# Intro

Mongo (DB)

document oriented database written in C++

Collection -> Documents -> sets of name-value pairs

Node.js(environment)

Node.js is a server side cross-platform open source JavaScript execution environment

Express.js(back end which runs on Node)

Express.js is a lightweight Node.js based application used for back-end development

Angular(Front End)

Angular is an open source JavaScript framework developed by Google, for building both mobile and desktop web applications used for rapid front-end development

URL(Uniform Resource Locator)

Protocol://Host:Port/Path

HTTP is a request/response based stateless and connectionless protocol.

stateless: each request is executed independently

connectionless: a new connection is established for every request

HTTPS is SSL (Secure Socket Layer) protocol

MIME (Multipurpose Internet Mail Extensions)

type/subtype;parameter=value

text/plain;charset=UTF-8

request header will pass information about request

request body will be absent in case of GET request

<div> tag to group the field names and the input fields of our form together

<span> is similar to <div> element, but <div> is a block-level element where as <span> is an inline element.

<html>

<head>

<meta>

<title>

</head>

<body>

<form>

<div>

<Label>

<Input>

</div>

<div>

<label>

<select>

<options>

</select>

</div>

</form>

</body>

</html>

select

datalist

only step signifies the interval.

“max”, “pattern”, “value” are designated to “set upper limit”, “matching data value”, “setting default value” respectively.

<a>

href

target

\_self(def)

\_blank

\_parent

\_top

CSS

Adding CCS to HTML

Inline (<input style = "color:white">)

Embeded (<style>p{color:white}</style>)

External (<link href="cssfile.css" rel="stylesheet">)

If all the three are used Inline conditions are applied

Incase of Embeded and External, the latest declared will be applied

If for same tag same type of different conditions are applied then,

it will take the combo of all of them, while in conflict the latest values.

Type selector < Class selector < ID selector

With pseudo-class selectors we can style the elements based on

their position within the Document or based on their interactive state

<a> :link :visited :hover :active MUST be in orderd format to work properly

<input> :focus :enabled :diabled :checked :intermediate :required

CSS Box

MARGIN

BORDER

PADDING

CONTENT

PADDING Top, Right, Left, Bottom

annotations : able to define the data type of the VARIABLE

eg: var i : number , var s: string

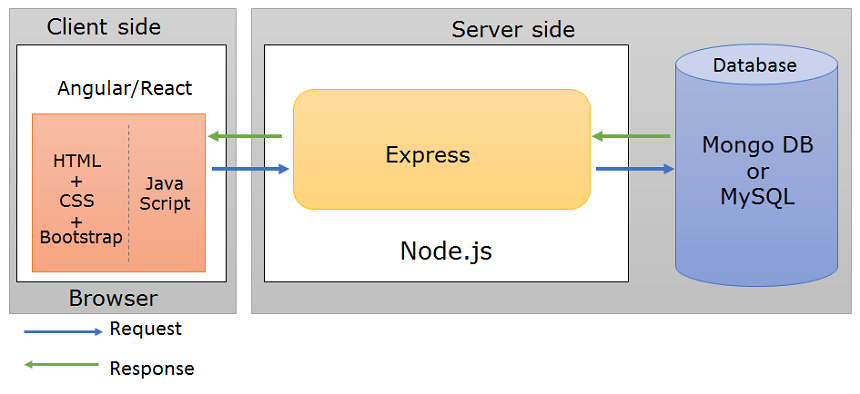
eg:- function f (a: string) : string{}

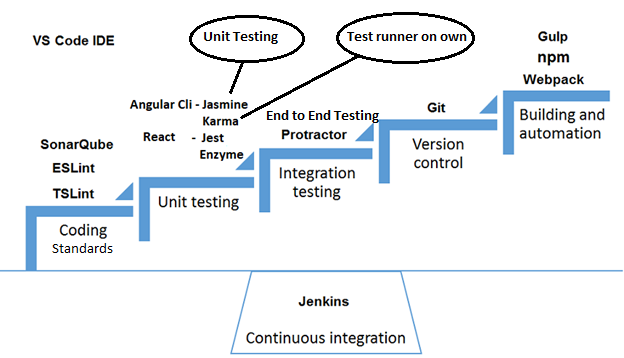
generics - parameter type dynamic so that function can be reusable with any datatype

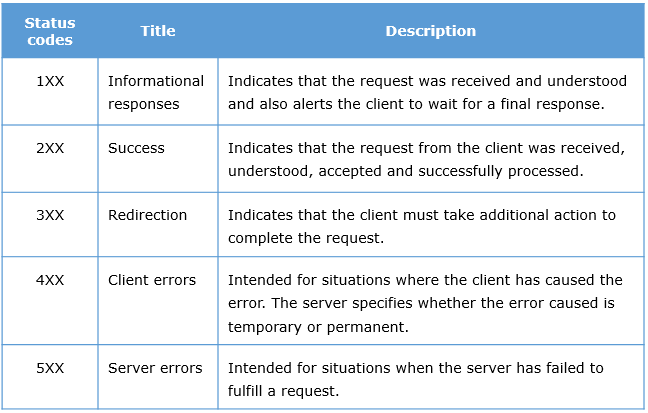
eg: function funarray<T>(s:T[ ] ):T[ ]{}

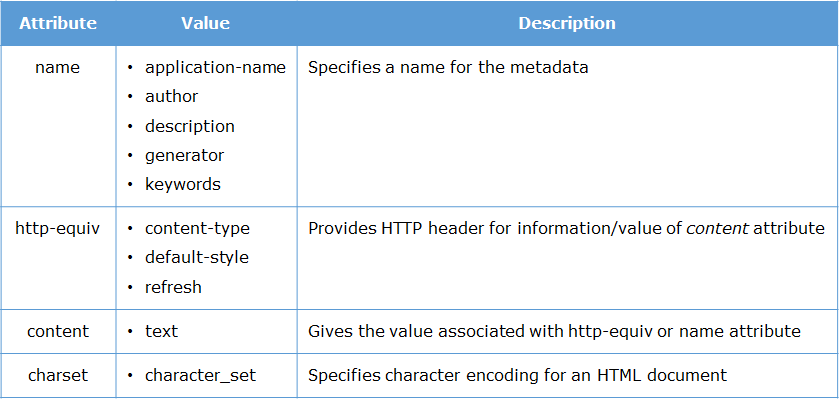
Decorator: All a decorator is, is a function that takes a class as an argument

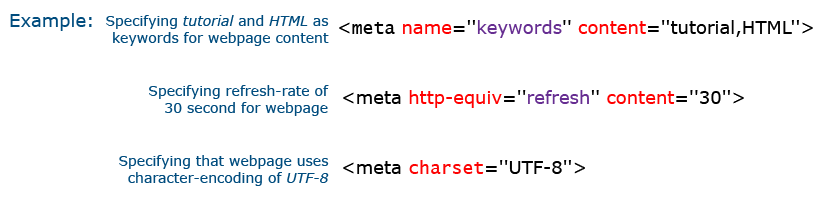
# **CSS**

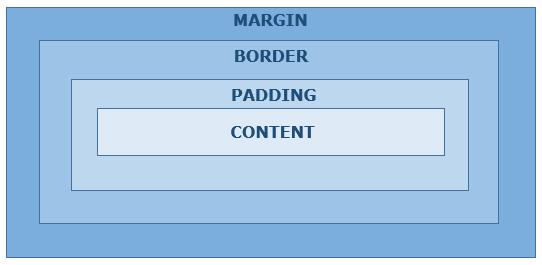


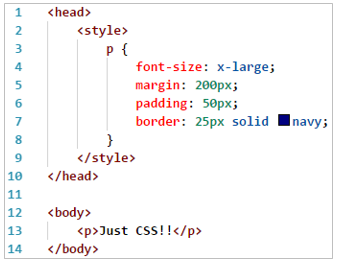


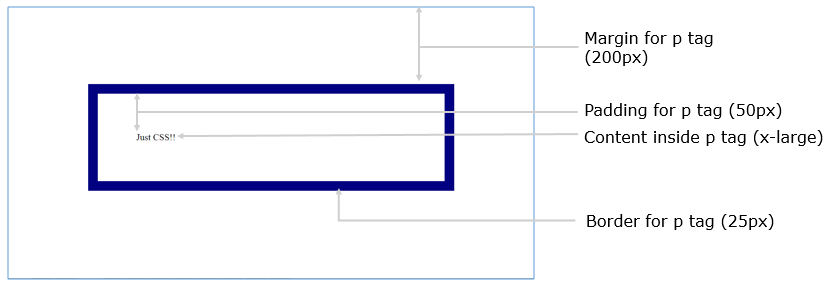












Directives used to change DOM (document object model)

structural directives - \*ngIf, \*ngFor, \*ngSwitch(A Structural directive changes the DOM structure by adding and removing DOM elements.)

attribute directives - [ngStyle],[ngClass](Attribute directives changes the appearance or behavior of a component or element.)

Animations in CSS are done using key-frames.

@keyframes exampleAnimation2 {

0% { background-color: red; }

25% { background-color: yellow; }

50% { background-color: blue; }

100% { background-color: green; }

}

div {

width: 100px;

height: 100px;

background-color: red;

animation: exampleAnimation 4s 1(iteration count) ease-in;

}

Media queries are used to specify how the web page should look on different sizes of screens

@media only screen and (max-width:500px){

body{

background-color: blue;

}

}

Syntactically Awesome Style Sheets(SASS) is considered as a CSS preprocessor and is a stylesheet language .(extension .scss)

used for declaring variables and reusing them.

$default-background-color: white;

.tour {

.tour-card {

border: 1px solid black;

background: $default-background-color;

}

}

used for nesting classes

.content-area {

display: flex;

flex-wrap: wrap;

.main {

display: flex;

flex-direction: column;

flex:8;

.tour {

display: flex;

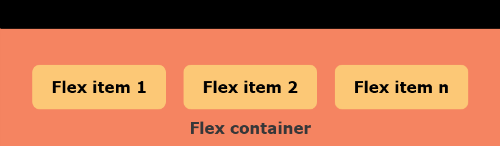
}

}

}

command to generate css file from sass

sass scss-demo.scss output.css



# **MONGO DB**

Mongodb:

db.product\_catalog.insert (

    {

        prodid:7000010,

        prodname:"nosql distilled",

        publisher:"Addison-Wesley",

        genre: {academic: "technical"},

        ISBN:1234567,

        price:400

    })

Insert acts like both insertone and InsertMany, output WriteResult({"nInserted":1})

db.product\_catalog.find({

$and:[

{manufacturer:"apple"},

{colors: { $all:["black", "silver"] } }

]},

{\_id:0, prodname:1, colors:1, price:1}

).pretty()

$all means shld contain all mentioned, $in anyone matches,

$and, $or, $not, $not

$gte, $ne

db.product\_catalog.updateMany(

{ "price" : { $gt : 80000 }, "manufacturer" : "apple" },

{ $set: { "prodname" : "iphone 7 plus" } },

{ upsert: true }

)

If upsert : true mentioned then if record not founf it will create a new record

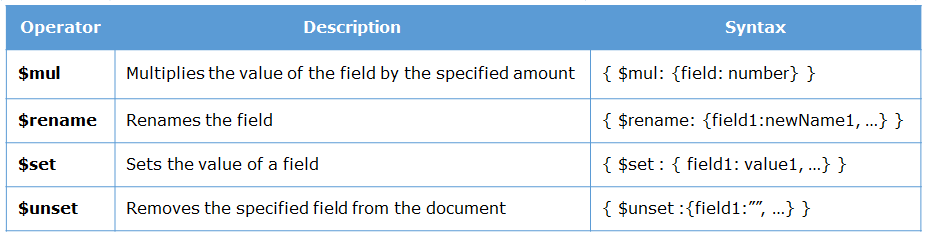
db.product\_catalog.update(

{ ISBN: 18407806 },

{ $inc: { price: 50 } }

)

$inc (+ or -), $mul (multiply or divide), $rename (change field name) , $set (sets field value), $unset (removes specified field from document)

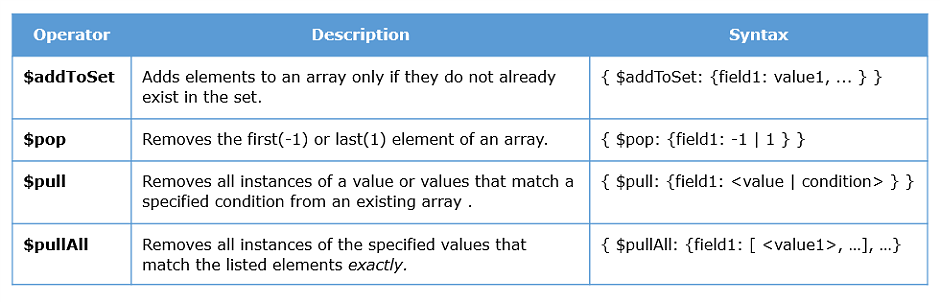


db.product\_catalog.update(

{ prodid: 7000001},

{ $push: { colors: "white" } } )

To push value into an array in db (if doesn’t exist create a new one)



Delete:

Example:

db.product\_catalog.deleteMany(

{ price: { $lt : 1000 } } )

Also has deleteOne

Aggregation :

db.product\_catalog.count( { "categories.sub": "smartphones" })

db.product\_catalog.distinct( "manufacturer" )

Example:

db.product\_catalog.aggregate( [

{ $project: { \_id: 0, manufacturer: 1, price: 1 } },

{ $group: { \_id: "$manufacturer", totalPrice: { $sum: "$price" } } }

] )

Example:(similar to find)

db.product\_catalog.aggregate(

{ $match: {$and: [{ manufacturer: "lenovo" }, {price : {$lt:10000} } ] } } )

Example:

db.product\_catalog.aggregate( [

{ $sort : { price: 1 } },

{ $limit: 5},

{ $out: "FiveCheapestMobiles"}

] )

Example:

db.product\_catalog.createIndex(

{ price:-1 })

Example:

db.product\_catalog.createIndex(

{ price:1, rating:-1 })

Example:

db.product\_catalog.createIndex(

{categories:"text"})

db.product\_catalog.find(

{ $text:{ $search:"smartphones"}})

Example:

db.product\_catalog.getIndexes()

Syntax:

If you know the name of the index

db.collection\_name.dropIndex( "index\_name" )

OR

For ascending indexes:

db.collection\_name.drop\_Index( { field\_name : 1 } )

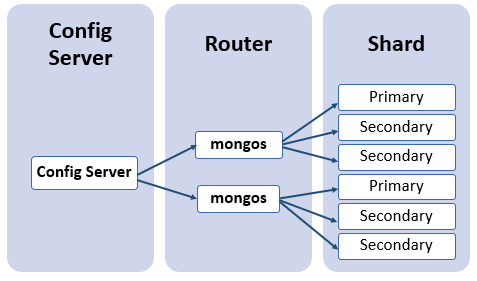
OR

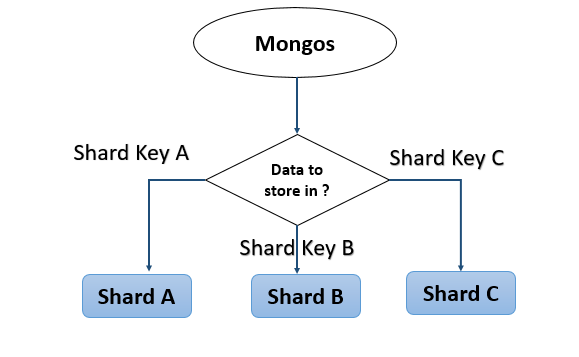
For descending indexes:

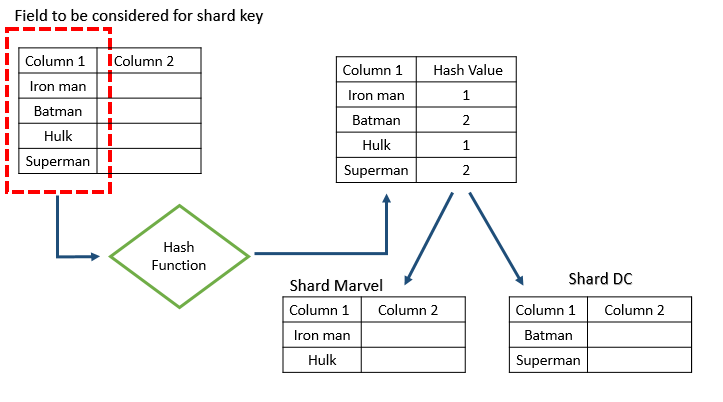
db.collection\_name.drop\_Index( { field\_name : -1 } )

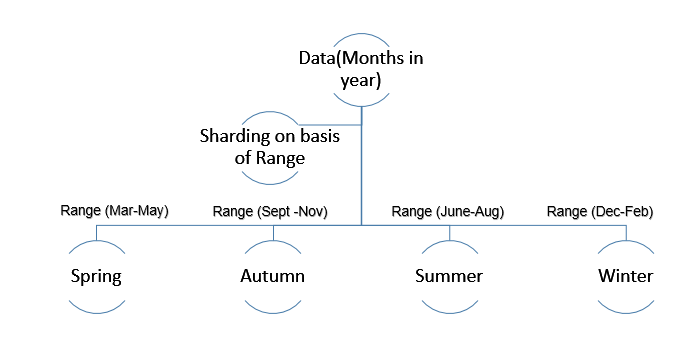
Syntax: to delete all indexes

db.collection\_name.dropIndexes()









English - is/am/are/was/were + used to + ing verb

superlative degree follows in (if it is place/group) else it follows of

eg: the shortest boy of/in the school (ans: in)

if ends with ce then it is abstract noun

present perfect/present perfect continuous we will use since not for

# **JAVASCRIPT**

* JavaScript is an interpreted language
* JS has 5 data types: number, string, undefined, null, object

# **Functions**

* Functions are objects in JS
* Functions can be stored in variables, passed as parameters or returned as values
* Functions without names are called anonymous functions
* Arrow functions are shorter way to write anonymous functions
* Variables can either be in global, local or block scope

1. Functions are actually Objects. That means a function can be stored in a Variable.
2. We can also pass Functions as a parameter to another function.
3. Before any function executes, all the Local Variables are **Hoisted**in the function. Hoisting is a phenomenon, where no matter where the variable is declared inside the function, they are all ***pushed as the first statements inside the function*** during the function execution.
4. However, only variable name is hoisted and ***not its value.***
5. The variable ***i*** has been declared with ***var*** keyword, it is ***accessible throughout the function***.
6. A ***variable with a block scope*** is accessible only within the block of statements and not throughout the function.
7. ***const*** is a keyword which is also used to create a block scoped variable. But the difference between const and let is that, a const variable ***cannot be modified***. It is Constant.

Objects have ***properties and methods***. JavaScript provides many standard **built-in** objects. In addition to that it also provides an option to create **user defined** objects.

Some of the commonly used built-in objects are:

* Array
* Date
* String

# **Array**

* .forEach()
* .map()
* .filter()
* .find()
* .push()
* .pop()
* .splice()

1. Consider the **forEach()** function of an array. This function takes another function as parameter and invokes the function for every item in the array.

placesToVisit= ["Paris", "New York", "Switzerland"];

placesToVisit.forEach(place => console.log("Trip to " + place));

// Trip to Paris

// Trip to New York

// Trip to Switzerland

1. We know that an array object has a **.map()** function that creates a new array based on what the passed callback function does.

placesToVisit= ["Paris", "New York", "Switzerland"];

placesUpperCase = placesToVisit.map(place => place.toUpperCase());

console.log(placesUpperCase);

// ["PARIS", "NEW YORK", "SWITZERLAND"]

1. We know that an array object has a **.filter()** function that returns a filtered sub array based on what the passed callback function does.

placesToVisit = ["Paris", "New York", "Switzerland"];

filteredPlace = placesToVisit.filter(place => place.length > 5);

console.log(filteredPlace);

// [ 'New York', 'Switzerland' ]

1. We know that an array object has a **.find()** function that returns the first element in the array based on what is passed as callback function.

placesToVisit = ["Paris", "New York", "Switzerland"];

findPlace = placesToVisit.find(place => place.length > 5);

console.log(findPlace);

// "New York"

**Palindrome**

function isPalindrome(s)

{

var reversedText = s.toLowerCase().split('').reverse().join('');

return s=== reversedText;

}

console.log(isPalindrome("level"));

# Objects

Creating **Object** using object literal

var empOne = {

name : "John",

empNumber : 1001,

emailId : "John@gmail.com",

swipeIn(){console.log("Swipe In by "+this.name)}

};

**Accessing object properties**

object.property

object[property]

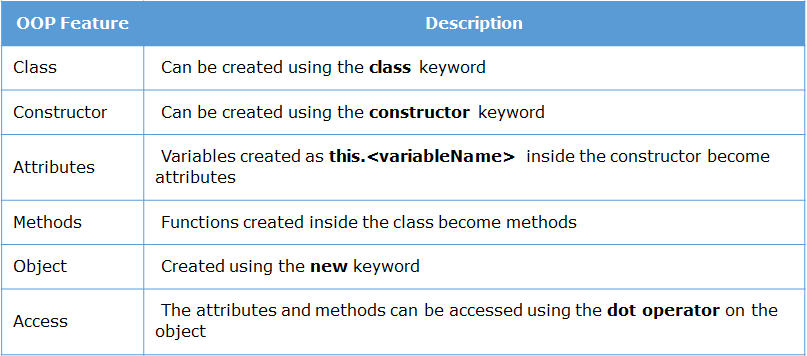
**Iterating an object**

for..in

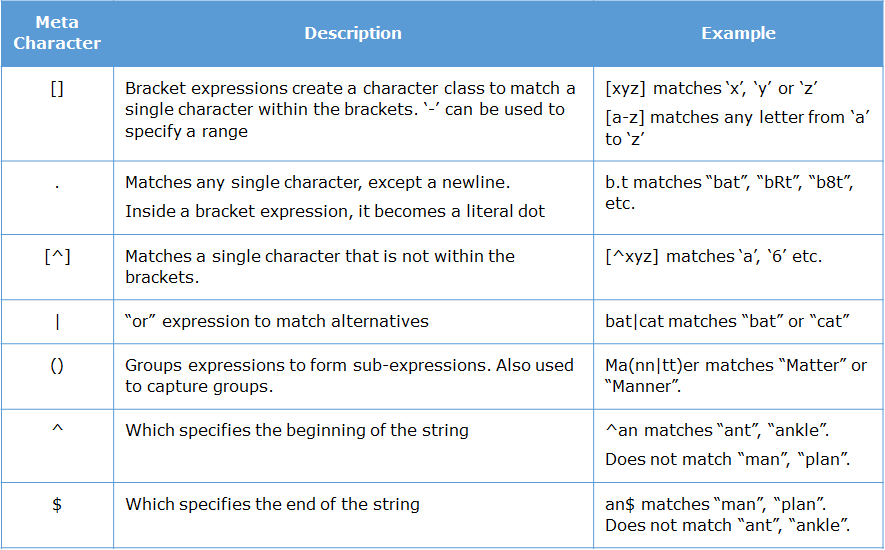
object.values(ObjectName)

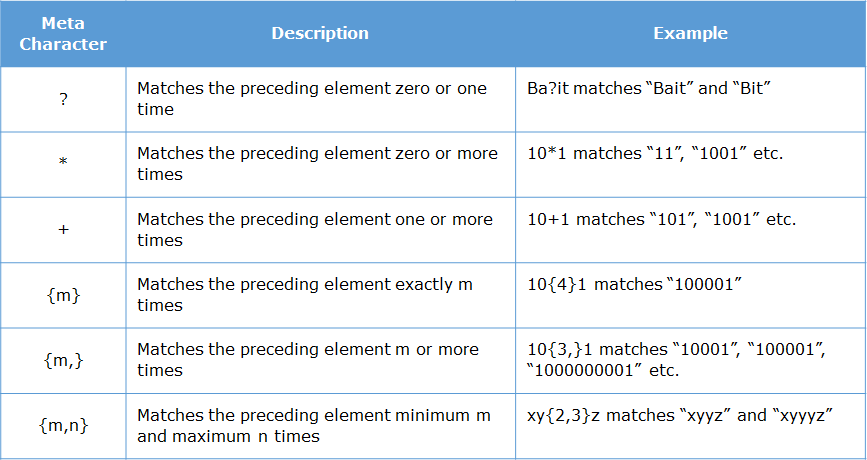
Using Object Destructuring we can destructure an existing object into variables. If the var is prefixed by 3 dots, then it is a Rest Variable and can store more than one property.

# Class



# RegEX





# URI and URL difference

[example.com/Project/posts](http://www.example.com/Project/posts) – URI (uniform resource identifier) can identify but not locate

http://www. [example.com/Project/posts](http://www.example.com/Project/posts) – URL ( UR Locator) can be used to locate and identify a resource

# WEB API

In fact console.log() is actually not part of the language. It is provided by the browser as part of its API. It is through that API we are able to interact with the browser and print something on the browser's console.

There are many Web API. Some of which we will cover in the course are:

* XMLHttpRequest
* DOM
* Notification
* Storage

**Note:** Since these API's are provided by the browser, they will not work in Node

XMLHttpRequest and DOM are Asynchronus Javascript And XML (AJAX) in nature

setTimeOut(fn(),timeInMilliSec) is DOM api following AJAX technique

Same, XMLHttpRequest and its methods like http.open(‘GET’,url) follow Asynchronus i.e AJAX techniques

To make sure the function executes once the response is received we use callbacks, if there are too many callbacks nested up like callback has another callback (callback hell) to avoid this problem we use promise.

function getTrip(){

return new Promise(function(resolve){

setTimeout(function() {

resolve("Lets go to Trip");

}, 2000);

});

};

This code promises a states that if everything goes well then it will resolve the “Lets go..” data

function bookFlight() {

return new Promise(function (resolve) {

setTimeout(resolve(5600), 2000);

})

}

function bookHotel(flightPrice) {

return new Promise(function (resolve) {

setTimeout(resolve(7000 + flightPrice), 1000);

})

}

function getTotal(){

bookFlight()

.then(function (flightData) { return bookHotel(flightData) })

.then(function (cumulativeData) { console.log(" Total is " + cumulativeData) })

}

getTotal()

we can use await inside async function to make code wait till promise gets resolved, above code can be written in

async function getTotal(){

var flightData=await bookFlight();

var cumulativeData=await bookHotel(flightData);

console.log(" Total is " + cumulativeData)

}

Suppose there is a JSON file in ../json url then we can access through XMLHttpRequest

Function getData(url){

var xhr = new XMLHttpRequest();

xhr.open(‘GET’,url);

xhr.onload = function () { // Invokes after getting response

var json = JSON.parse(xhr.responseText)

document.getElementById(“response”).InnerText = JSON.stringify(json);

}

}

# UNIT TESTING:

. JS doesn’t have any inbuild compiler to detect errors, we can only find them directly in the browser. So unit testing is very important during coding in the Js. Keeping console.log() statements in the code in every function is lengthy process so automated testing JASMINE is used. Test suits created are executed with the help of karma.

Eg for test suite:

describe('TotalTravelFare calculation Suite:',function(){

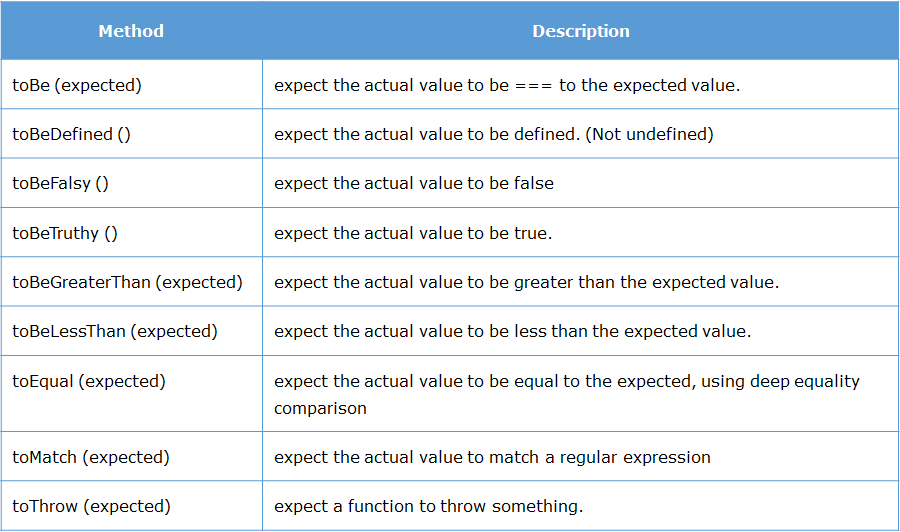
it('Test Case 1: Inputs are correct',function(){

expect(totalTravelFare(1000,20)).toEqual(1200);

});

})

describe,it,expect is global jasmine function, toEqual are Jasmine matchers. Others are



Can have not like – it(“sd”,function(){expect(12).not.toBeLessThan(10)})

To maintain resources and not to invoke or access common functions or data we use beforeEach and afterEach

beforeEach(function(done) {

setTimeout(function() {

string = 'Stormhold';

}, 1500);

});

describe('Value of n', function() {

var n = 0;

afterEach(function(done) {

setTimeout(function() {

n++;

done();

}, 1500);

});

it('is 0', function() {

expect(n).toEqual(0);

});

it('is 1', function() {

expect(n).toEqual(1);

});

});

After the test cases are written in .js file to run those we need KARMA (Test runner tool)

npm install -g karma jasmine-core

Karma is:

* A tool that spawns a web server which executes the source code against the test code for each browsers connected.
* When executed, it automatically captures the browser specified by the developer during Karma configuration.
* It then displays the results on the command line.
* It watches all the files specified within configuration file and if there are any changes, it will trigger the corresponding spec again on the browser.

Conclusion: We create test cases using jasmine global functions and run them on KARMA tool by command ( karma init ) to get karma.config file with all dependencies followed by (karma start).

# Document Object Model

DOM API – Using the DOM API we can interact with the HTML elements and the browser window using JavaScript. DOM stands for Document Object Model. It models the HTML document into an object. That means, everything we see in the HTML page is represented as objects in JavaScript.

**DOM**is not a programming language, it is an interface through which JavaScript access the elements of HTML pages.

One of the most common method associated with the document object is:

**getElementById(id):** It is used to access element by its id.

**getAttr("attributeName"):**It is used to access the tag attribute of the DOM object

**innerHTML:** innerHTML is used to set or get the HTML content of the element.

**innerText:** innerText is used to set or get the Text content to element.

**value:** value is used to get or set value of the element.

**alert("message")** - this will display an alert box with the given message

**confirm("message")** - this will display an confirm box with the given message

**write("message")**- this will overwrite the existing HTML content of the page with the message given

console.log(document.getElementById("username").getAttribute("placeholder"));

console.log(document.getElementById("password").value)

//All elements are also stored as arrays. We can also access based on the index position.

console.log(document.forms[0].elements[1].getAttribute("maxlength"))

directly like document.formName.fieldName.value

<form><input type=”text” name = “iname” id=”ID”></form>

Document.form.iname.value

Also we can get by: const field = document.querySelector(‘.className or id’) (it runs a query and gives the first matching css class or id with string passed) to return all use querySelectorAll

Field.addeventListener(‘click’, () => { }} (will invoke if field is clicked)

Field.classList.add(‘new class’)v(adds new class to the field)

We can get details of the element on which the event took place using **event.target**

<input type="radio" name="gender" value="male" onclick="display(event)">Male

<script>

function display(e){

console.log(e.target.name);

console.log(e.target.value); }

</script>

Suppose there is Div1 which has onclick event and has child Div2 inside it which also have onclick event then if we click on div2 then both div2 click event followed by div1 click event gets invoked this is called **Event Bubbling**. If you wat to prevent this we can use **event.preventDefault()** .

Tip: Just to make page reactive use container/fluid class at begging and then create div with class row and inside it use classes col-md-4 etc then create forms etc in it

# Session Storage

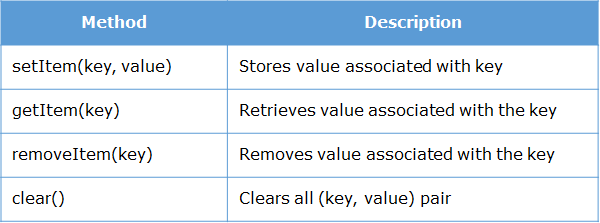
Session storage is a Web Storage API introduced in HTML5

We can store the data using web storage in two ways:

1. For particular session i.e. **session**storage: For session storage, *sessionStorage*object is used
2. Across sessions i.e. **local**storage: For local storage, *localStorage*object is used

Both these objects are created by JS runtime engine of browser.

Web storage API provides following four methods for managing data:



# Notification

We can create desktop notifications in JavaScript using the Notification API

To gain access from user: Notification.requestPermission()

To display notification:

1. function notify(){
2. Notification.requestPermission();
3. if(Notification.permission === "default"){
4. alert("Please grant permission");
5. }
6. else {
7. var notify = new Notification("New Mail",{body:"You have 1 unread email"});
8. }
9. }
10. Possible values for permission are denied,granted,default, states current choice

6. body – Defines notification body

# Cross-site scripting (XSS)

Cross-site scripting (XSS) is a security bug that can affect websites through injecting code

Eg: <img src = 1 onerror=”s=document.createElement(‘script’);s.src=’../evil.js;document.body.appendChild(s)’”>

<img src = x onerror = “alert(document.cookie)”> //gives session storage id

This can be prevented by running regEx on every input fields we provide before processing that data(validating input), using a template with context-aware by auto escaping or manually doing the escaping on every input data (<script>alert(‘hello’)</script> is converted into &lt;script&gt;alert(‘hello’) &lt;/script&gt; this is called escaping which is encoding actually)

Objects:

Objects are mutable, you cannot create a replica of it

Eg: var emp = {name : “Pavan”}

Var x = emp; // this is calling emp by x, not creating a new x

x.name = “XYZ” // this will change both the emp.name and x.name value

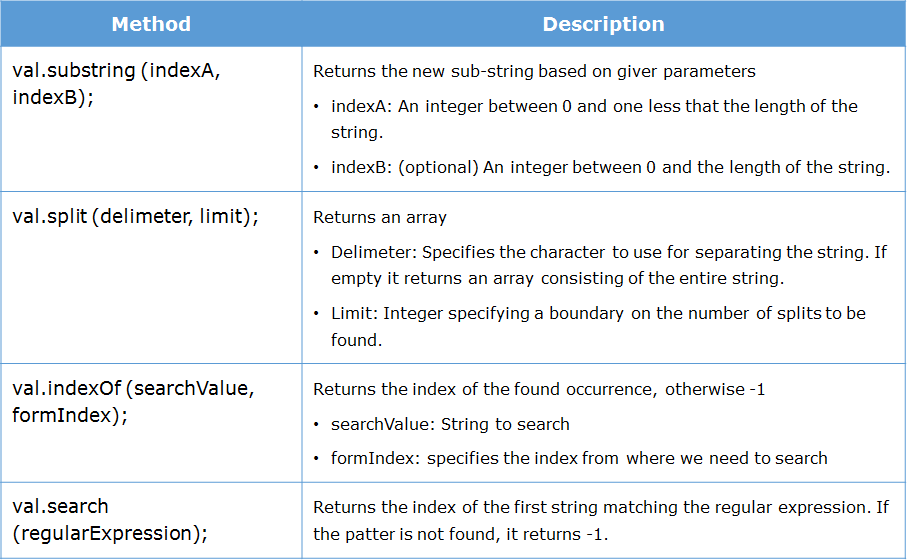
IF we want to change the key name just assign old key value to new key value and delete the old one

Eg: emp.fullname = emp.name

Delete emp.name

To know whether object emp has ‘example’ property then emp.hasOwnProperty(‘example’) returns true or false.

# String operations



# WEB Worker

When we need to do I/O operations or run scripts in the background simultaneously then we use web workers.

Eg: this is worker.js file which sends message through **postMessage** method of var i continuously with 500millisec gap

var i = 0;

function countNumbers(){

if(i < 100000){

i = i + 1;

postMessage(i);

}

setTimeout("countNumbers()", 500); // Wait for sometime before running this script again

}

countNumbers();

HTML page to use this

<html>

<script type="text/javascript">

if(window.Worker){

var worker;

worker = new Worker("worker.js");

worker.onmessage = function(counter){

document.getElementById("countervalue").innerHTML = counter.data;

};

} else{

alert("Web worker is not supported in your browser");

}

</script>

</head>

<body>

<div id="countervalue">

<!-- Messages received from web worker will be displayed here -->

</div>

</body>

</html>

# Closures

If a function takes another function as parameter then it is called Higher order functions. If functions have another functions inside it then it is called closures.

A closure is an inner function that has access to the variables present in the **lexical scope**(parent scope) of the outer/containing function. Therefore it can outlive the lifetime of the outer function and hence can continue to access the variables of the outer function.

function outer() {

var count = 0;

function inner() {

count=count + 1;

return count;

}

return inner;

}

private\_function=outer();

console.log(private\_function()); // 1

console.log(private\_function()); // 2

console.log(private\_function()); // 3

Here we can access inner function without invoking outer thus it can outlive outer fn.

# **Immediately Invoked Function Expression (IIFE)**

If function(mostly anonumous) is invoked immediately after it is created then it is call IIFE

Eg: (function (num1, num2){return num1 + num2;})(100, 200) // 300

So if we invoke () immediately after the anonumous fn then it will run and gives result.

Another example:

var outer = (function () {

var counter = 0;

return function () { return counter += 1; }

})(); // var outer = (...an anonymous function...)();

console.log(outer()); //1

console.log(outer()); //2

console.log(outer()); //3

here also anonymous fn gets invoked by () in the end and it returned a function which gets called in the console.log by outer().

Note: IIFE means ()() it should not be ();()

# Object with Function

An object can be created by using **constructor**function. The syntax is:

function Name(param1,param2,param3) {

this.param1 = param1;

this.param2 = param2;

this.param3 = param3;

this.swipeIn=function(){console.log("Swipe in by "+this.param1);}

};

var obj = new Name(p1,p2,p3);

# ParseInt and ParseFloat

Used to convert string to intiger

parseInt("5"); // 5 parseFloat("1.23"); // 1.23

parseInt("5.5"); // 5 parseFloat(".123"); // 0.123

parseInt("5p0"); // 5 parseFloat("1.23abc"); // 1.23

parseInt("p50"); // NaN (Not a Number) parseFloat("a123") // NaN

tip: typeof(NaN) = number //NaN = 0

typeof(3.14) = number //there is no type called float

# NODE and Express

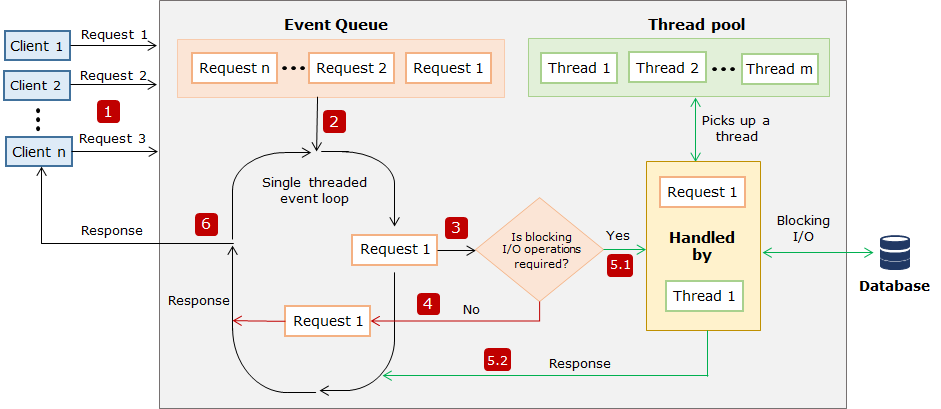
**Node**.**js**® is a open-source JavaScript runtime built on Chrome's V8 JavaScript engine.

Express is the most popular framework for creating web applications in Node.js. It is lightweight and provides easy connectivity with database like MySQL and MongoDB. Using Express we can handle requests, create views and manage routes.

**Reduced resources**, **Library support (NPM), Asynchronous and Event-driven**, **Wide client side (Angular, react) and database connectivity (Mongo, MYSQL)**

* V8 is Google's open source high-performance JavaScript engine, written in C++ and used in Google Chrome, designed to improve the performance of the JavaScript execution in browsers.
* At the time of execution, it converts the JavaScript code into machine code using **Just-In-Time (JIT) Compiler**, to achieve high speed. Therefore byte code or intermediate code will not be created

Node.js environment is created based on **Single Threaded with Event Loop Model** which is built using JavaScript's callback mechanism.



To begin with we create package.json file with npm init command with all the basic info(version,author, dependencies etc) then we install required like npm i directly or npm i express body-parser.

# Modularization

Modularization is a software design technique in which the functionality of a program is separated into independent modules, such that each module contains a desired functionality.

Syntax to export: exports.<module\_name> = <module\_implementation>

export.add = (x,y){return x+y} //just like in js export class abc or export function xyz() or export default fn e()

const myCalculator = require('./calculator'); //just like in js import e,{abc,xyz} from ‘filepath’

myCalculator.add(1,2) //3

if ther is one export then we can use module.exports = add so if we import by require in another file it will directly refer to add i.e myCalculator(1,2) // 3

# Modules

These modules are used first by assigning them to local variable by required() method

Eg: const express = require(‘express’);app = express(); const fs = require(‘fs’)

Nodemon – it is a package which will automatically refresh the server if any changes occur (just like how ng serve and autosave works in angular), we install it just like express and other modules(npm i nodemon -g) and run the filesby command npx nodemon app.js in cli instead of node app.js

fs – this module is used to read,write,append a log file (error , request logger)

fs.writeFile(file\_path, data, callback) , appendFile() (write will overwite the content and creates new if not exist) (callback function is invoked eventhough the write is success or not)

eg: const fs = require('fs');

let fileWrite = () => {

let str = "Hey, there.. ";

fs.writeFile('./log.txt', str + " At: " + new Date().getFullYear(), (err) => {

if (!err)

console.log('Data Written!');

else

throw err;

})

}

fileWrite();

eg: const fs = require('fs');

fs.readFile('demo.txt', 'utf8', (err, content) => { // charset is optional

if(!err)

console.log(content);

else

throw err; });

## http: (webserver module)

permits us to send data over the HTTP protocol, is used to build a HTTP server

let server = http.createServer((req, res) => {

res.write('Hello World! I have created my first server!');

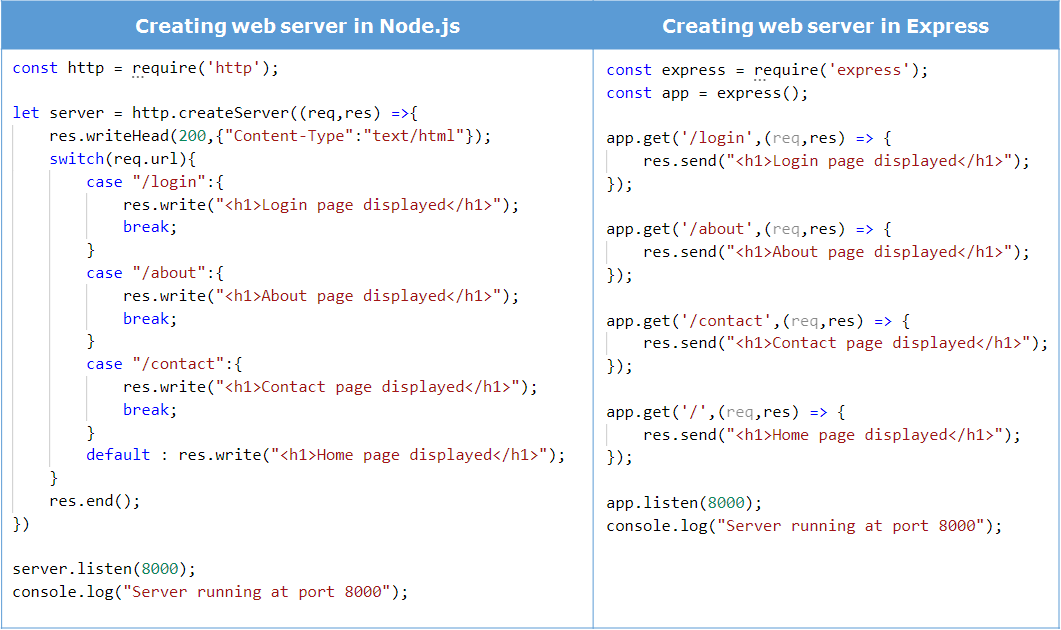
res.end();

})

server.listen(3000);

console.log("Server started... Running on localhost:3000")

this is creating server in node.js, later this entire thing is replaced by express with simple app.listen(3000) (app.listen takes a callback just like fs.readfile above which is optional)



if we send res.send(“<i>italic</i>”) then it will take it as plain text not as html so to solve this we use response.writeHead(200,{"Content-Type":"text/html"}); now it will print as *italic* , for express it is res.set('Content-Type', 'text/html');

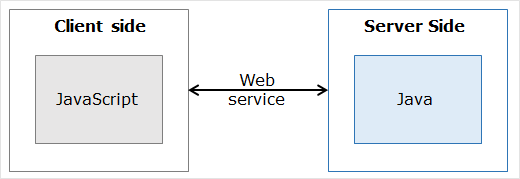
Tip: http and fs are built in modules(provided internally by node.js) directly used by required() but a NPM(provided externally by node.js) module like express has to install and use it

npm install <package\_name>[@<version>] //version optional takes latest

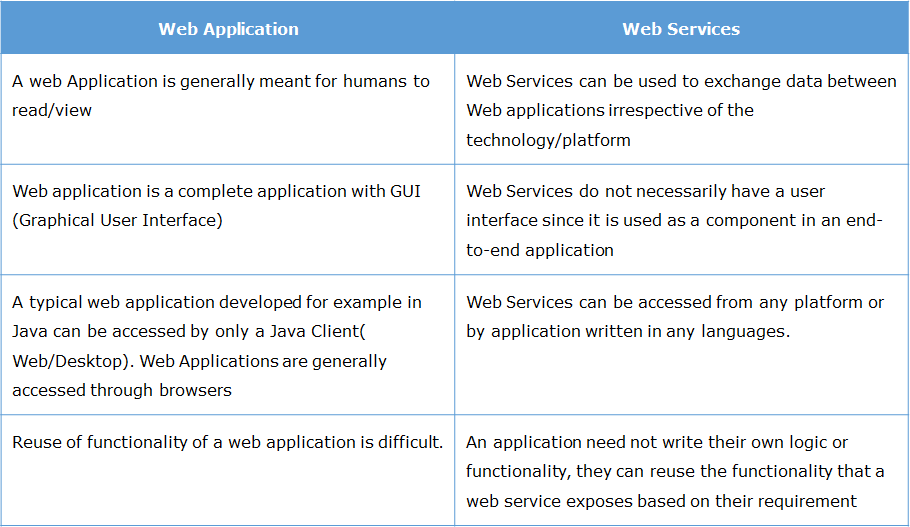
This will create a folder **node\_modules** in the current directory and put all the packages related files inside it.

# Express

Express is a **layer** built on Node.js which helps us to **manage our web server**and **routes**.



If any two different platforms want to communicate or transfer data it can happen through webservice. So http using www is a webservice, google maps in other apps is acheivble through webservice.



# RESTful API

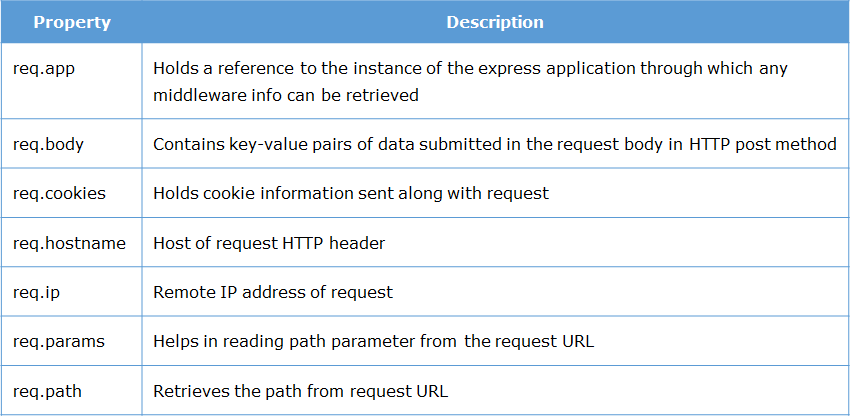
Web services can be classified into two types based on the transfer of data.

* SOAP based web service (XML based)
* RESTful web service (REST  stands for  "***Representational State Transfer***". It is an architectural style for the web which uses HTTP Protocol for communication.)

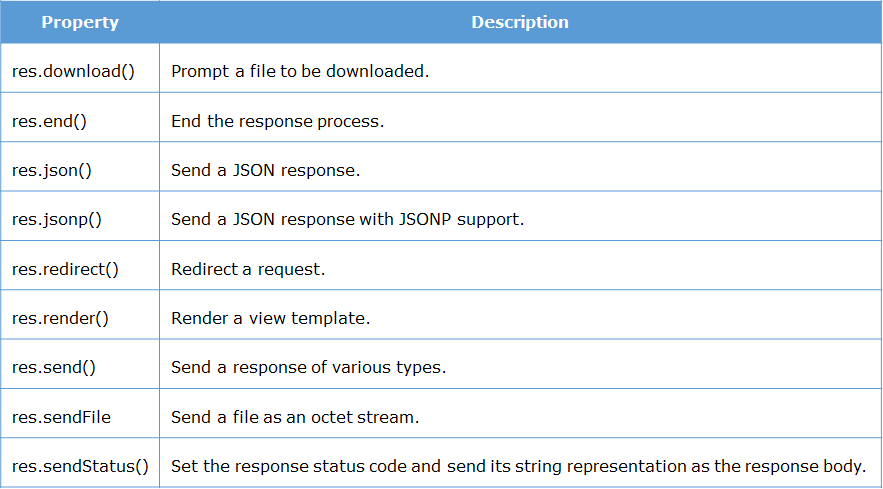
RESTful API is typically used to refer to web services implementing REST architecture.

In RESTful, resources are manipulated using GET, PUT, POST and DELETE methods of HTTP to perform read, update, create and delete operations on the resource.

Objects which can be accessible with request from client



Response objects



Also res.type()

To send a file through response , eg for res.sendFile()

Eg: const path = require(‘path’)

app.get('/register', (req, res) => {

console.log('request for register.html file');

res.sendFile(path.join(\_\_dirname + '/views/register.html')); });

// to join the paths we used path, cause generally outer paths in server changes. \_\_dirname is a predefined variable that provides the absolute path of the resource in which it is used.

Query strings are the data appended as part of request URL. Query strings begin with a question mark and the name-value pair that are separated by an '&'(ampersand) .

'***http://localhost:3000/login?username=john&email=john%40infy.com&login=Login***', (%40 is encoded @)

app.get('/login',(req,res) => {

res.send(`<h1>Hello, ${req.query.username}</br>Your email - ${req.query.email} is registered</h1>`);

});

request.query.<querystring name-value>

# Routing

if we use app.get for so many routes it will be messy so we use router class in a separate router.js file with all routes at one place

const router = express.Router();

router.get('/', (req, res) => { res.send('home page') });

we call this router in main app.js file as a microservice i.e app.use(‘/’,router)

main advantage is we can use router method for different alike routes

const express = require('express');

const userRouter = require("./routes/user\_router.js");

const adminRouter = require("./routes/admin\_router.js");

const app = express();

app.use('/user', userRouter);

app.use('/admin', adminRouter); //code is clean and maintainable this way

next() this is used for invoking next matching handler

app.get('/login',(req,res,next) => {console.log('Request to display login page has received');

next(); }); //invokes response.send

app.get('/login',(request,response) => { response.send('Login Form'); });

**Array Syntax For chaining of routes Handlers**

let handler1 = (req, res, next) => { console.log('Request to display login page has received');

next();}

let handler2 = (request, response) => { response.send('Login Form');}

app.get('/login', [handler1, handler2]);

or we can directly write route handlers back to back separated by comma

app.get('/login', (request, response,next) => { console.log('Request to display login page has received');

next(); } ,

(request, response) => { response.end(); }); //both handles passed as parameters.

We can group multiple HTTP methods and their handlers, for the same route path using route(), this method can also be used with both app.route() or router,route()

router.route('/login')

.get((req, res) => { res.send('display login form'); })

.post((req, res) => { res.send('process login form'); })

**Note:** body-parser is a Node.js body parsing middleware which parse incoming request bodies and makes the data readable under the **request.body** property. Without this middleware, data sent through POST request is **inaccessible**. We use it as middleware. i.e app.use()

const bodyParser = require("body-parser");

app.use(bodyParser.json());

app.get() etc takes numerous functions are parameters like app.get(‘/login’,requestLogger,(req,res)=>{},errorLogger). These functions will be invoked by next()

# Middleware

By middleware we can run set of code in another place by app.use() method. next() method is used to invoke or pass the handler to the next middleware.

app.use(PATH,CALLBACK)… Path is optional here.

## User-defined middleware

let mylogger = (req,res,next) => {

console.log('Request method is '+ req.method);

console.log('Request url is '+ req.url);

next(); }

app.use(mylogger);

app.get('/adminLogin',(req,res) => {

res.send("Admin login Success"); });

app.get('/userLogin',(req,res) => {

res.send("User login Success"); });

app.get(‘\*’,(req,res)=>{ // for other than adminlogin and userlogin

err = new Error(‘unknown link’);

err.status = 404;

next(err); // this will invoke the next match err handler (u can also throw err which will

}) be handled by next error handler) err name can be anything no need to match

app.use((err,req,res,next)=>{ res.send(‘Error: ’ + err.status + ‘occured’)}) // this will get called if error exists, it’s a error handler so placed in the end. Only gets called if err value exists.

Note: error can be captured only if first parameter is error (can be any name like err, er or e etc)

Eg: app.use((myerror,req,res,next)=>{ res.send(‘Error: ’ + myerror.status + ‘occured’)}) //works

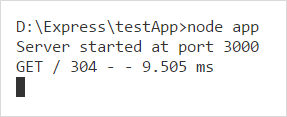
In the same way during throwing error or next(error) the var error name can be anything, no need to match with the error handler parameter, it is only captured by first parameter by error handler.\* If no error is invoked then this handler method is not invoked, so if we keep any simple console.log() this wont execute if there is no error invoked \*

## Third Party middleware

Morgan – can be used to log requests

Eg: const morgan = require(‘morgan’);

app.use(morgan('tiny'));



Express-error-handler – can be used to log error stacks

const errorhandler = require('express-error-handler');

app.use(errorhandler);