TOPSTECHNOLOGIES

Training | Outsourcing | Placement | Study Abroad

Node JS Course Slide

Modules

Module 1 - [Node - JavaScript Fundamental]

Module 2 - [Node - NodeJS - Introduction]

Module 3 - [Node - Web Development with Node]

Module 4 - [Node - Node with Mongo dB]

Module 5 - [Structuring REST API]

Module 6 - [Node - [API Authentication & Security]]

Module 1

Node - JavaScript Fundamental

JavaScript Introduction

JavaScript was created by Brendan Eich

It came into existence in September 1995, when Netscape 2.0 (a web browser) was released

JavaScript was designed with a purpose to make web pages dynamic and more interactive

JavaScript Features

It can be used for client and server applications

It is platform independent which means it can run on any operating systems

JavaScript codes are needed to be embedded or referenced into HTML documents then only it can run on a web browser

It is an interpreted language

It is a case-sensitive language and its keywords are in lowercase only

Difference between Java & JavaScript

Java and JavaScript both are completely different languages

Java is general purpose Object Oriented language from Sun Microsystems JavaScript is

Object based Scripting language

Script refers to short programming statements to perform a task

Client Server Model

- **Node** - Node is a component or terminal connected to a network. The components like laptops, PDAs, Internet enabled mobiles etc., can be considered as node in a computer network
- **Client** - It is a node computer that establishes connection with the server, collects data from the user, sends it to the server, receives information from the server and presents it to the user
- **Server** - In the context of client-server model, server is the counter part of client. It is a computer that serves queries from the client. The programs which respond to the request of clients are known as server applications. The computer designed to run server application is known as server machine. Web server, database server and mail server are some examples of servers.

Server Side JavaScript - Server-side JavaScript is also known as "**LiveWire**". Like client-side JavaScript, server-side JavaScript is also embedded within a HTML document.

Client Side JavaScript - Client-side JavaScript refers to JavaScript code that gets executed by the web browser on the client machine.

Getting Started With JavaScript

```
<script [Attributes = [“Value”] . . . ]>
  ◦ ... JavaScript statements...
</script>
```

Attributes	Value	Description
Type	text/javascript text/ecmascript text/vbscript	The type of script
Language	Javascript vbScript	Name of scripting language
Src	URL	a URL to a file that contains the script

Placing JavaScript Code

Embedded/Inline JavaScript : JavaScript code can be placed either in the HEAD or in the BODY section of a HTML document.

It is advised to place JavaScript code in HEAD section when it is required to be used more than once.

If the JavaScript code is small in size and used only once, it is advisable to put it in the BODY
section of the HTML document.

External JavaScript : In case, same JavaScript code needs to be used in multiple documents then it is the best approach to place JavaScript code in external files having extension as "**.js**". To do so, we will use **src** attribute in **<SCRIPT>** tag to indicate the link for the source JavaScript file

Statements in JavaScript

Statements are the commands or instructions given to the JavaScript interpreter to take some actions as directed

A JavaScript interpreter resides within almost Internet browsers

Example

```
a=100;

b=200;

c=a+b;

document.write("Sum of A and B : "+c);
```

Comments

Single line comment using double-slash (//).

Multiple lines comment using /* and */

Example

```
//This is Single Line Comment

/*

This is Multi-line Comment.

It can be of any length.

*/
```

Literals

Literals refer to the constant values

Example

```
a=10;

b=5.7;

document.write("Welcome");
```

In above statements 10, 5.7, “Welcome” are literals

Identifiers

Identifiers refer to the name of variables, functions, arrays, etc. created by the programmer

It may be any sequence of characters in uppercase and lowercase letters including numbers or underscore and dollar sign

An identifier must not begin with a number and cannot have same name as any of the keywords of the JavaScript

Valid Identifiers	Not Valid Identifiers
RollNo	to day
bus_fee	17Nov
_vp	%age
$amt	

Reserved Words or Keywords

Reserved words are used to give instructions to the JavaScript interpreter and every reserved word has a specific meaning.

These cannot be used as identifiers in the program.

This means, we cannot use reserved words as names for variables, arrays, objects, functions and so on.

These words are also known as “**Keywords**”

Variables

A variable is an identifier that can store values.

These values can be changed during the execution of script.

Once a value is stored in a variable it can be accessed using the variable name.

Variable declaration is not compulsory, though it is a good practice to use variable declaration.

Keyword **var** is used to declare a variable

We should use meaningful name for a variable.

Data Types

Number : The number variable holds any type of number, either an integer or a real number. Some examples of numbers are : `29, -43, 3.40, 3.4323`

String : A string is a collection of letters, digits, punctuation characters, and so on.

A string literal is enclosed within single quotes or double quotes (‘ or “).

Examples of string are : `'welcome', "7.86" , "wouldn't you exit now", ' country="India“ '`

Boolean : A boolean variable can store only two possible values either true or false.

Internally it is stored as 1 for true and 0 for false.

It is used to get the output of conditions, whether a condition results in true or false.

Arrays : An array is a collection of data values of same types having a common name.

Each data element in array is referenced by its position in the array also called its index number.

Array Declaration :

```
var a = new a( );
var x = [ ];
var m = [2,4,sun];
```

Null Value : JavaScript supports a special data type known as null that indicates “no value or blank”.

Note that null is not equal to 0.

Example :

```
var distance=new object();

distance=null;
```

Objects

JavaScript is an object based scripting language.

It allows us to define our own objects and make our own variable types.

It also offers a set of predefined objects.

The tables, forms, buttons, images, or links on our web page are examples of objects.

The values associated with object are properties and the actions that can perform on objects are methods or behaviour.

Document Object

The Document object is one of the parts of the Window object.
It can be accessed through the **window.document** property.

The document object represents a HTML document and it allows one to access all the elements in a HTML document

Properties	Purposes
Title	Returns / set title of the current document
bgColor	Returns / set the background color of the current document
fgColor	Returns / set the text color of the current document
linkColor	returns/ sets the color of hyperlinks in the document
alinkColor	returns/ sets the color of active links in the document
vlinkColor	returns/ sets the color of visited hyperlinks
URL	returns a string containing the URL of the current document
Location	to load another URL in current document window.

Date Object

This object is used to set and manipulate date and time

JavaScript dates are stored as the number of milliseconds since midnight, January 1, 1970.

Dates before 1970 are represented by negative numbers.

A date object can be created by using the new keyword with Date()

Syntax :

```
new Date();
new Date(milliseconds);
new Date(dateString);
new Date(yr_num, mo_num, day_num, [hr_num, min_num, sec_num, ms_num]);
```

```
today = new Date();
dob = new Date("October 5, 2007 12:50:00");
doj = new Date(2007,10,5);
bday = new Date(2007,10,5,12,50,0);
```

Methods	Meaning
getDate()	Returns day of the month
getDay()	Returns the day of the week
getFullYear()	Returns the full year
getHours()	Returns the hour
getMinutes()	Returns the minutes
getMonth()	Returns the month
getSeconds()	Returns the seconds
getTime()	Returns the numeric value corresponding to the time
getYear()	Returns the Year

Math Object

This object contains methods and constants to carry more complex mathematical operations.

This object cannot be instantiated like other objects

All properties and methods of Math are static

Properties	Description
Math.PI	Returns the value p
Math.E	Euler's constant and the base of natural logarithms
Math.LN2	Natural logarithm of 2
Math.LN10	Natural logarithm of 10, approximately 2.302
SQRT1_2	Square root of ½
SQRT2	Square root of 2

Method	Description
pow(x, p)	Returns X^p
abs(x)	Returns absolute value of x
exp(x)	Returns e^X
log(x)	Returns the natural logarithm of x
sqrt(x)	Returns the square root of x
random()	Returns a random number between 0 and 1
ceil(x)	Returns the smallest integer greater than or equal to x
floor(x)	Returns the largest integer less than or equal to x
min(x, y)	Returns the lesser of x and y
max(x, y)	Returns the larger of x and y
round(x)	Rounds x up or down to the nearest integer
sin(x)	Returns the sin of x, where x is in radians.
cos(x)	Returns the cosine of x, where x is in radians
tan(x)	Returns the tan of x, where x is in radians

Expression and Operators

Expression is combination of operators operands that can be evaluated.

Examples :

x = 7.5 // a numeric literal

“Hello India!” // a string literal

false // a Boolean literal

{feet:10, inches:5} // an object literal

[2,5,6,3,5,7] // an array literal

v= m + n; // the variable v

tot // the variable tot

Arithmetic Operators

Operator	Descriptioin
+	Addition
-	Subtraction
*	Multiply
/	Division
%	Modulo
++	Increment by 1
--	Decrement by 1

Assignment Operators

Shorthand Operators	Example	Is equivalent to
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Relational Operators

Operator	Description	Example
==	Is equal to	4==8 returns false
!=	Is not equal to	4!=8 returns true
>	Is Greater than	8>4 returns true
<	Is less than	8<4 returns false
<=	Is less than or equal to	8<=4 returns false
>=	Is greater than or equal to	8>=4 returns false

Logical Operator

Operators	Description
&& (AND)	Returns true if both operands are true else it return false
| | (OR)	Returns false if both operands are false else it return true
! (NOT)	Returns true if the operand is false and false if operand is true

Popup Box - Alert

An alert box is used if we want to display some information to the user. When an alert box appears, the user needs to click “OK” button to proceed

Confirm Box

Confirm box returns a Boolean value.
If the user clicks on “OK”, it returns true. If the user clicks on “Cancel”, it returns false

Prompt Box

Prompt box allows getting input from the user.

We can specify the default text for the text field.

The information submitted by the user from prompt( ) can be stored in a variable

Example :

```
var name = prompt("What's your name? ", "Your name please…");
document.write(name);
```

Branching with If Statements

An if statement is used to execute a statement or a block of statements on based of logical expression (condition)

There are three different forms

- If… statement

```
if(condition){
... statements to be executed if(condition) is true...
}
```

- If… else statement

```
if(condition)
      {
  True statement(s)...
  }
  else
  {
  False statement(s)...
  }
```

- If … else if … statement

```
 if(condition1) {
 code to be executed if condition1 is true
}
else if(condition2){
code to be executed if condition2 is true
}
.
Else {
 code to be executed if any of the conditions is not true }
```

Selection with Switch statement

A switch statement is used to execute different statement based on different conditions

It provides a better alternative than a long series of if… else if … statements

Syntax :

```
switch(expression){
  case label1://executes when value of exp. evaluates to label
  statements;
  break;
case label2://executes when value of exp. evaluates to label
  statements;
  break;
...
}
```

Loop Statements

For : -

The for loop consists of three optional expressions separated by semicolon, followed by a block of statements executed in the loop.

Loop statements executed repeatedly again and again until the condition is false.

The for loop is used when we know in advance how many times the script code should run

```
for([initial-expression]; [condition]; [increment-expression])
{
  statements
}
```

While :-

The while loop statement is simpler than the for loop. It

consists of a condition and block statement.

The condition is evaluated before each pass through the loop. If the condition is true then it executes block statement.

```
while (condition)
{
statements
}
```

Do...while

The do...while loop is much like a while loop.

It will repeat the loop until the specified condition is false.

This loop will always be executed at least once, even if the condition is false, because the block of statements is executed before the condition is tested.

In this loop statement, curly braces are optional

```
do
{
statements
}
while (condition);
```

Label

A label is an identifier followed by a colon that can be helpful in directing the flow of program

```
label: statement
```

The value of label may be any JavaScript identifier. The statement that you identify with a label may be any statement.

BREAK

Break statement is used to exit from the innermost loop, switch statement, or from the statement named by label. It terminates the current loop and transfers control to the statement following the terminated loop

```
break [label]
```

The break statement includes an optional label that allows the control to exit out of a labeled statement

CONTINUE

The continue statement skips the statement following it and executes the loop with next iteration.

It is used along with an if statement inside while, do-while, for, or label statements

```
continue [label]
```

Object Handling Statements : for... in

The for... in statement iterates a specified variable over all the properties of an object.

```
var book = new Object();
book={
   Title:"Book Title", Price :
   500
};
for (var property in book)
{
   document.write(book[property]);
}
```

With

With statement saves lot of timing when property of same object have to be accessed.

```
var area, circumference; var r=10;
with (Math)
{
area = PI * r * r  circumference = 2*PI*r
}
with (document)
{
write("Area of the Circle: "+area+"<br>");
write("Circumference of the Circle: "+circumference);
}
```

Function

Function is a named block of statements which can be executed again and again simply by writing its name and can return some value.

```
function Welcome()
{
  alert("Welcome to NCERT");
}
```

OR

```
hello = function() {
  return "Hello World!";
}
```

Nested Function

```
function area(r){
    function square(x){
        return x*x;
    }
    return 3.14*square(r);
}
```

Arrow function

Arrow function introduced in ES6

```
hello = () => {
  return "Hello World!";
}
```

Return value by default

```
hello = () => "Hello World!";
```

With Parameter

```
hello = (val) => "Hello " + val;
```

Function call()

The call() predefined method in javascript

It can be used to invoke (call) a method with an owner object as an argument (parameter).

```
var person = {
fullName: function() {
  return this.firstName + " " + this.lastName;
}
}
var person1 = {
  firstName:"John",
  lastName: "Doe"
}
var person2 = {
  firstName:"Mary",
  lastName: "Doe"
}
person.fullName.call(person1);// Will return "John Doe"
```

Function apply()

The apply() method takes arguments in array

```
var person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName +
    ","
+ city + "," + country;
  }
}
var person1 = {
  firstName:"John",
  lastName: "Doe"
}
person.fullName.apply(person1, ["Oslo", "Norway"]);
```

```
var person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName +
"," + city + "," + country;
  }
}
var person1 = {
  firstName:"John",
  lastName: "Doe"
}
person.fullName.call(person1, "Oslo", "Norway");
```

Function bind()

bind method returns a method instance instead of result value, after that need to execute a bound method with arguments.

```
var person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city
+ "," + country;
  }
}
var person1
  = {
  firstName:
  "John",
  lastName:
  "Doe"
}
var
bound=person.fullName.bind(person
1); bound("Oslo", "Norway");
```

Function closures()

JavaScript variables can belong to the **local** or **global** scope.

Global variables can be made local (private) with **closures**.

A local variable can only be used inside the function where it is defined. It is hidden from other functions and other scripting code.

Variable name without declaration keyword (**var, let or const**) are always global, even if they are created inside a function

```
var add = (function () {
  var counter = 0;
  return function () {counter += 1; return counter}
})();

add();
add();
add();

// the counter is now 3
```

The variable **add** is assigned the return value of a self-invoking function

The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

This way add becomes a function.

The "wonderful" part is that it can access the counter in the parent scope

This is called a JavaScript **closure.** It makes it possible for a function to have "**private**" variables.

The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

A closure is a function having access to the parent scope, even after the parent function has closed.

Javascript classes

ES6 introduced class

A class is a type of function, but instead of using the keyword **function** to initiate it, we use keyword **class** and properties is assigned inside a **constructor()** method

The constructor method is called automatically when the object is initialized.

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
}
mycar = new Car("Ford");
```

```
class Car {
  constructor(brand) {
    this.carname = brand;}
  present() {
    return "I have a " + this.carname;
  }
  }

mycar = new Car("Ford");
document.getElementById("demo").innerHTML = mycar.present();
```

Static method

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  static hello() {
    return "Hello!!";
  }
}

mycar = new Car("Ford");

//Call 'hello()' on the class Car:
document.getElementById("demo").innerHTML = Car.hello();

//and NOT on the 'mycar' object:
//document.getElementById("demo").innerHTML = mycar.hello();
//this would raise an error.
```

inheritance

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model;
  }
}

mycar = new Model("Ford", "Mustang");
document.getElementById("demo").innerHTML = mycar.show();
```

Getter and setter method

```
class Car {
  constructor(brand) {
    this._carname = brand;
  }
  get carname() {
    return this._carname;
  }
  set carname(x) {
    this._carname = x;
  }
}
mycar = new Car("Ford");
document.getElementById("demo").innerHTML = mycar.carname;
```

Javascript JSON

JSON is a format for storing and transporting data

JSON is often used when data is sent from a server to a web page.

JSON stands for **J**ava**S**cript **O**bject **N**otation

JSON is a lightweight data interchange format

JSON is language independent

JSON is "self-describing" and easy to understand

```
{
"employees":[
  {"firstName":"John", "lastName":"Doe"},
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Peter", "lastName":"Jones"}
]
}
```

Converting Json String to Object

```
var obj = JSON.parse(text);
```

Convert Javascript object into String

```
var myJSON = JSON.stringify(obj);
```

Module 2

NodeJS - Introduction

Introduction

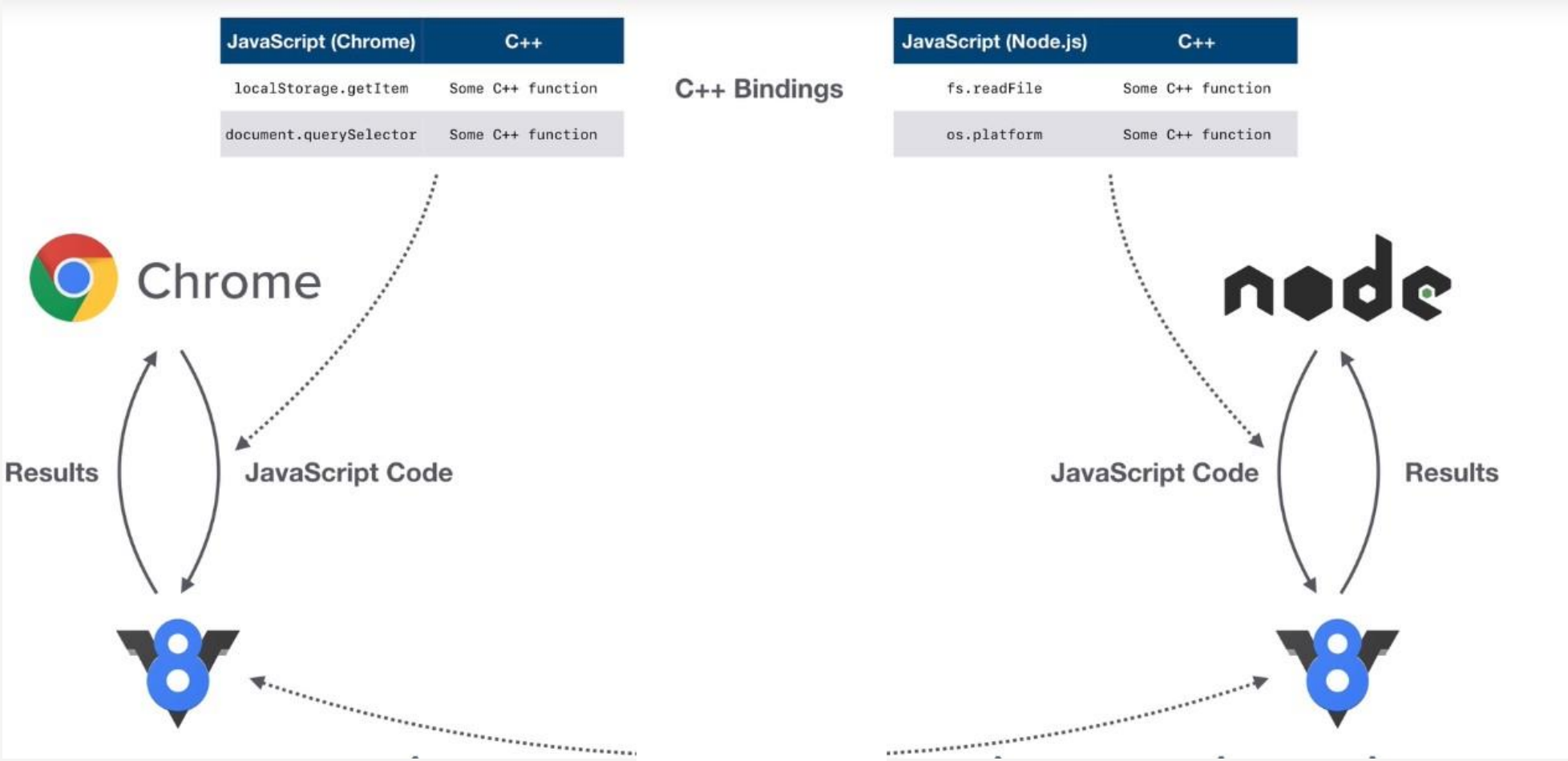
Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser.

A Node.js app is run 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 and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm

When Node.js needs to perform 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.


JavaScript (Chrome)
C++
localStorage.getItem
Some C++ function
document.querySelector
Some C++ function
C++ Bindings
JavaScript (Node.js)
C++
fs.readFile
Some C++ function
os.platform
Some C++ function
Chrome
node
Results
JavaScript Code
JavaScript Code
Results

Written in C++

Node JS module system

Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application

Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope. Also, each module can be placed in a separate .js file under a separate folder.

1. Core Modules
2. Own Modules or Local Modules
3. NPM Modules or Third Party Modules

Core Modules

Core Modules covers minimum functionality of Node.js

These core modules are compiled into its binary distribution and load automatically when Node.js process starts

You only need to import the core module first in order to use it in your application

Loading Core Modules

```
var module = require('module_name');
```

- Specify module name in the require() function.

- The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns

Example

Node js File Module System

Own Modules or Local Modules

Local modules are modules created locally in your Node.js application.

These modules include different functionalities of your application in separate files and folders.

You can also package it and distribute it via NPM, so that Node.js community can use it.

We need **module.exports** to define what could be access from other file

Importing our own module Example

NPM Modules or Third Party Module

There are many modules available online which could be used in Node.js.

Node Package Manager (NPM) helps to install those modules, extend them if necessary and publish them to repositories like Github for access to distributed machines

To start working with npm module

1. we have to initialize npm in our application

   npm init

2. we have to install all module we want to use

   npm install validator@version

Validator Example

-Sometimes when download or refer code online there is no **node_modules** directory but it required to run code

npm install

- This command will check json file and download all dependency

- Also use chalk npm library for practice

Chalk Example

Global npm module & nodemon

- Search for nodemon in npm it helps to run code automatically as we save code

npm i nodemon@version –g

- Here –g for Global install in OS

nodemon app.js

- To run script

Example

File System & Command line arguments

- We can take input from user using command line arguments

node app.js Ankit

- process.argv is Global object like console

console.log(process.argv)

- so we can pass value from command line like what operation we want to perform

node app.js add

Command Line Arguments Example

Arguments Parsing with Yargs

Now in above example

node app.js add –title=“This is Title”

This will print same text that we pass through commands

We need to parse the value for that we can use yargs

npm install yargs@version

As shown in code snippet for adding note we can also write code for removing note, listing notes, and reading note similarly

Yargs Example

Storing data with json

- Convert JavaScript object to Json String using JSON.stringify() method.
- Convert Json string to JavaScript object using Json.parse() method

Json Write into File Example

Write Note into File Example

Read data from file

- Read data from file
- **readFile()** does not return actual data

Example

Add | Remove | Save | Load Notes

Example

Add | Remove | Save | Load Notes

Example

Filter vs Find in Javascript

In addNote() code we did previously filter will go through each note from starting to end suppose we have 1000 notes and duplicate note found on 10th position still filter will go through each note till 1000.

So find is better than filter for this situation

```
Const duplicateNote = notes.find((note)=>{
        note.title===title
})
If(duplicateNote===undefined){
        // add note…
}else{
        // note already exist
}
```

Listing & Reading Notes

Example

Asynchronous Node js

Non blocking event driven

In synchronous programming code executed line by line

Example

Making http request

- register darksky.net/dev

Open Weather Map

npm install request@version

Example

Automatically Json parsing

```
const request = require ('request')
const url='...'
request ( {url:url, json:true }, (error, response)=>{
        console.log(response.body.currently)
})
```

Chrome extension json formatter

Example

Geocoding

- Process of taking address and convert it to latitude – longitude
- Mapbox start with free account
- Get token
- API documentation
- Search service -> Geocoding
- Implement in node application similar to temperature API

Example

Error handling

Example

Callback function

EX1

```
setTimeout(()=>{
        console.log('Two seconds are up')
}, 2000)
```

EX2

```
Const names = ['A','B','C']
Const shortNames=names.filter((name)=>{
        return name.length<=4
})
```

Example

- callback function required in following example

- instead of return data

```
Const geocode=(address, callback)=>{
        setTimeout(()=>{
                const data={
                        lat=0, longitude=0
                }
                return data
        }, 2000)
}
Const data=geocode('Philodophia')
Console.log(data)  // error
```

```
Const geocode=(address, callback)=>{
        setTimeout(()=>{
                const data={
                        lat=0, longitude=0
                }
                callback(data)
        }, 2000)
}
const data=geocode('Philodophia', (data)=>{
        Console.log(data)
})
```

Use callback in weather application

Put all code or function inside geocode.js then **module.exports**

In app.js

Const geocode=require(./utils/geocode.js)

Example

Reusable function for getting the forecast

1. setup the forecast function in utils/forecast.js
2. Require the function in app.js and call it
3. the forecast function should have three potential calls to callback
   - low level error, pass string for error
   - coordinate error, pass string for error
   - success, pass forecast string for data

```
forecast(-75.7800, 44.1545, (error, data)=>{
        console.log('Error',error)
        console.og('Data',data)
})
```

Callback chaining

Example

Get location from user and then geocode

1. Access the command line argument without yargs
2. use the string value as the input for geocode
3. Only geocode if a location was provided
4. Test your work with a couple of locations

Example

ES6 : De-structure object

