NodeJS

^ : caret or semantic versioning or SemVer

^4.13.6 : major-minor-patch

#If bug iis found and removed, now It becomes

^4.13.7

#^4.14.0 minor changes without changing the existing code

NOTE : ^ tells any version is fine until newer major version comes, ^4.13.5 is fine or 4.x

# ~4.13.5 or 4.13.x : any version until major is 4 and minor is 13, i.e supports any patches

#4.13.6 is for exact versioning

Npm list //to see all installed packages with version

Npm list –depth=0 //dependency of only our application

Below is request processing pipeline

request response

json() route()

note:

app.use(express.json())//using expresses middleware function

app.use(function(req,res,next){

console.log(‘next is the reference to the next middleware function in the pipeline’);

next();//if we don’t write this then the req and res cycle //is hanging and wont terminate

});

Note: if we create the function in different file logger.js and then use in other

Logger.js

Function log(req,res,next){

console.log(‘next is the reference to the next middleware function in the pipeline’);

next();//if we don’t write this then the req and res cycle //is hanging and wont terminate

}

Module.exports=log//exports a single file

And where we are using it needs to be required to a variable and that variable will be used as a function

As below:

Var loggerFunction= require(‘./logger’)

App.use(loggerFunction)

ASYNC NODE\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1 )create a folder and npm init –yes to create a package.json file

Create a index.json

console.log('Before');

setTimeout(()=>{

console.log('Reading user')

},2000);

console.log('After');

note: output is

Before

After

Reading user

Note:setTimeout is asyncronous non blocking function, also asynchronous does not mean concurrent or multithreaded.

console.log('Before');

user= getUser(1);

console.log(user)

console.log('After');

function getUser(id){

setTimeout(()=>{

console.log('Reading user');

return { id: id,gitHubUser:'Farman'}

},2000);

}

o/p:

Before

undefined

After

Reading user

//to handle the send during any call, we have 3 methods

// 1)Callbacks

// 2)Async/Await

// 3)Promise

1)Callbacks

console.log('Before');

getUser(1,function(user){

console.log('User',user)

});

console.log('After');

function getUser(id,callback) {

setTimeout(() => {

console.log('Reading user');

callback({ id: id, gitHubUser: 'Farman' });

}, 2000);

}

O/P:Before

After

Reading user

USer { id: 1, gitHubUser: 'Farman' }

console.log('Before');

getUser(1,function(user){

console.log('User',user)

getRepositories(user.gitHubUser,function(repos){

console.log(repos)

})

});

console.log('After');

function getUser(id,callback) {

setTimeout(() => {

console.log('Reading user');

callback({ id: id, gitHubUser: 'Farman' });

}, 2000);

}

function getRepositories(username,callback){

setTimeout(()=>{

callback(['repo1','repo2','repo3'])

},2000)

}

Before

After

Reading user

User { id: 1, gitHubUser: 'Farman' }

[ 'repo1', 'repo2', 'repo3' ]

But the nested structure creates a CALLBACK HELL problem or CHRISTMAS TREE PROBLEM

To resolve this, in first case , we remove the unnamed functions with a function and don’t call it but pass a reference to the function

console.log('Before');

getUser(1,displayUser);

console.log('After');

function getUser(id,callback) {

setTimeout(() => {

console.log('Reading user');

callback({ id: id, gitHubUser: 'Farman' });

}, 2000);

}

function displayUser(user){

console.log('User',user)}

2) PROMISE

console.log('Before');

// const userPromise= getUser(1);

// userPromise.then(user=>{ console.log('user : ',user)})

//or

getUser(1)

.then(user=>{ console.log('user : ',user)})

console.log('After');

function getUser(id) {

return new Promise((resolve, reject)=>{

setTimeout(() => {

console.log('Reading user');

resolve({ id: id, gitHubUser: 'Farman' });

}, 2000);

});

}

Before

After

Reading user

user : { id: 1, gitHubUser: 'Farman' }

Note: also chaining of the .then can be done to reolve the CALLBACK HELL using promise

//create an already resolved promise

const p=Promise.resolve({id:'123'})

p.then(result=>{ console.log(result)})

//create an already rejected promise

const p2=Promise.reject(new Error('reason of rejection'))

p2.catch(error=>{ console.log(error)})

const p1 =new Promise((resolve)=>{

setTimeout(()=>{

console.log('Async Operation 1...')

resolve(1)

},2000);

})

const p2 =new Promise((resolve)=>{

setTimeout(()=>{

console.log('Async Operation 2...')

resolve(2)

},2000);

})

Promise.all([p1,p2])

.then(result=>{console.log(result)})

//note: there is no multi thread

// its running a single thread almost at a same time

Async Operation 1...

Async Operation 2...

[ 1, 2 ]

const p1 =new Promise((resolve)=>{

setTimeout(()=>{

console.log('Async Operation 1...')

resolve(1)

},2000);

})

const p2 =new Promise((resolve)=>{

setTimeout(()=>{

console.log('Async Operation 2...')

resolve(2)

},2000);

})

Promise.race([p1,p2])

.then(result=>{console.log(result)})

//note: there is no multi thread

// its running a single thread almost at a same time

Async Operation 1...

1

Async Operation 2...

