**mongoDB**

[**https://docs.mongodb.com/manual/crud/**](https://docs.mongodb.com/manual/crud/)

MongoDB is a schema-less NoSQL document database. It means you can store JSON documents in it, and the structure of these documents can vary as it is not enforced like SQL databases. This is one of the advantages of using NoSQL as it speeds up application development and reduces the complexity of deployments.

* **Tables : Collection**
* **Records : Document**
* **Column : Field**

1. **To create/switch into a database, we need to use following syntax:**

use DATABASE\_NAME

1. **To display following database into your system, use following syntax:**

show dbs

**It will only display your DB when you have atleast one document into your collection. Collection will be created itself when you insert document into it.**

1. **To insert a single document into a collection, we use following syntax:**

db.COLLECTION\_NAME.insertOne({name: “ABC”, age: 20, active: true})

It will create one extra field as ID of that document by default.

1. **To check your current database in which you are, use following syntax:**

db

1. **To check number of collections in any database, use following syntax:**

show collections

1. **To read all documents of any collection, use following syntax:**

db.COLLECTION\_NAME.find(query, projection) **OR**

db.COLLECTION\_NAME.find(query, projection).pretty()

**query** will defined condition onto which you want to read your document and **projection** is used to either show or not any field of document.

1. **To insert more than one document into collection, use following syntax:**

db.COLLECTION\_NAME.insertMany([{name: “ABC”, age: 20, active: true}, { name: “XYZ”, age: 25, active: true }])

1. **To read any document based on some condition, use following syntax:**

db.COLLECTION\_NAME.find({name: “ABC”}).pretty()

1. **To read any document with age = 20 but only name field, use following syntax:**

db.COLLECTION\_NAME.find({age: 20}, {name:1}).pretty()

**name:1** means that this field is to be displayed only but,

**name:0** means that this field is excluded only and all others fields are displayed.

db.COLLECTION\_NAME.find({name: “ABC”}, {\_id: 0, name:1}).pretty()

1. **To get only first field with active:true value, use following syntax:**

db.COLLECTION\_NAME.find({active:true}).pretty().limit(1) OR

db.COLLECTION\_NAME.findOne({active:true})

1. **To get only 2nd field by skipping 1st with active:true value, use following syntax:**

db.COLLECTION\_NAME.find({active:true}).pretty().limit(1).skip(1)

1. **To update any document, use following syntax:**

db.COLLECTION\_NAME.updateOne(<filter>, <update>) OR

db.COLLECTION\_NAME.updateMany(<filter>, <update>): if you want to update more than one document based on any condition.

1. **To update name ABC to DEF, use following syntax:**

db.COLLECTION\_NAME.updateOne({name: “ABC”}, {$set: {name: “DEF”}})

1. **To delete any document from collection, use following syntax:**

db.COLLECTION\_NAME.deleteMany(deletion\_criteria)

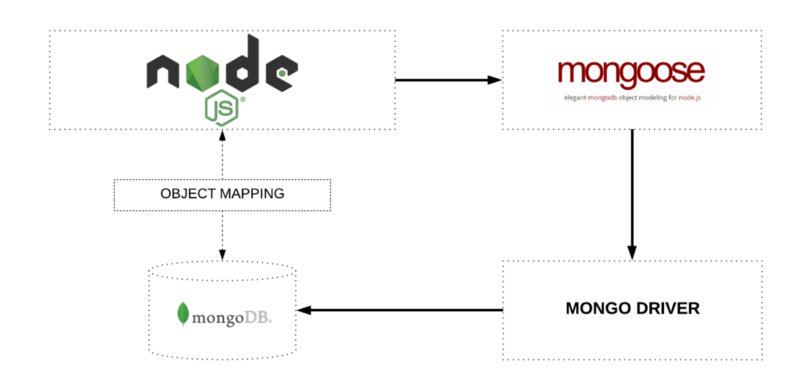
db.COLLECTION\_NAME.deleteMany({name: “ABC”})

db.COLLECTION\_NAME.deleteMany({}): This will delete all documents from collection.

**Mongoose**

It help us to establish connection between mongoDB and nodeJS.

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB.



MongoDB is a schema-less NoSQL document database. It means you can store JSON documents in it, and the structure of these documents can vary as it is not enforced like SQL databases. This is one of the advantages of using NoSQL as it speeds up application development and reduces the complexity of deployments.

