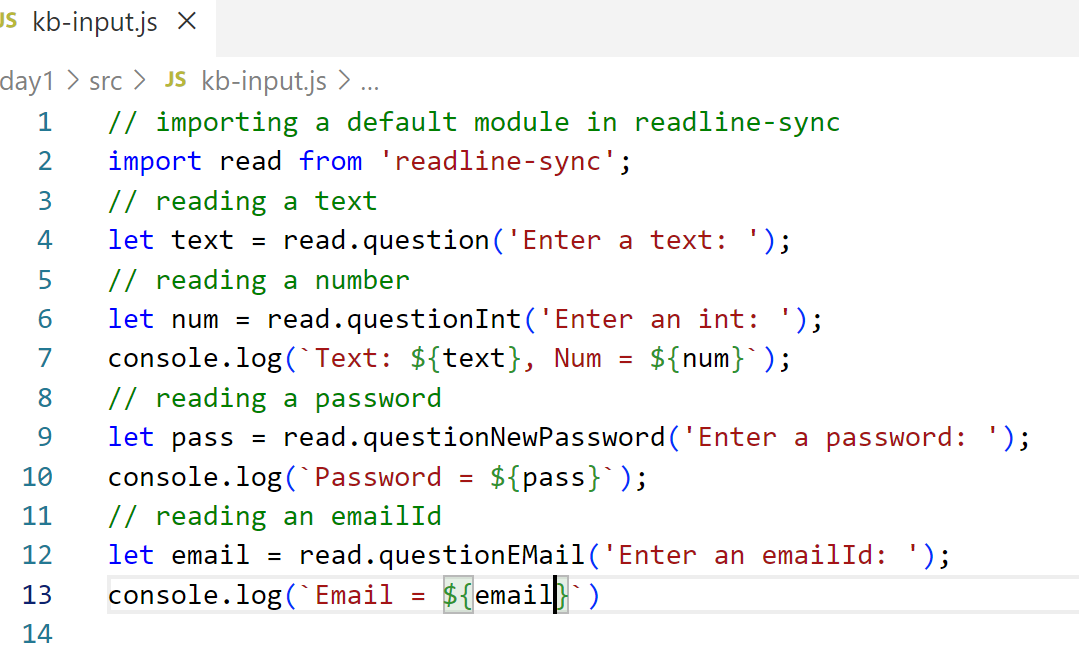
// named exports  
export class Xyz { }   
export class Demo { }

Importing the above modules  
import DB, { Xyz, Demo } from ‘./util.js’;  
DB.store(), DB.retrieve();