3) ASYNC/AWAIT on Promise

console.log('Before');

// const userPromise= getUser(1);

// userPromise.then(user=>{ console.log('user : ',user)})

// .catch(error=>console.log('error :',error))

//or

async function getUserAsync()

{

try{

const user=await getUser(1);

console.log(user)

}

catch(error){

console.log('error',error)

}

}

//created a async function so that we can use await

getUserAsync()

console.log('After');

function getUser(id) {

return new Promise((resolve, reject)=>{

setTimeout(() => {

console.log('Reading user');

resolve({ id: id, gitHubUser: 'Farman' });

}, 2000);

});

}

Note: A function call can await only if it resides in a function which is async

MONGODB

const mongoose = require('mongoose');

//this connect returns a promise

mongoose.connect('mongodb://localhost/vidly')

.then(() => console.log('Connected to MongoDB..'))

.catch(err => console.error('Could not connect to MongoDB...'));

SCHEMA (in mongoose)

const anySchema= new mongoose.Schema({

});

const anySchema= new mongoose.Schema({

name: String,

author: String,

tags: [String],

date: { type: Date,default: Date.now},

isPublished: Boolean

});

Types of Schema:

String

Number

Date

Buffer

Boolean

ObjectID

Array

const AnyModelClass=mongoose.model('anymodel',anySchema)

//we want a collection called anymodels so here we give a singular name anymodel

//this function returns a class which is in this case AnyModel

Note: Schema is a class and model is a function

Now creating the object of the class

const anyModelObject= new AnyModelClass({

name: 'Node',

author: 'Farman Abbasi',

tags:['node','MEAN'],

isPublished:true

});

How to save this document to the database og mongodb

const result= await anyModelObject.save()

console.log(result)

Complete snippet:

const mongoose= require('mongoose');

mongoose.connect('mongodb://localhost/test')

.then(()=>console.log('connected to the db'))

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

const anySchema= new mongoose.Schema({

name: String,

author: String,

tags: [String],

date: { type: Date,default: Date.now},

isPublished: Boolean

});

const AnyModelClass=mongoose.model('anymodel',anySchema)

//we want a collection called anymodels so here we give a singular name anymodel

//this function returns a class which is in this case AnyModel

async function createCourse(){

const anyModelObject= new AnyModelClass({

name: 'Node',

author: 'Farman Abbasi',

tags:['node','MEAN'],

isPublished:true

})

const result= await anyModelObject.save()

console.log(result)

}

createCourse();

//querying

async function getCourses(){

const any=await AnyModelClass

.find({name: 'Angular'})

//starts with

//.find({name: /Angular$/})

//ends with

//.find({name: /^Angular/})

// append I to make case insensitive

//.find({name: /^Angular/i})

//contains farm

//.find({name: /.\*farm.\*/})

console.log(any)

}

getCourses()

update

async function updateCourse(id){

//query first

const course= await AnyModelClass.findById(id);

if(!course) return;

course.isPublished=false;

course.author='another author';

const result=await course.save();

console.log(result)

//or

// course.set({

// isPublished:false,

// author:'another author'

// });

//update first

}

updateCourse("5e0de55def231676188177a0")

async function updateCourse(id){

//update first

//when dont know what to update

const course= await AnyModelClass

.update({\_id: id},

{ $set :

{ author: 'Farman'}

});

console.log(course)

}

updateCourse("5e0de55def231676188177a0")

SCHEMA VALIDATION

As we create the schema without any required field so now, how to create schema using required field,

But note these validations are meaningful to mongoose only not to mongoDB it doesn’t care while saving to db.

So to check at the point where data is going inside db we can use JOI.

const anySchema= new mongoose.Schema({

name: { type: String,required: true},

author: { type: String,required: true},

tags: [String],

date: { type: Date,default: Date.now},

isPublished: Boolean

});

//these all validators are present in mongoose

//type: String,

//required: true,

// minlength:5,

// maxlength:50,

// enum:['a','b','c'],

// match:/pattern/

Also the object created from schema,class has a validate() function which returns fail if any valdation is failed.

async function createCourse(){

const anyModelObject= new AnyModelClass({

//name: 'Angular 7', //as name is not provided so it will fail

author: 'Farman Abbasi',

tags:['Angular','MEAN','A7'],

isPublished:true

});

try{

await anyModelObject.validate();

//or

// const result= await anyModelObject.save();

// console.log(result)

}

catch(e){

console.log(e.message);

}

}

createCourse();

Note:

In mongodb , when creating document in db it will be done on the OBJECT of the class

And when fetching or updating it will be done on the CLASS.

exports.Customer = Customer; //or module.exports.Customer = Customer

//for exporting single thing

//module.exports=router // exports everything

RELATIONSHIPS IN NODE

//using references i.e normalization -> gives CONSISTENCY

//but to perfrom query in course have to perform an extra query in author

let author={

name:'Farman'

}

let course={

author:'id'

}

//Using embedded Documents i.e Denormalization -> gives PERFORMANCE but have to update in multiple cases

//causing some inconsistency

let course={

author:{

name:'Farman'

}

}

//Third approach is hybrid

let author={

name:'Mosh'

//50 other properties

}

let course={

author:{

id:'ref',

name:'Farman'

}

}

HOW TO REFERENCE: