* How to connect mongoDB with nodeJs ?

1st , create a nodejs server and then also connect mongoDB to mongodb server and then install mongoose in the javascript file== npm i mongoose

2nd ,then require the mongoose in the javascript file and make a connection with mongoDB file

mongoose.connect("mongodb://localhost:27017/database\_name ")

One system of connecting nodejs With mongoDB with promise:

let express=require("express")

let app= express()

let port= 3000

let mongoose =require("mongoose")

mongoose.set("strictQuery",false)

mongoose.connect("mongodb://127.0.0.1:27017/products")

.then(()=>{

    console.log("mongoDB is connected")

})

.catch((error)=>{

    console.log("mongoDB is not connected")

    console.log(error)

    process.exit(1)

})

app.listen(port,()=>{

    console.log(`Your server is running at http://localhost:${port}`)

})

**mongoose.set("strictQuery",false)** [this is optional]

Another system of connecting mongoDB with try-catch:

// another system of connecting

let connectDB= async () =>{

    try {

       await mongoose.connect("mongodb://127.0.0.1:27017/products")

        console.log("mongoDB is connected")

    } catch (err) {

        console.log("mongoDB is not connected")

    }

}

* What is schema and model and how we can create schema and model of a collection?

A schema is a JSON object that defines the the structure and contents of your data

// creating schema

let product\_schema= new mongoose.Schema({

    title:String,

    price:Number,

    description:String,

    created\_at:{

        type:Date,

        default:Date.now()

    }

})

When we want to use schema then we must use model

// creating product Model

// let model=mongoose.model("collection\_name", schema)

let product\_model=mongoose.model("products",product\_schema)

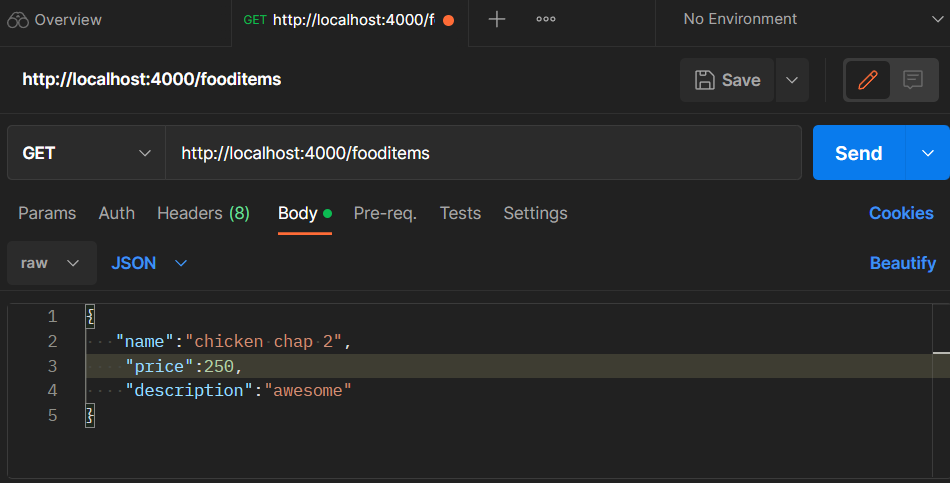
* **CRUD(create read update delete) operation Using post man :**

We must add some document to use post man in our projects



Then we need to go to postman> body> select json in row

then write the json file



* **Create operation:**

// Create operation

app.post("/fooditems",(req,res)=>{

    try {

        let foods=food\_model({

            name:req.body.name,

            price:req.body.price,

            description:req.body.description

        })

       foods.save()

       res.status(200).json(foods)

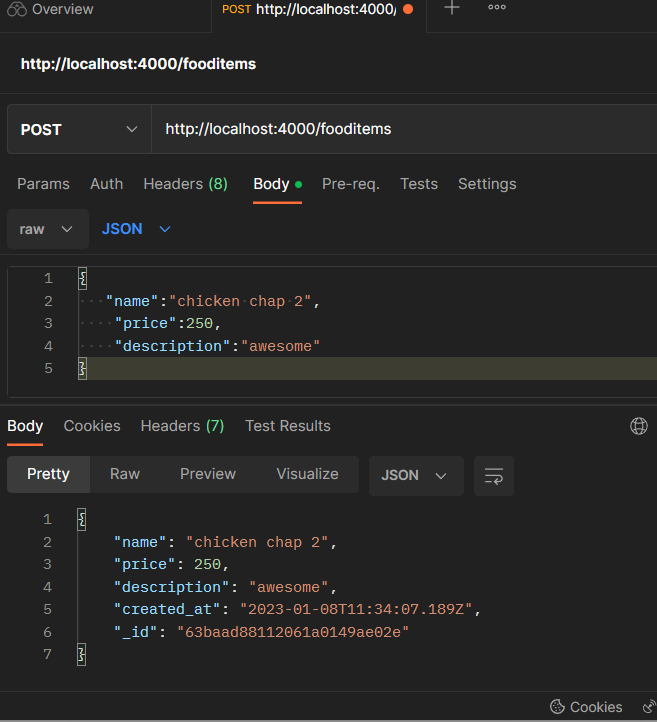
    } catch (error) {

        res.status(404).json({massage:error})

    }

})

When we post it in postman then we can see this



* **Read or getting Operation:**

Getting all items :

// Read operation or find all food items

app.get("/fooditems",async(req,res)=>{

    try {

       let fooods=await food\_model.find({price:{$gt:400}})

       if(fooods)

       {

        res.status(202).json(fooods)

       }

       else

       {

        res.status(404).json({massage:"product not found"})

       }

    } catch (error) {

        res.status(500).json({massage:"someThing is wrong"})

    }

})

Getting a specific product using ID and decide what things we want to see and what not :

// find a specific food items

Decide what items we don’t want to watch

app.get("/fooditems/:id",async(req,res)=>{

  try {

Getting the item according to id

    let id=req.params.id

    let food=await food\_model.findOne({\_id:id},{\_id:0,description:0,created\_at:0})

    if (food) {

        res.status(202).json(food)

    } else {

        res.status(404).json({massage:"food not found"})

    }

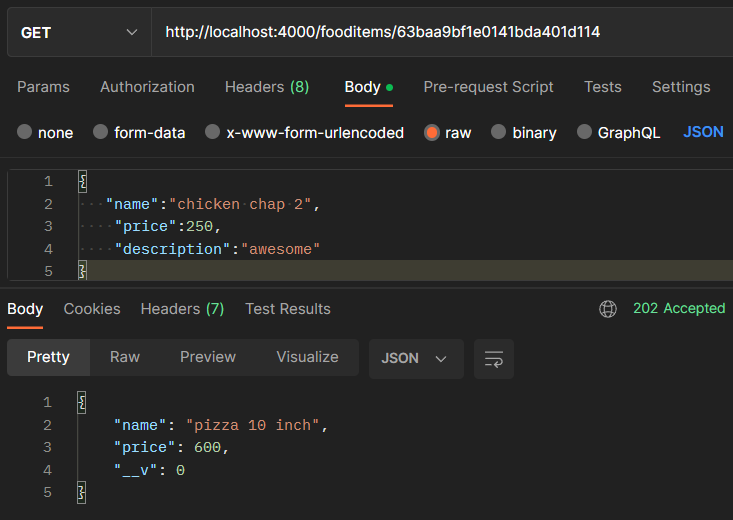
  } catch (error) {

    res.status(500).json({massage:"something is wrong"})

  }

})