```
const [latitude, longitude, address] = data
```

Example

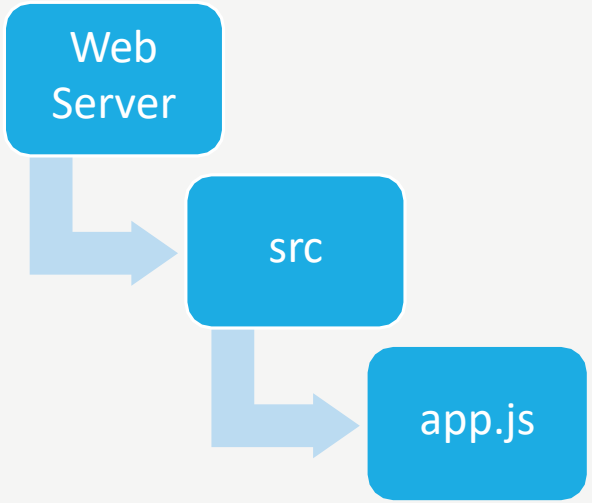
Module 3

Web Development with Node

Web server with Express

- Express makes very easy to create web server with node
- It allows to serve all of the assets to web application
- HTML, CSS, Javascript, JSON data

```
npm init –y
npm i express
```



Example

Static Assets

App.js
```
console.log(__dirname)
console.log(__filename)
```

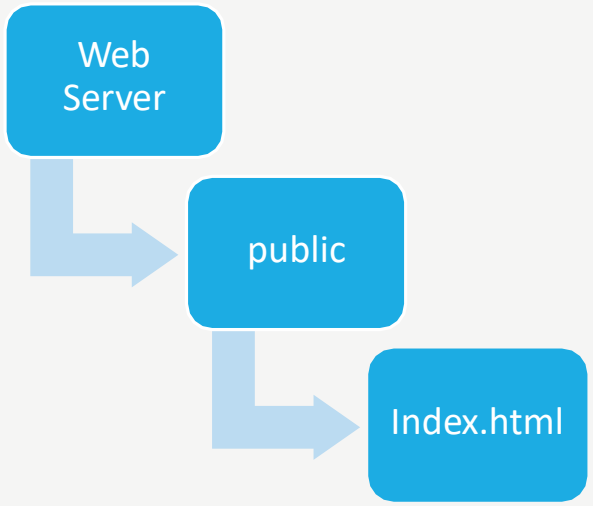
Path module

App.js
```
const path=require('path')
console.log(path.join(__dirname,',.public/)'))
```

Customize server

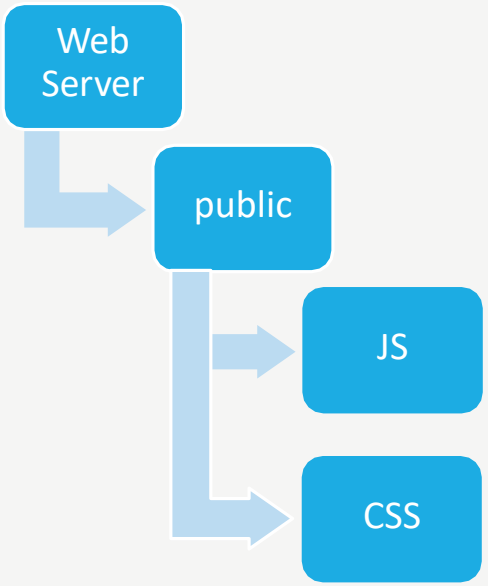
Run and Check

App.js
```
const publicDirectoryPath=path.join(__dirname,',,.public/')
app.use(express.static(publicDirectoryPath))
```



Create html pages for about.html and help.html url

Access them directly by typing about.html in url



Serving Css, JS, Images and more

Link CSS and Javascript file with index.html file.

Create public -> img directory

put images inside and display it in index.html

Static Server

Static Assets

Template Engine – Dynamic pages

- Handlebar library
- we will use hbs which is using handlebar in background

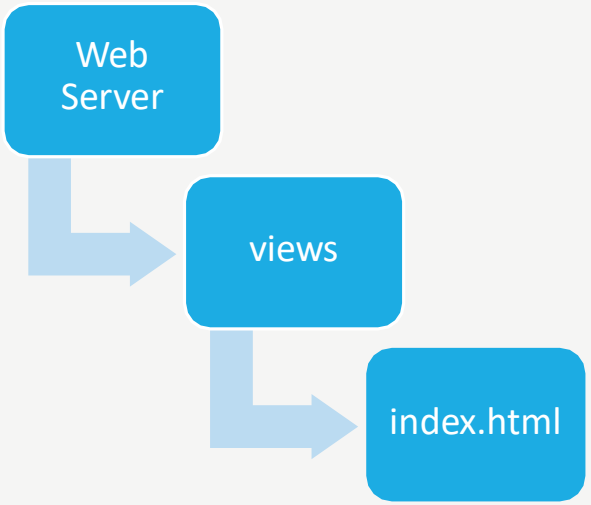
  npm install hbs

- web-server -> views -> index.hbs

  app.js

  app.set('view engine', 'hbs')

- **we have inform express about our engine using app.set() method**
- **remove index.html from public directory**



Example

Passing data to hbs file

hbs page also called template

```
app.get(‘’, (req, res)=>{
        res.render(‘index’, {
                title:’My title’
        })
})
```

index.hbs

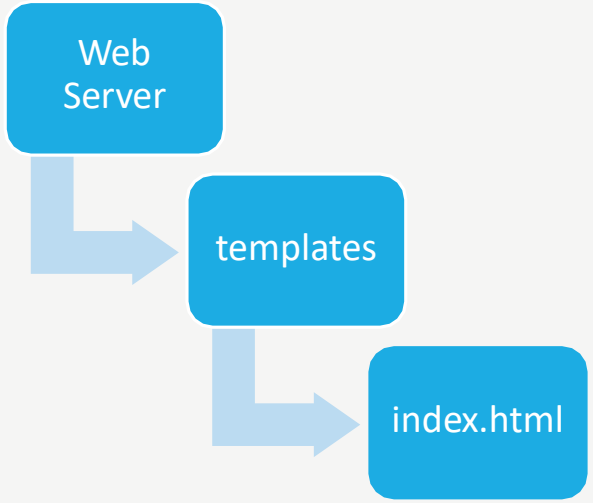
```
{{title}}
```

Repeat process for **about.hbs** and **help.hbs** and remove static html pages which are no longer required.

Customizing the views Directory

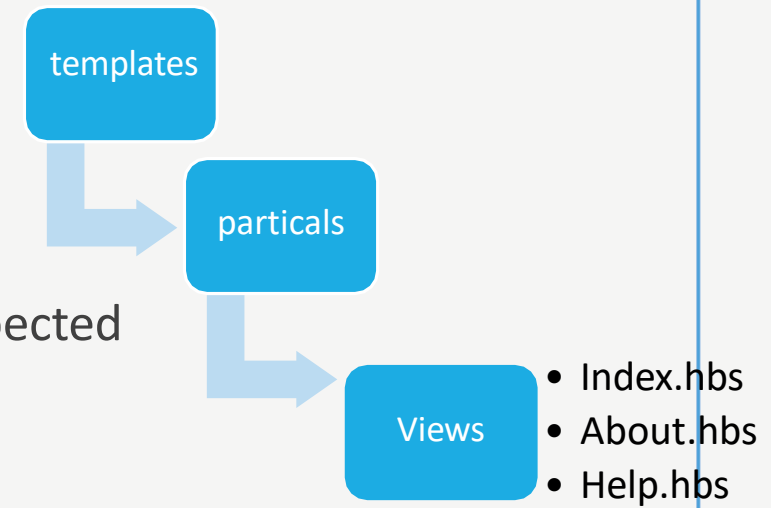
- What if I want to change views directory name and location how handlebar handle it.
- If change name views directory name then run it show **error**
- If you customize name and location you have to inform express about same.
- Rename (views -> templates)

Example



Handlebars Particles

- Pages we can reuse in multiple other pages like header, footer
- create **header.hbs** inside partical folder with some dummy text
- Example
- restart server automatic by nodemon might not show output as expected
- nodemon only get changes done inside .js file not .hbs file
- use below commad to listen changes of .hbs file




```
nodemon src/app.js –e .js , -hbs
```

Error 404 Pages

- create footer partial page and use inside all views
- set link of all views inside header partial template
- after all request handling inside app.js file
- create 404.hbs view with header & footer

404 – Example

Styles Example

Accessing API from Browser : Query String

to check query string

- console.log(req.query)

```
app.get('/products', (req, res)=>{
        console.log(req.query.search)
        if( ! req.query.search){
                return res.send({
                        error : 'Your Error message'
                })
        }
})
```

localhost:3000/products?search=games&rating=5

Weather API Request

Example

Challenge

Goal : Write up /weather

- Copy utils folder from weather-app console project and paste into web-server > src folder

- npm i request

1. Require geocode /forecast into app.js
2. Use the address to geocode
3. Use the coordinates to get forecast
4. send back the real forecast request

ES6 : default function parameter

- if we don’t pass argument to function then it take that argument undefined

-so if we set default function parameter then function will execute with that default value if don’t pass the value

Example

Default value with De-structuring

```
const transaction = ({type, label, stock=0}={})=>{
        console.log(type, label, stock)
}

transaction('order')
transaction('order','product')
```

Back to weather server app

What happen if we pass **address=!** With query string

- Server will crash

- so that we have to provide default value while de-structuring

Example

Browser HTTP Request with fetch()

- fetch is browser based api not available on node

- Use any dummy api which return JSON data

```
fetch('http://puzzle.mead.io/puzzle')
.then((response)=>{
        response.json().then((data)=>{
                console.log(data)
        })
})
```

Run and check chrome developer console (F12)

Challenge : fetch() weather

1. setup call to fetch to fetch weather for Button
2. Get the parse JSON response
   - if error property, print error
   - if no error property, print location and forecast
3. Refresh the browser and test your work

Example

Module 4

Node with Mongodb

Mongodb vs Mysql

Mongodb.com

- launch in 2009

- we can use any database with node and we can also use mongo with other technology

SQL Terminology	Mongodb (No-SQL) Terminology
Table	Document
Rows	Collections
Columns	Fields

Installing MongoDB on Windows

Download MongoDB

- We can download any file msi or zip
- Extract downloaded zip folder inside **Mongodb/bin**
- move **Mongodb** folder inside **user/{username}**
- create folder **mongodb_data**
- Open **visual Code** terminal with

  mongodb/bin/mongod.exe –dbpath=/users/{username}/mongodb-data

- leave terminal running

Install Database GUI viewer

Mongodb Admin tool

Download **Robo 3t not** Studio 3T

Create connection with

Name – Local Mongodb Connection

Address – localhost

Port – 27017

Double click Connection -> Open shell

db.version() => check connection with mongodb database

- It is like Javascript

Connecting to Mongodb

Task-Manager Project
npm i mongodb

Mongodb Driver for Node.js

Ctrl + C to close connection

With mongodb no need to create database

Example

Inserting Document

After Insertion complete

Open robo 3t

Right click **Local MongoDB Connection > Refresh**

- Check data view documents

- **_id** we did not create which is unique id created by mongodb

Example

Insert Bulk (Many)

Ctrl + R to refresh **robo 3T**

Challenge

Insert 3 tasks into new tasks collections

1. Use insertMany to insert 3 documents

- description(String), completed (Boolean)

2. setup the callback to handle error to print ops

3. Run script

4. Refresh the database in robo 3t and view data in tasks collection

Example

The Object ID

- In sql we have id with auto increment behavior
- In mongo _id is generated with GU Id (Global Unique Id)
- Search Mongodb Object Id on Google
- we can use id shown in code to insert collection

```
const {MongoClient, ObjectID} =
require('mongodb')
const id=new ObjectID()
console.log(id)
console.log(id.getTimeS
tamp)
```

- In robo 3t we can see ObjectID(5c….) it is just visualization to easy human readable

```
console.log(id.id.length)    => 12
console.log(id.toHexString().length) => 24
```

- So ObjectID use function internally do process to convert into hexa code

Example

Read Documents from Mongo

If data match with multiple collection then it always return first

Example

find()

find() returns cursor which is pointer

```
db.collection('users').find ( { age : 28 } )
        .toArray( (error, users) => {
                console.log(users)
})
```

- Count data