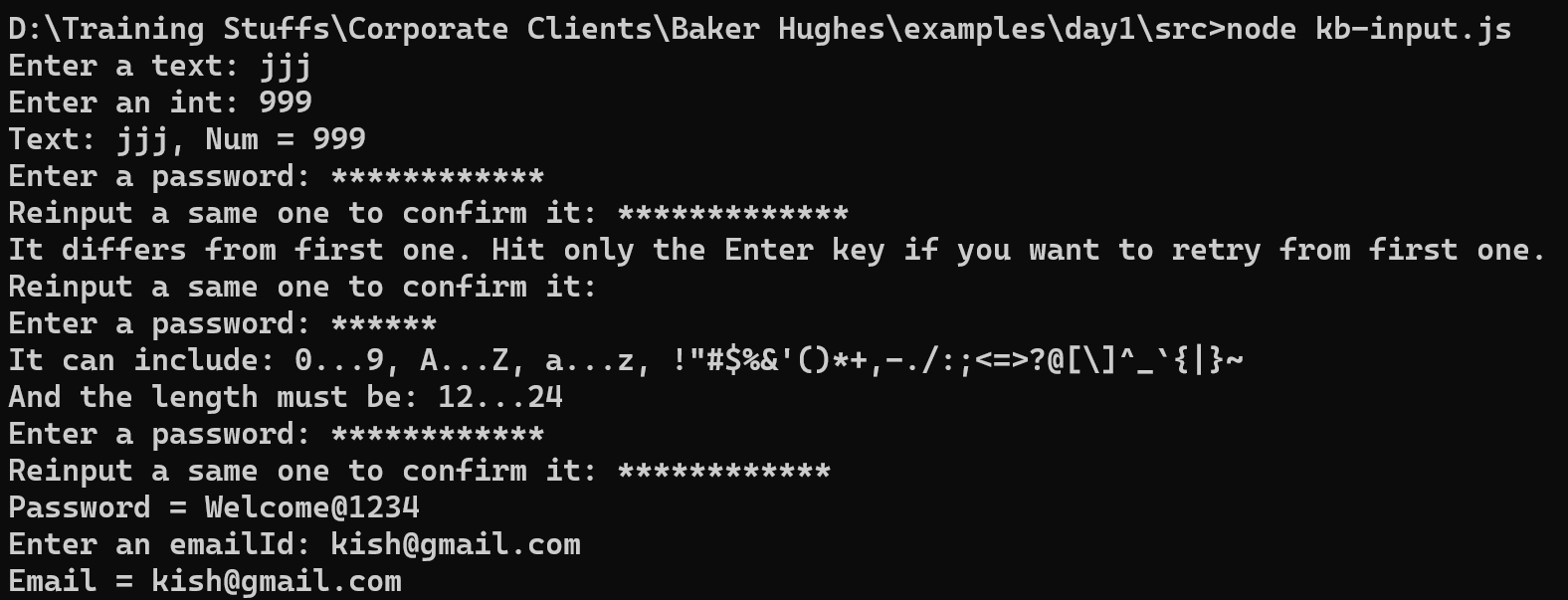
How to read input from the keyboard

npm install readline-sync // this is a third party module which is used to take input from the keyboard, it has a default module which provides methods to read strings, numbers, password, email-ids

Syntax: import readline from ‘readline-sync’;



Output:



Summary:

1. Node.js – event loop, libuv, v8
2. Callbacks, Promises, async/await
3. Modules – export, import, require
4. Console utilities – taking input from the keyboard
5. Basic apis – setTimeout, array methods

Agenda

1. OS module
2. FS module
3. HTTP/HTTPS module
4. Events module
5. Streams
6. Networking module – TCP & UDP
7. Cluster

OS Module:

To access OS features like platform, version, type, cores

import os from ‘os’;   
os.platform(); // win32  
os.type(); // Linux, Windows\_NT  
os.availableParallelism() // number of cores  
os.machine(); // x86\_64, arm, arm64, i3

OS module are used in some other node tools like React, Angular

ng –version: angular version and the OS angular is installed

os-features.js

// os-features.js

// importing os module

import os from 'os';

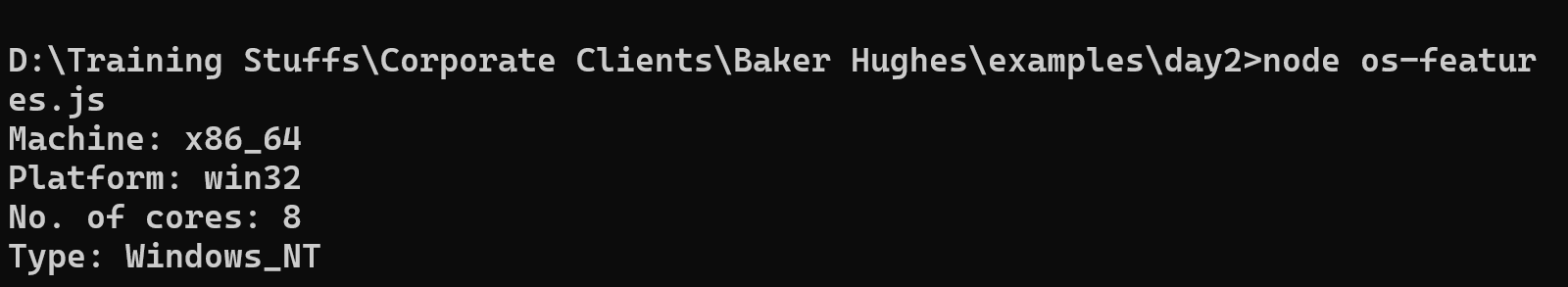
console.log(`Machine: ${os.machine()}`);

console.log(`Platform: ${os.platform()}`);

console.log(`No. of cores: ${os.availableParallelism()}`);

console.log(`Type: ${os.type()}`);

Output:



Node.js Events module

Events can be handled at the backend also using ‘events’ module, it gives you a class EventEmitter: Which can emit/generate an event also can handle the events that are generated.

EventEmitter: has 2 methods

1. emit(‘eventName’, data1, data2,…): This is used to generate the event
2. on(‘eventName’, (data1, data2,…) => { … } );

Example: Http servers listens to the request & runs a logic to handle the request, TCP server programs will be waiting for a data, once the data arrives it can execute some logic

Suppose you got an error in the code, you can have an event listener to listen to the error

events-demo.js

import EventEmitter from 'events';

let eventEmitter = new EventEmitter();

// you can event multiple listeners for the same event

// they are executed in the order they appear

eventEmitter.on('message', (value) => console.log(value));

eventEmitter.on('message', (...value) => console.log(value));

eventEmitter.on('error', (...err) => console.log(err));

// lets emit an event using setTimeout

setTimeout(() => {

    eventEmitter.emit('message', 'Hello');

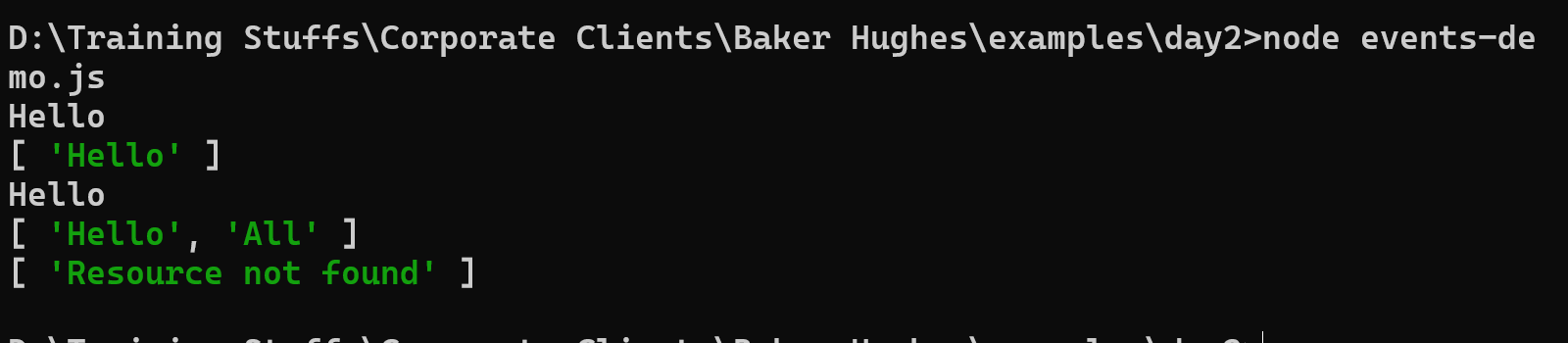
    eventEmitter.emit('message', 'Hello', 'All');

    try { throw 'Resource not found' }

    catch(err) { eventEmitter.emit('error', err) };

}, 3000);

Output:



FS Module

Gives you methods to read/write files in both synchronous & asynchronous way

Synchronous methods

1. readFileSync(filename)
2. writeFileSync(filename, data, options);

Asynchronous methods

1. readFile(filename, callback);
2. writeFile(filename, data, options, callback);

Promise based methods in fs/promises

1. readFile(filename).then(callback).catch(callback)
2. writeFile(filename, data, options).then(cb).catch(cb)

fs-writing.js

import fs from 'fs';

import fsp from 'fs/promises';

//writing synchronously: writeFileSync(fn,d,options)

// {flag:'a+'} appends with the old content

fs.writeFileSync('./myFile.txt', 'Hello World\n',

    {flag: 'a+'});

//writing asynchronously: writeFile(fn, d, op, cb)

fs.writeFile('./myFile.txt',

    'Hello World Async\n', {flag: 'a+'}, (err) => {

        if(!err) console.log('File is written')

    }); // end of writeFile

//writing asynchronously using promise based methods

// writeFile(fn, d, op).then().catch()

fsp.writeFile('./myFile.txt',

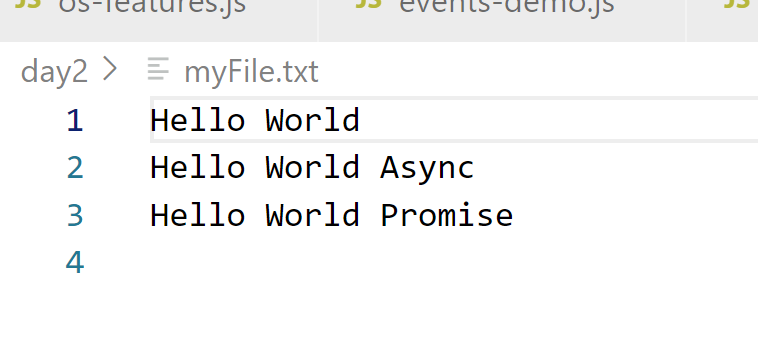
    'Hello World Promise\n', {flag: 'a+'})

    .then(() => console.log('Written using promise'))

    .catch(err => console.log(err));

Output:

myFile.txt will be created with below content



Create a program that can read files in all the 3 ways – synchronous, asynchronous, promise based

fs-reading.js

// use fs.readFileSync, fs.readFile, fsp.readFile methods

import fs from 'fs';

import fsp from 'fs/promises';

// synchronous - returns a Buffer - Binary form data

let buffer1 = fs.readFileSync('./myFile.txt', {encoding:'utf-8'});

console.log(`${buffer1}`);

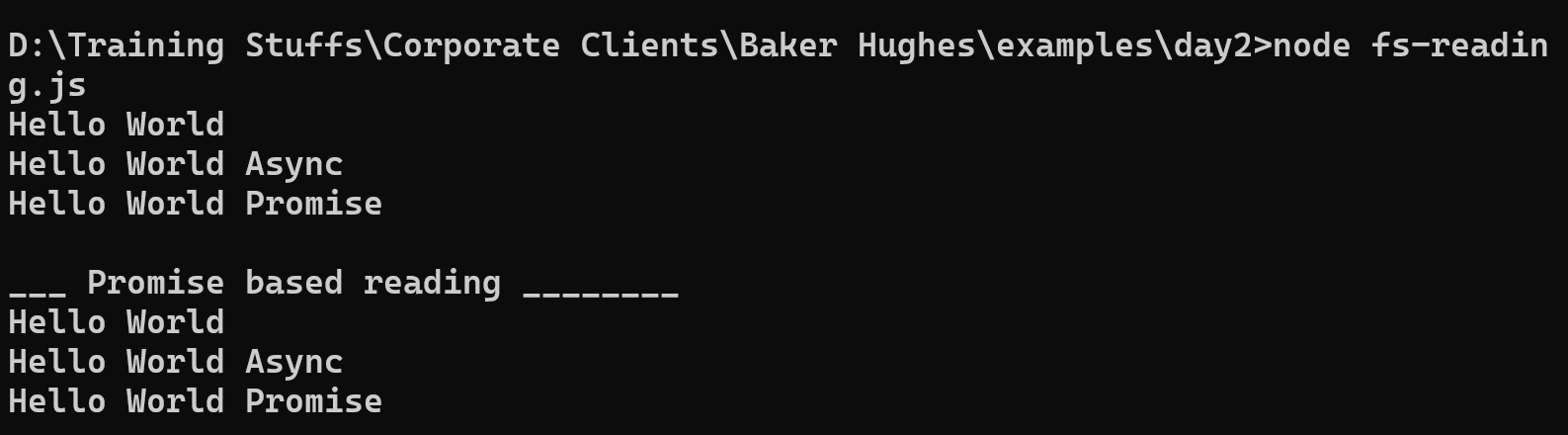
// promise based

console.log('\_\_\_ Promise based reading \_\_\_\_\_\_\_\_')

fsp.readFile('./myFile.txt', {encoding:'utf8'})

    .then(value => console.log(value));

Output:



Performing Todo list: Task manager using json files and fs module

1. Create a reusable read & write methods so that it can be called from different code
2. Create a main file that can call the above methods

Streams:

It helps to process the data part by part called chunk of data instead of loading the entire data in the memory.

We can use streams while reading, writing, based on that we have 4 types

1. Readable Streams: Reading data in the form streams
2. Writable Streams: Writing data in the form streams
3. Duplex Streams: Does Read & Write both, used in TCP / UDP programs
4. Transform Streams: It modifies the stream while reading/writing ex: compressing data

Readable/Writable Streams you can create using ‘fs’ module

let read = fs.createReadableStream(filename, {bytes});

let write fs.createWritableStream(filename, { } )

read.on(‘data’, (chunk) => { } );

write.on(‘data’, (chunk) => {});

read.resume(); // to resume the streaming

write.resume(); // to resume the streaming

read/write.pause(); // to pause the streaming

streaming-demo.js

import fs from 'fs';

let counter = 0;

// highWaterMark to mention the size we want to stream

let read = fs.createReadStream(

    './myFile.txt', {highWaterMark: 10}

    );

read.on('data', (chunk) => {

    counter++;

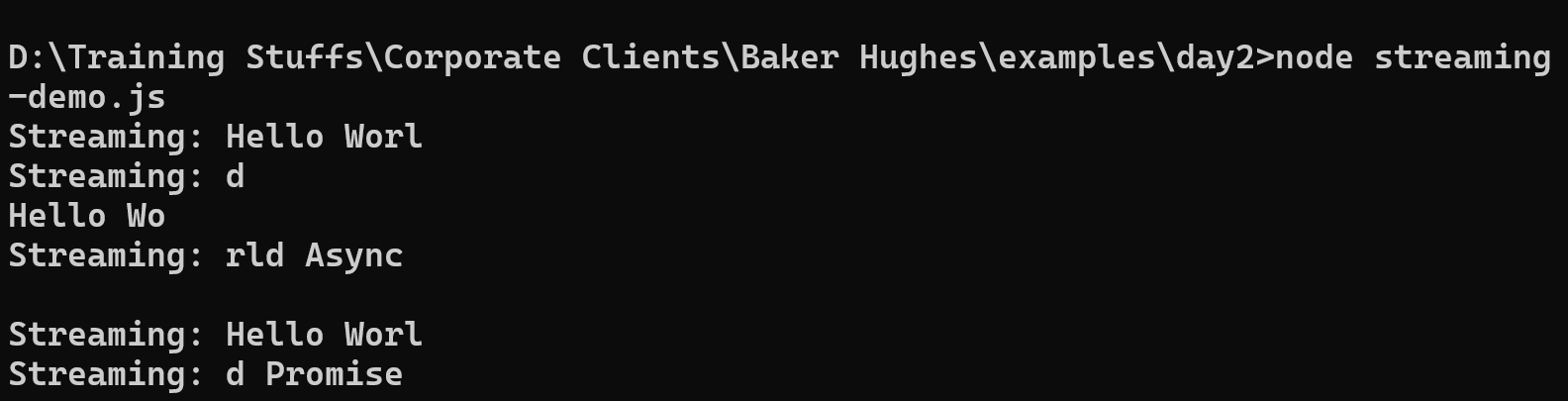
    if(counter == 2) read.pause();

    setTimeout(() => {read.resume();}, 2000);

    console.log(`Streaming: ${chunk}`);

});

Output:



HTTP / HTTPs Servers

HTTP servers can handle the request & process it to generate the response

HTTPs does the same thing, but it uses SSL

import http from ‘http’;  
import https from ‘https’;

let server = http.createServer( callbackFn )

let server2 = https.createServer( option, callbackFn )

option = {   
 key : fs.readFile(locationOfkey),   
 cert : fs.readFile(locationOfSSLCertificate)  
}

// to start the server

server.listen(port\_number, callbackFn)

Webservices

It is implemented in two principles

1. HTTP protocol methods : get, post, put, delete
2. URL

Consumer (Webservice) Producer (Webservice)

axios.get(url) store() { } /POST  
axios.post(url) delete() { } /DELETE  
 fetch() { } /GET  
 update() { } /PUT

Node.js uses express library to create api’s, express is not inbuilt

npm install express

Express: Is build on top of http server

import express form ‘express’;

let app = express();   
  
app.use( cors() }  
app.post(url, cb); // post request  
app.get(url, cb);

Net module

TCP server is used to in order to have a connection oriented protocol, where there wouldn’t be any data loss.

Note: Receiver must be ready first, then only sender can send the message

tcp-receiver.js

import net from 'net';

// TCP servers are event based: data, end, connection

let counter = 0;

let server = net.createServer(client => {

    console.log('Connected...');

    counter++;// to track how many clients connected

    client.on('data',(data)=>console.log(data.toString()))

    client.on('end', ()=>console.log('client disconn..'));

    // send data to the client

    client.write(`Hello you're client no.: ${counter}`);

});

// starting the TCP server

server.listen(7777, () =>

    console.log('TCP Reciever is ready in :: 7777'));

tcp-sender.js

import net from 'net';

import readline from 'readline-sync';

let client = new net.Socket();

let message = '';

client.connect(7777, 'localhost', () => {

    do {

        message = readline.question('Type a message: ');

        client.write(message);

    } while(message != 'quit');

    client.destroy(); // close the client

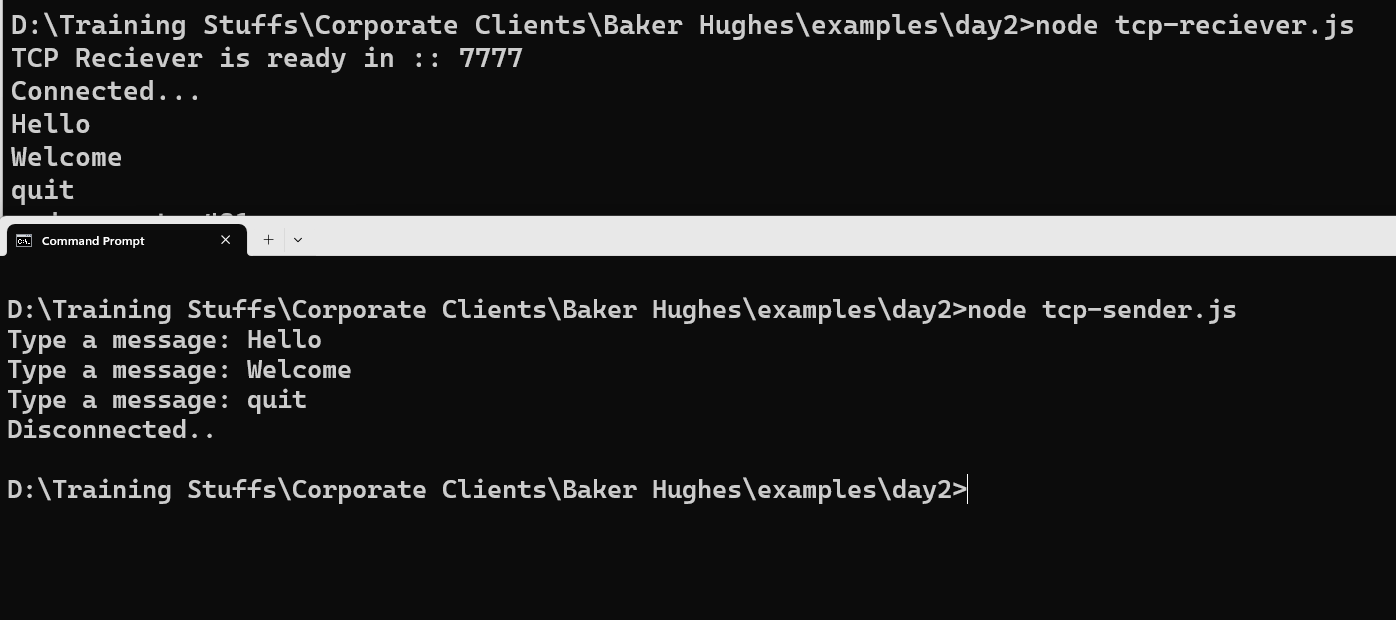
});

client.on('data', (data) =>

    console.log(`Server Sent: ${data.toString()}`))

client.on('close', () => console.log('Disconnected..'))

Output:



LoadBalancer

myapp.js

import express from 'express';

let app1 = express();

let app2 = express();

app1.listen(8081, () => console.log('Running in 8081'));

app2.listen(8082, () => console.log('Running in 8082'));

let handler = serverName => (request, response) => {

    console.log(`Server:${serverName},URL: ${request.url}`)

    response.end(`Request handled by ${serverName}`)

};

app1.get('\*', handler('a-instance'));

app2.get('\*', handler('b-instance'));

loadbalancer.js

import request from 'request';

import express from 'express';

// list of applications running in different port

let servers = ['http://localhost:8081', 'http://localhost:8082'];

// counter to track to which server the request must be sent

let current = 0;

// handler to take all the incoming request to the loadbalancer

const handler = (req, res) => {

    // select the server to forward the request

    const server = servers[current];

    req.pipe(request({url: server+req.url})).pipe(res);

    // update the current server to be 0 or 1

    current = (current + 1) % servers.length;

}

const lbServer = express();

// handle favicon

lbServer.get('/favicon.ico', (req, res) => {res.send('./favicon.ico')});

// pass the request to handler method

lbServer.use((req, res) => {handler(req, res)});

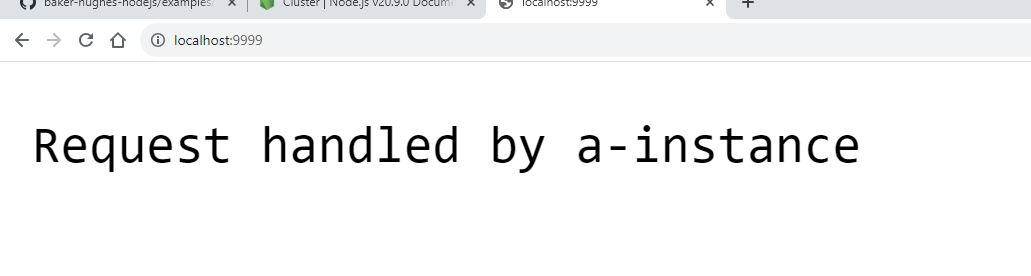
lbServer.listen(9999, () => console.log('load balancer started in 9999'))

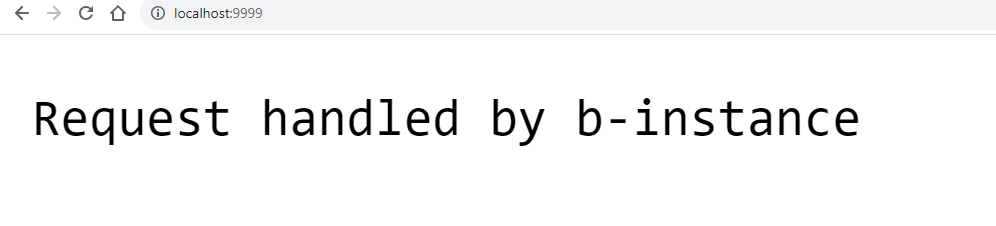
Output:

Run >> myapp.js

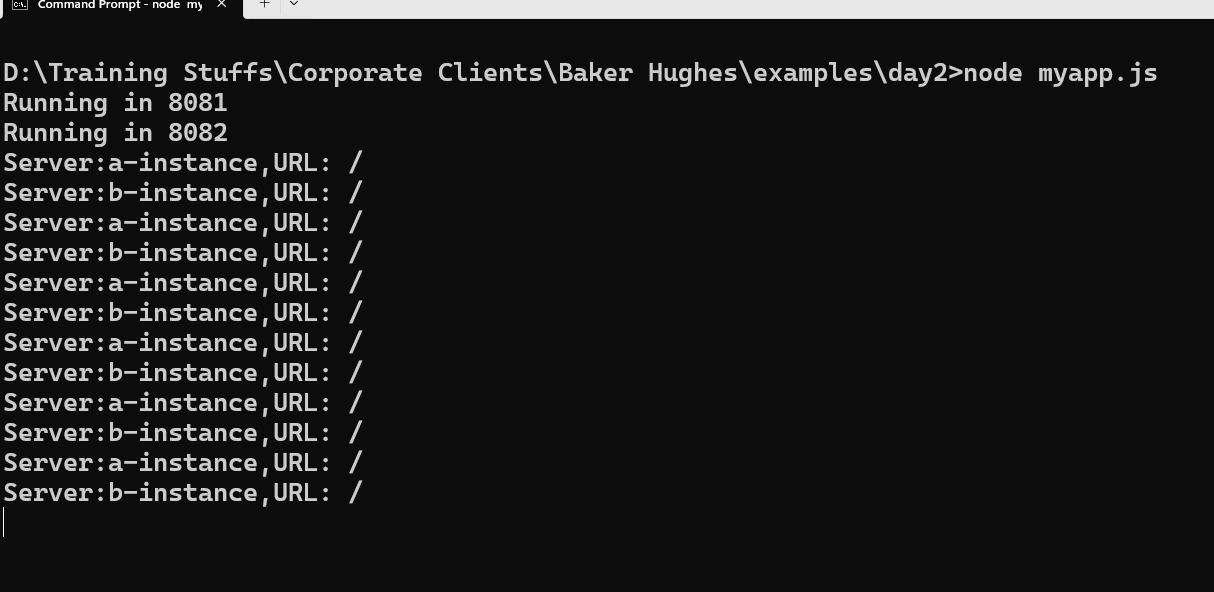
Run >> loadbalancer.js

Send multiple request to the load-balancer which sends request to the same application having multiple instances in a round-robin fashion.





In Console you will see



Node.js can interact with any database because it provides libraries from the npm.

mongo-client.js

import mongodb from 'mongodb';

let client = mongodb.MongoClient;

client.connect('mongodb://127.0.0.1:27017')

.then(client => {

    let db = client.db('demodb');

    // request.body

    let body = {"\_id":3, "name":"Rohit"};

    // storing

    db.collection('employees').insertOne(body)

    .then(value => console.log(value))

    .catch(err => console.log(err))

    // retrieving

    db.collection('employees').find().toArray()

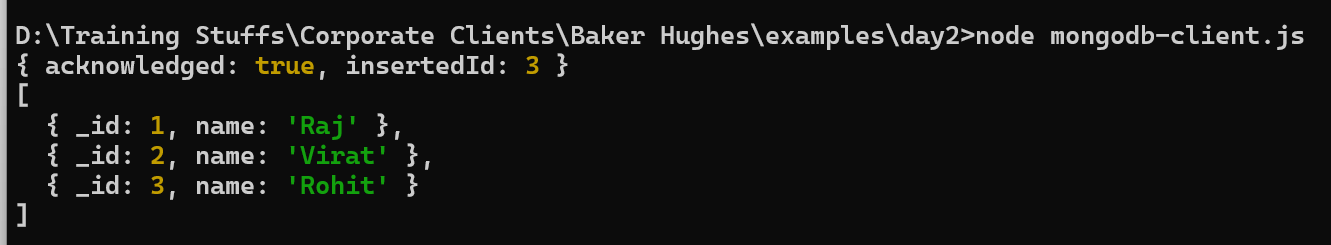
    .then(value => console.log(value))

    .finally(() => client.close());

})

.catch((err) => console.log(err))

Output:



Node.js reference:

1. Official document: <https://nodejs.org/dist/latest-v20.x/docs/api/fs.html>
2. Node.js book: Node.js in Action