When we go to postman and get a specific product using id



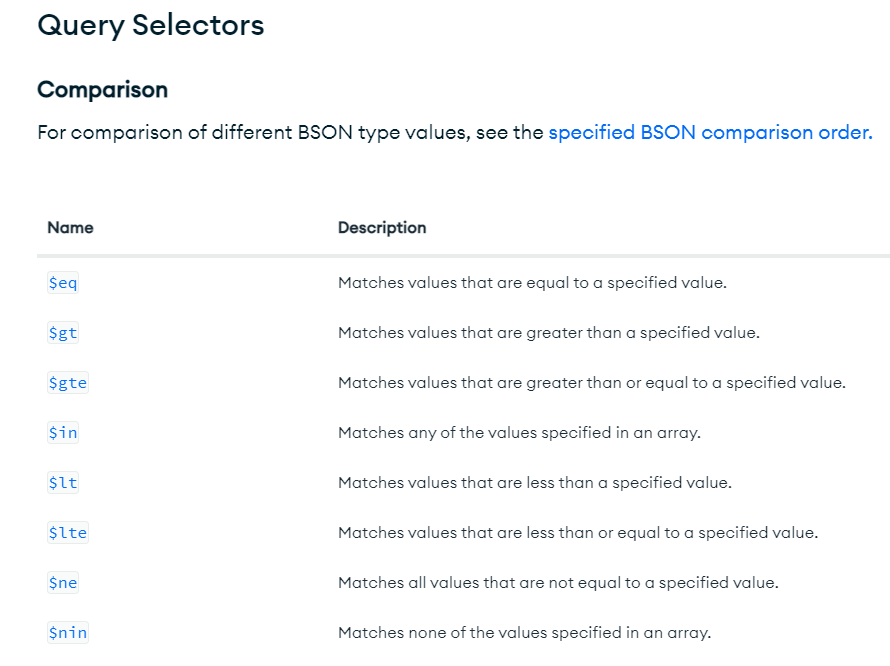
Product item according to ID

ID

* **Query Operation:**

If we want the items according to price **greater than** a specific value or **smaller than** a specific value or **equal to** a specific value then we use query operators

* **Comparison Operators:**



$eq 🡺 Equal to

$lt 🡺 less than

$lte 🡺 less than and equal to

$gt 🡺 greater than

$gte 🡺 greater than and equal to

$ne 🡺 not equal to

$nin 🡺 none of the values specified in an array

$in 🡺 matches any other value specified in an array

**How to use these operators** :

We use this operators in find section

**{price:{$gt:400}}**

**Or,**

**{item\_name : {$comparison\_operator : Value}}**

and put this thing into find section

// getting items using query operators

app.get("/fooditems",async(req,res)=>{

Here how we can use query operation

    try {

       let fooods=await food\_model.find({price:{$gt:400}})

       if(fooods)

       {

        res.status(202).json(fooods)

       }

       else

       {

        res.status(404).json({massage:"product not found"})

       }

    } catch (error) {

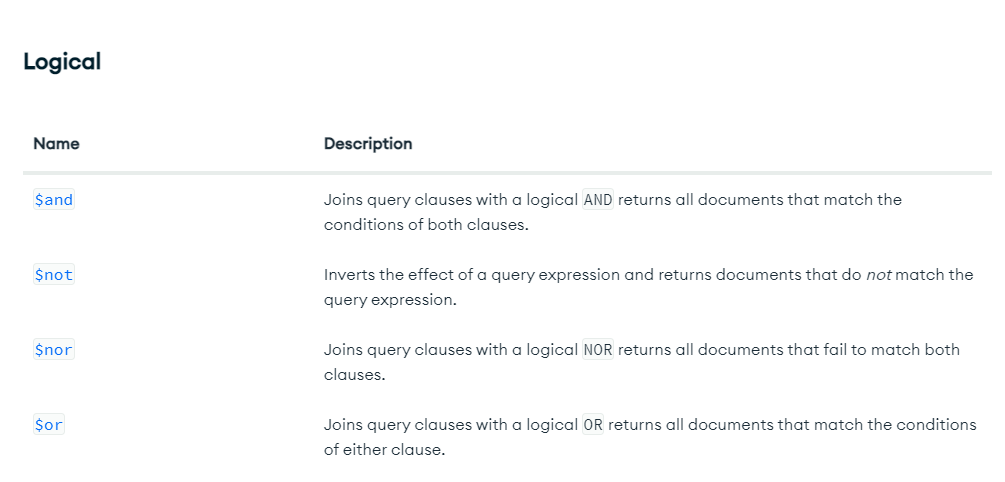
        res.status(500).json({massage:"someThing is wrong"})

    }

})

* **Logical Operators:**

When we want to combine two or more comparison operators then we use logical operators



$and 🡺 যখন দুইটাই সত্য হবে

$or 🡺 যখন দুইটার যেকোনো একটি সত্য হবে

$not 🡺 যখন দুইটার কোনোটাই সত্য হবেনা

$nor 🡺 opposite of and

How to use Logical operators:

**({$and:[{price:{$gt:200}},{rating:{$gt:4.5}}]})**

Or)

**({$logical\_operator:[{item\_name:{$comparison\_operator :value }},{item\_name:{$comparison\_operator :value }})**

// Read operation or find all food items

This is how we add logial operators

app.get("/fooditems",async(req,res)=>{

    try {

    let fooods=await food\_model.find({$and:[{price:{$gt:200}},{rating:{$gt:4.5}}]})

       if(fooods)

       {

        res.status(202).json(fooods)

       }

       else

       {

        res.status(404).json({massage:"product not found"})

       }

    } catch (error) {

        res.status(500).json({massage:"someThing is wrong"})

    }

})

* **Counting Sorting and selecting Data**

🡺 For the counting of the data that means how much data we have, we use countDocument() after the find() function

app.get("/motorbikes",async(req,res)=>{

    try {

        let bikes= await motorBikes\_model.find().countDocuments()

        if (bikes) {

            res.status(200).json(bikes)

        } else {

            res.status(404).json({massage:"bikes not found"})

        }

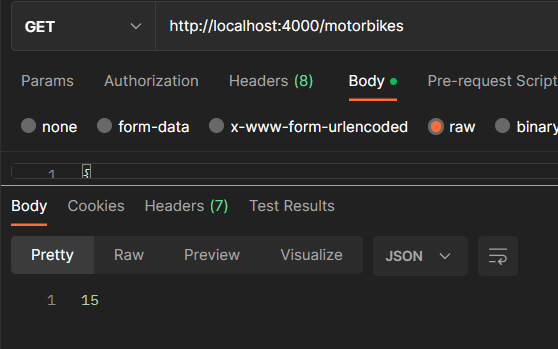
    } catch (error) {

        res.status(500).json({massage:"something is wrong"})

    }

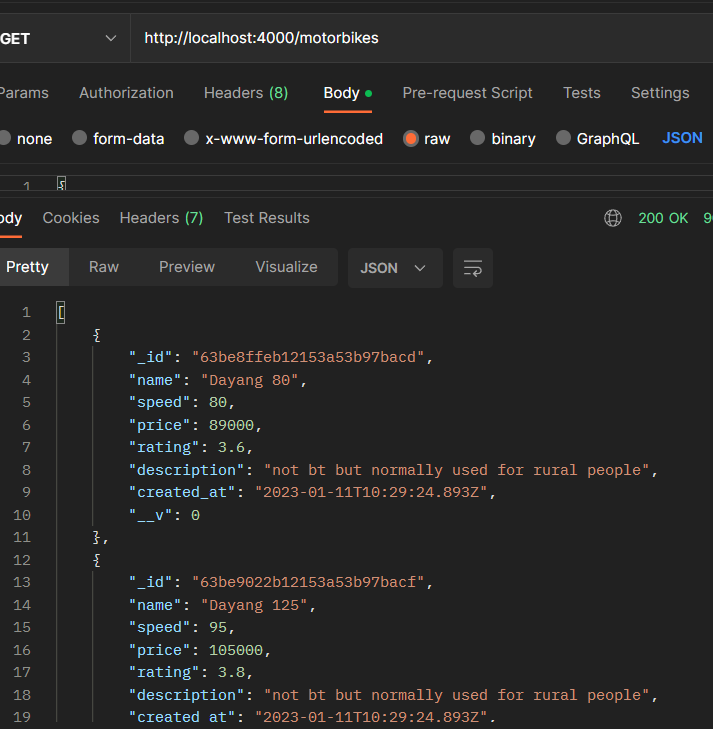
})

Here we can see the counter of the data in the postman, 15



🡺 For sorting , we use sort() function after the find() function

* If we use .find().sort({price:1})then the document are sorted in increasing order
* if we use .find().sort({price:-1}) then the document are sorted in decreasing order



The documents are sorted in accending order

🡺 if we want some specific information in our document then we normally use select() function.

Sorting and selecting for what information we need and what information we don’t

app.get("/motorbikes",async(req,res)=>{

    try {

        let bikes= await motorBikes\_model.find().sort({price:1}).select({name:1,\_id:0,price:1})

        if (bikes) {

            res.status(200).json(bikes)

        } else {

            res.status(404).json({massage:"bikes not found"})

        }

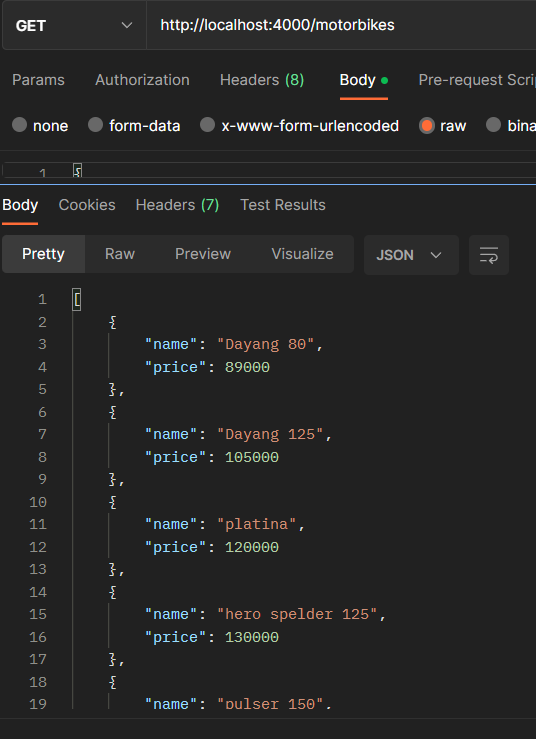
    } catch (error) {

        res.status(500).json({massage:"something is wrong"})

    }

})

In postman we can see a sorted data in specific items,



* **Delete Items:**

🡺 If we want to delete without showing its data then we use

// Delete items

Delete single item without showing its real data

app.delete("/motorbikes/:id", async(req,res)=>{

    try {

        let id=req.params.id

        let bike= await motorBikes\_model.deleteOne({\_id:id}) //if we want to delete single item without showing its real data

        if (bike) {

            res.status(202).json({bike})

        } else {

            res.status(404).json({massage:"bike not deleted"})

        }

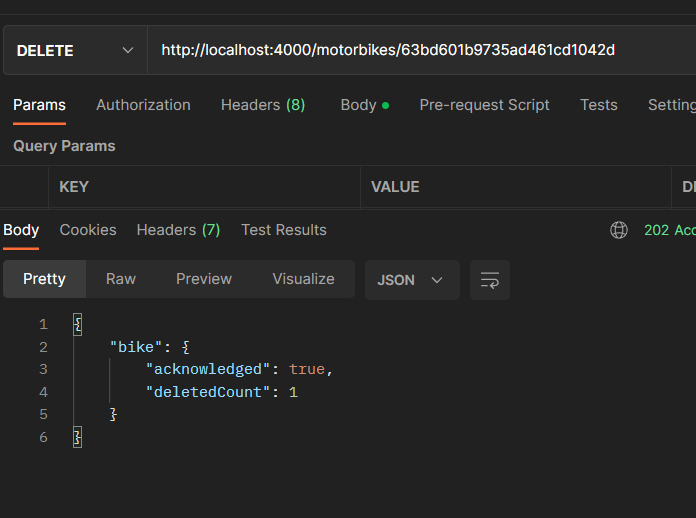
    } catch (error) {

        res.status(500).json({massage:"something is wrong"})

    }

})

When we delete items from postman then we only see acknowledged:true



🡺 Then if we want to delete items by seeing its real data then we use

let bike = await motorBikes\_model.findByIdAndDelete({\_id:id})

// Delete items

Deleting by seeing its real data

app.delete("/motorbikes/:id", async(req,res)=>{

    try {

        let id=req.params.id

        // let bike= await motorBikes\_model.deleteOne({\_id:id}) //if we want to delete single item without showing its real data

        let bike = await motorBikes\_model.findByIdAndDelete({\_id:id})   // deleting by seeing its real data

        if (bike) {

            res.status(202).json({bike})

        } else {

            res.status(404).json({massage:"bike not deleted"})

        }

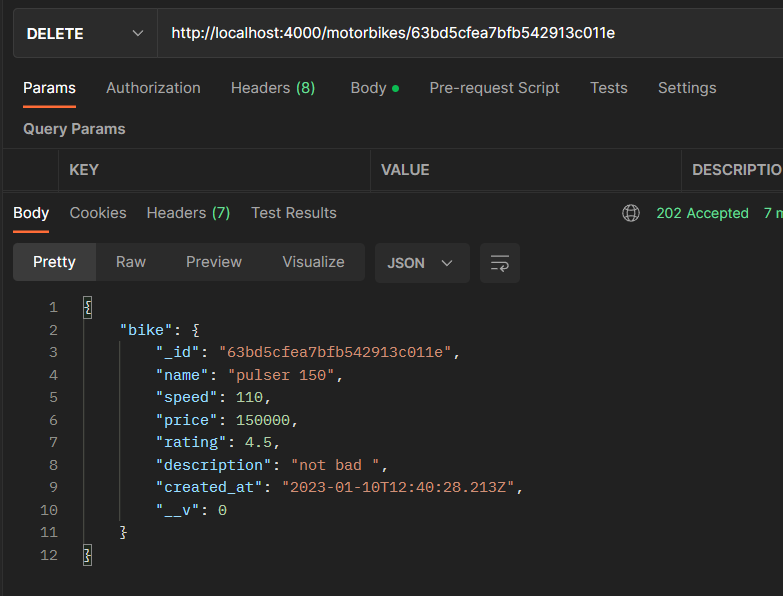
    } catch (error) {

        res.status(500).json({massage:"something is wrong"})

    }

})

When we see the result in postman



* **Update Items:**

🡺 If we want to update without showing its real data

Then we use

Name\_model.updateOne(

{

\_id:id

},

{

$set:{

Updated\_items

}  
},

{

new:true

}

)

🡺 if we want to update by showing its real data then,

Name\_model. findByIdAndUpdate (

{

\_id:id

},

{

$set:{

Updated\_items

}  
},

{

new:true

}

)

Vs code:

// update items

app.put("/motorbikes/:id",async(req,res)=>{

    try {

        let id=req.params.id

        let name=req.body.name

        let price=req.body.price

        let speed=req.body.speed

        let description=req.body.description

        let rating=req.body.rating

        let updatedBike=await motorBikes\_model.findByIdAndUpdate(

            {

                \_id:id

            },

            {

                $set:{

                   name:name,

                   price:price,

                   speed:speed,

                   description:description,

                   rating:rating

                }

            },

            {

                new:true

            }

        )

        if (updatedBike) {

            res.status(202).json(updatedBike)

        } else {

            res.status(404).json({massage:"bike not updated"})

        }

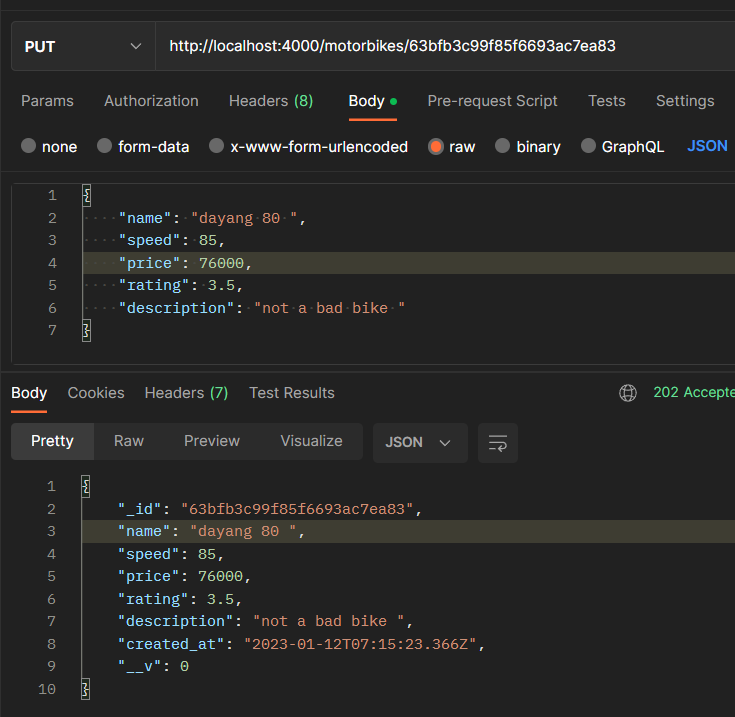
    } catch (error) {

        res.status(500).json({massage:"something is wrong"})

    }

})

If we see the postman then we see the information what we updated right now



* Schema validation:
* Build-In validator:

When we use schema for our project then we need validation for our project. The build-in validation is used in schema as build-in things like min,max, required

Website\_link: <https://mongoosejs.com/docs/validation.html#update-validators-only-run-on-updated-paths>

* Custom Schema Validator:

Custom validator used when building not working

// creating schema

Build-in Schema validation

let motorBikes\_schema = new mongoose.Schema(

    {

    name:{

        type:String,

        // below are schema validation

        required : [true, "name must be included"],

        minlength:[3, "min length is 3"], //minimum length is 3cherecter

        maxlength:[100, "max length is 15"], //maximum cherecter is 15 cherecter

        // uppercase:true,  // name strored in database in uppercase

      lowercase:true,  // name strored in database in lowercase

         trim:true,   //reduce the space in front and back

         //  enum:{values:["pulsar150, fz version 2"], message:`{VALUE} is not supported`},  //that means if give name that must be between the enum

 unique:true,  // the name must be unique

    },

    speed:{

Custom schema validation

        type:Number,

        required : [true, "speed mustbe mentioned"],

        validate:{

            validator:function(v){

                return v.length<=3

            },

            message:(props)=>`${props.value} is not a valid speed`

        }

    },

     price:{

        type:Number,

        min:[50000,"minimum cost for a bike is 50000taka"],

        max:[1000000,"maximun cost for a bike is 1000000taka"],

        required:true,

        validate:{

            validator:function(v){

                return v.length<=4

            },

            message:(props)=>`${props.value} is not a valid speed`

        }

    },

    rating:{

        type:Number,

        required:true

    },

    description:{

        type:String,

        required:true

    },

    created\_at:{

        type:Date,

        default:Date.now()

    }

})