```
db.collection('users').find ( { age : 28 } )
        .count( (error, count) => {
                console.log(count)
})
```

Challenge

1. Use findOne() to fetch the last task by its id (print document in console)
2. Use find() to fetch all tasks that are not completed (print docs to console)
3. Test your work

Tasks

description : String

completed : boolean

ES6 : Promises

Make easy to work with asynchronous call

If we write

reject('Things went wrong')

response(['7,4,1'])

No App will crash only reject call then no other code execute

Use Promise with MongoDB

Mongo has in built support for Promises

Example

Updating Documents

We can also complete without **updatePromise** variable

Search mongodb update operator for $set

Example

Challenge : updateMany

1. Check the documentation for updateMany
2. setup the call with the query and the updates
3. Use promise methods to setup the success/error handler
4. Test your work

Deleting Documents

Example

Setting up Mongoose

-So far we have done basic operation like create, read, update, delete but how we can decide which field are required compulsory and which are optional and what type of data required

- We can also validate with data type like string, Boolean etc

- Tasks are added by valid user

- We are going add authentication using mongoose

- user1 created some task then user2 not allowed to access those data

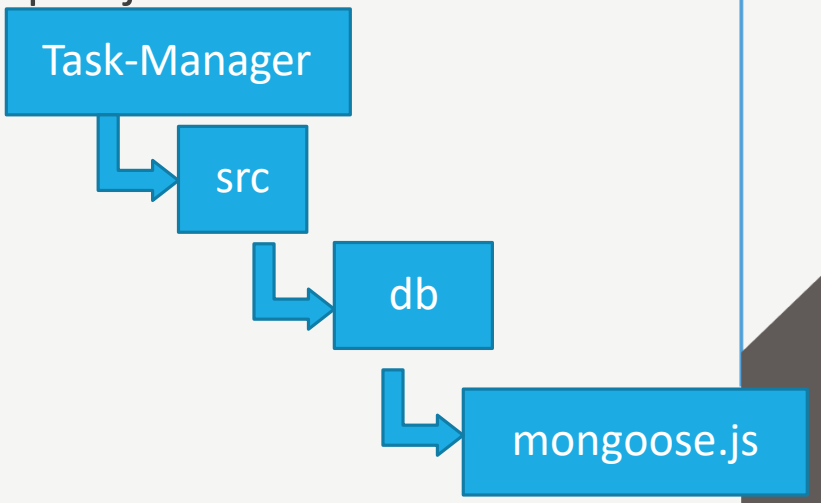
- Download from Mongoose.com

Model

- Like real world model which is going to be stored in database
- The methods we use so far in CRUD operation are very low level APIs
- ODM – Object Document Mapping tools in mongoose allows to map objects into Document

```
npm i mongoose
```

- Example



Continue...

- if we set **age:'abc'** then it will throw an error

- Challenge

1. Define the model with description and completed fields
2. Create a new instance of the model
3. Save the model to the database
4. Test your work.

Example

Data Validation & Sanitization

- Validation : User age>18
- Alter data before saving – Sanitization
- Example : Remove space around username
- mongoose docs check **Search Validation**

Example

Custom Validator (age > 0)

For Email, Postal code, Credit Card, Pancard we have to use well tested library

npm i validator

Validator Example

Validation Example 2

Schema Types

Check SchemaTypes on Mongoose Docs

Here we can see different validation options like trim, lowercase, default etc.

```
const User=mongoose.model('User', {
        name:{ type : String, required : true, trim : true },
        age : { type : Number, default : 0 },
        email : { type: String, trim : true, lowercase : true }
})
// Test with
const me = new User({ name : '    Ankit     ',
                      email : '   ankit@tops-int.com'})
me.save()….
```

Challenge

Add a password field to User

1. Setup the field as required string
2. Ensure the length is greater than 6
3. Trim the password
4. Ensure that password doesn’t contain ‘password’ (use javascript string includes method)
5. Test your code

Challenge

Task Validation and Sanitization

1. Trim the description and make it required
2. Make completed optional and default it is false
3. Test your work and without error

Module 5

Structuring REST AP

Structuring REST API

The Task Resource

Operation	Method	Path
CREATE	POST	/tasks
READ	GET	/tasks /tasks/:id
UPDATE	PATCH	/tasks/:id
DELETE	DELETE	/tasks/:id

Request - Response

Request

POST /tasks HTTP/1.1
Accept : application
Connection : keep-alive
Authorization : Bearer eyJhGc…
{ description : 'order new drill bits' }

Response

HTTP/1.1 201 Created
Date : Sun, 28 Jul 2020
Server : Express
Content-Type : 'application/json'
{"_id":"5c13…", "description": "Order…", "completed' : false}

INSTALL POSTMAN

Resource Creation Endpoints

- npm i nodemon –save –dev
- npm i express
- delete mongodb.js

Example 1

Example 2

Run & Test

npm run dev

package.json

```
“script” : {
        “start” : “node src/index.js”,
        “dev” : “nodemon src/index.js”
}
```

Test with post man

Postman>body>raw>json/application

```
{
“name”:”Ankit”,
“email”:”ankit@gmail.com”,
“password”:”Pass123”
}
```

To convert json to object automatically

```
app.use(express.json())
```

HTTP status

- Check httpstatuses.com

- Suppose if we provide password ‘ABC’ then it throws error but status is 200 which is invalid

Index.js

```
.catch((error)=>{
        res.status = 400
        res.send(error)
})
```

Or

```
res.status(400).send(error)
```

Resource Creation Part II

1. Create separate file for task model (load it into index.js)
2. Create the task creation endpoint (handle success and error)
3. Test the endpoint from postman with good and bad data

```
res.status(201).send(user)
```

Represent User Created which is more suitable for API behaviour

Resource Reading Endpoints Part I

Check mongoose documentation for query

Example

Setup end points for Tasks

1. Create an endpoint for fetching all tasks
2. Create an endpoint for fetching a task by its id
3. Setup new requests in postman and test your work

Promise Chaining

Example

Challenge

1. Create promise-chaining-2.js
2. Load in mongoose and task model
3. Remove a given task by id
4. Get and print the total number of incomplete tasks
5. Test your work

Use **findByIdAndDelete()** method

Example

Async / Await

Make more easy Asynchronous promise request

Playground > async-await.js

```
const doWork=()=>{        }

console.log(doWork())
```

If we don't return anything then by default we get **undefined**

- **async** allows us to create async function in that function we can use **await** feature

```
const doWork = async ()=>{        }

console.log(doWork())
```

- Now it will not return **undefined** instead it will return **Promise** Object

- async function always returns promise, that promise perform the value, we as developer wants to return

```
const doWork = async ()=>{
                    return 'Tops Technology'
                }
console.log(doWork())
```

O/P : Promise { 'Ankit' }

```
doWork().then((result)=>{
    console.log(result)
}).catch((e)=>{
    console.log(e)
})
```

- Await operator can be used only inside async function

-When we use async/await we are not changing internal structure of Promise instead we are changing the way we work with it

- Mongoose support promises so as a consumer we can use async/awat to replace promises

-Example

- async / await doesn’t make things faster it makes things easier

- Another problem with Promise chaining is difficult manage all value in single scope

- means we can not access sum , sum1, sum2 all at one place

- what if one of Promise reject in async/await

```
Const add = (a, b)=>{
        return new Promise((resolve, reject)=>{
                setTimeout(()=>{
                        if(a<0 || b<0){ return reject(‘Numbers must be positive’)}
                        resolve(a+b)
                }, 2000)
        })    }
```

Use Async/Await with Task-Manager

Example Patch & Delete & Post & etc

Challenge

1. Create deleteTaskAndCount as an async function
    - Accept id of task to remove
2. Use await to delete task and count up incomplete tasks
3. Return the count
4. Call the function and attach then/catch to log results
5. Test your work

Integrating Async/Await

Index.js

- **Similarly change each use API with Async/await**
- **and also for each task api**

```
App.post('/users', async (req, res)=>{
        const user=new User(req.body)
        try{
                await                user.save()
                res.status(201).send(user)
        }catch(e){
                res.status(400).send(e)
        }
})
```

Challenge

Update Task By Id

1. Setup the route handler
2. Send error if unknown updates
3. Attempt to update the task
    - Handle task not found
    - Handle validation errors
    - Handle Success
4. Test your work

Separate route files

- Move all user routers to routers/user.js and also fix all required methods

- rename app with router

Challenge

1. Create new file that creates/exports new routers
2. Move all the task routes over there
3. Load in and use that router with the express app
4. Test your work

Example

Module 6

API Authentication & Security

API Authentication & Security

Securely Storing Password :

npm bcrypt.js

- Hashing is one way algorithm

- Encryption algorithm we can also turn password back to original algorithm

Example

Comparing password when login

```
const isMatch = await bcrypt.compare('Red12345', hashedPassword)
console.log(isMatch)
```

Mongoose Middleware

- Before save user encrypt password
- Middleware specified on the schema level
- we are going to use document Middleware

- Example

Update operation code change

Challenge

1. Find the task

2. Alter the task parameter

3. Save the task

4.Test your work by updating a task from postman

Routers/user.js

```
router.patch('/users/:id', async (req, res)=>{
        …
        try{ const user= await User.findById(req.params.id)
        updates.forEach((update)=> {
                user[update]=req.body[update]
        })
        await user.save()
        }catch(error){ … }
})
```

Logging users

- Before test above code we have to confirm with email must be unique for that set **unique:true** with userSchema

- And drop database to make work it on

- Now test it in postman try to save 2 users with same email

- Then test login url

Example

JSON Web Tokens

- Authenticated User can only delete task or other task related operations

- After login user get Token and that token will be helpful for that user to perform other task like create new entry, delete, …

npm jsonwebtoken

- Json token made of 3 parts

Seprated by dot(.)

1st part – base64 encoded string

2nd part – base64 unique Id we provide

Copy it and check on base64decode.org

Example

Verify Token and set expiry

Verify Token ->

```
const data = jwt.verify (token, 'thisismytoken')
console.log(data)
```

We can also set expiry

Test with {0 seconds}

```
const token = jwt.sign({ _id : 'abc123' }, 'thisismytoken',
        {expiresIn : '7 days' })
```

Generating Authentication Tokens

- here user.generateAuthToken() is instance method not static method like findByCredential()

- Test the code with json

- Now we have generated token but it has not been saved so we can not compare for next request, so we have to store the token once it is generated

Example

Store token

Test code with login url

The same operation we need to perform when we create new user

Challenge

1.When create new user generate token and return with response

2.Also store token in database

Test the code with create user

Models/user.js

```
Const userSchema = mongoose.Schema({
        name : { … }
        email : { … }
        tokens : [{
                token : { type:String, required : true }
                }]
})
userSchema.methods.generatedToken = async function(){
        const user=this
        const token = …
        user.tokens=user.tokens.concat({token})
        await user.save()
        return user
}
```

Express Middleware

- Now we have to use token return after login or create user to authenticate user for other request

Every request require auth except signin and create user (signup)

Without Middleware

new request -> run route handler

With Middleware

new request -> do something -> run router handler

Test code it will not Complete the

Request without next()

Example

Index.js

```
app.use((req, res, next)=>{
        console.log(req.method, req.path)
        next()      // required to complete request
})
```

Accepting Authentication Token

Test the code Get Users

Pass headers with

Authoriization=“Bearer 5cToken“

- we can also validate user to read only his data instead of all users information

req.user=user

next()

Logging out

Test the code

-To logout from all devices

Remove all tokens and save user

- Setup POST users/logoutAll

req.user.tokens=[]

await req.user.save()

Hiding private Data

Instead of returning use return only public profile

res.send({ user: user.getPublicProfile() })

- we can also use **toJson** instead of **getPublicProfile** name of the function

- It will behave same as **toString**() in Java

-when we return an object or print object this method invokes automatically and we can return whatever we want to.

```
userSchema.methods.getPublicProfile=function(){
        const user = this
        const userObject=user.toObject()
        delete userObject.password
        delete userObject.tokens
        return userObject
}
```

```
userSchema.methods.toJson=function(){
        .....
}
```

Email Sending :

Install nodemailer module : npm i nodemailer

- Check out EmailEngine – a self-hosted email gateway that allows making REST requests against IMAP and SMTP servers. EmailEngine also sends webhooks whenever something changes on the registered accounts.
- Using the email accounts registered with EmailEngine, you can receive and send emails. EmailEngine supports OAuth2, delayed sends, opens and clicks tracking, bounce detection, etc. All on top of regular email accounts without an external MTA service.

- Example :

Chat app :

Install socket io module : npm i socket.io

- Socket.IO is a library that enables low-latency, bidirectional and event-based communication between a client and a server.
- It is built on top of the WebSocket protocol and provides additional guarantees like fallback to HTTP long-polling or automatic reconnection.
- Example :

File Uploading:

Install multer module : npm i multer

- Multer is a node.js middleware for handling multipart/form-data, which is primarily used for uploading files. It is written on top of busboy for maximum efficiency.
- NOTE: Multer will not process any form which is not multipart (multipart/form-data).
- Example :

Payment:

payment with razorpay : npm i razorpay

- Example :