**CONNECT NODEJS, EXPRESS TO MONGODB USING MONGOOSE**

Create a new folder named mongoose > src > app.js

**To add package.json file on your project:** D:\Learning\mongoose> npm init –y

**Install mongoose onto your project:** D:\Learning\mongooseLearning> npm i mongoose

**Establish Connection:**

const mongoose = require('mongoose'); **//require mongoose module**

**mongoose.connect**("mongodb://localhost:27017/**aatechnical**") **//connection establish**

.then(() => console.log("Connection Successfull..")) **//return promise**

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

aatechnical is a database name we have provided. If it is present we can work on it but if it is not present then it will create a new database for us. BUT it will not shown till one record is inserted into it which is basic requirement of mongoDB.

# MONGOOSE SCHEMA AND MODELS | CREATE COLLECTIONS USING MONGOOSE

# A mongoose schema defines the structure of the document, default values, validators, etc.

# Example: defining the schema

const playlistSchema = new **mongoose.Schema**({ **//defining schema for documents**

    name: { **//playlistSchema is an object**

        type: String,

        required: true

    },

    ctype: String,

    videos: Number,

    author: String,

    active: Boolean,

    date: {

        type: Date,

        default: Date.now

    }

});

# A mongoose model is a wrapper on the mongoose schema which provides an interface to the database for creating, querying, updating, deleting records, etc.

# Example: creating a collection

const **Playlist** = new **mongoose.model**("Playlist", playlistSchema); **//creating collections. Here, Playlist is a class. Collection name should be singular and start with capital letter**

# CREATE AND INSERT THE DOCUMENT USING EXPRESS IN MONGODB USING MONGOOSE

# Example: insert single document into collection

const reactPlaylist = new Playlist({ **//creating document or instance of class Playlist**

    name: 'React JS', **//inserting document**

    ctype: 'Front End',

    videos: 100,

    author: 'Aayush Agarwal',

    active: true

})

**reactPlaylist.save();** **//need to save the document. This save method will return a promise**

# NEW METHOD (async function):

const **createDocument** = **async** () => {

    try{

        const reactPlaylist = new Playlist({

            name: 'Node JS',

            ctype: 'Back End',

            videos: 10,

            author: 'Aayush Agarwal',

            active: true

        })

        const result = **await** reactPlaylist.save();

        console.log(result);

    }catch(err){

        console.log(err);

    }

}

**createDocument(); //calling function to insert data**

**INSERT MULTIPLE DOCUMENTS INTO COLLECTION**

**Example: insert multiple documents into collection**

const **Playlist** = new mongoose.model("Playlist", playlistSchema);

const createDocument = async () => {

    try{

        const jsPlaylist = new **Playlist**({

            name: 'JS',

            ctype: 'Front End',

            videos: 50,

            author: 'Aayush Agarwal',

            active: true

        })

        const nodePlaylist = new **Playlist**({

            name: 'Node JS',

            ctype: 'Back End',

            videos: 40,

            author: 'Aayush Agarwal',

            active: true

        })

        const mongoPlaylist = new **Playlist**({

            name: 'Mongo DB',

            ctype: 'Database',

            videos: 30,

            author: 'Aayush Agarwal',

            active: true

        })

        const expressPlaylist = new **Playlist**({

            name: 'Express JS',

            ctype: 'Back End',

            videos: 60,

            author: 'Aayush Agarwal',

            active: true

        })

        const result = await **Playlist**.**insertMany**([jsPlaylist, nodePlaylist, mongoPlaylist, expressPlaylist]);

        console.log(result);

    }catch(err){

        console.log(err);

    }

}

createDocument();

# HOW TO READ OR QUERYING THE DOCUMENTS USING MONGOOSE IN EXPRESS

# Example: Find name of those records whose course type is Front End

const getDocument = async () => {

    try{

        const result = await **Playlist**.find({ctype: "Front End"}, {name:1}).limit(1); **//Playlist is a name of collection**

        console.log(result);

    }catch(err){

        console.log(err);

    }

}

getDocument();

**NOTE**:

ctype: "Front End" **//condition to be specified**

name:1 **//which field is to be included**

limit(1)  **//top nth field to be retrieved**

# MONGODB COMPARISON QUERY OPERATORS USING MONGOOSE AND NODE(EXPRESS JS)

MongoDB uses various comparison query operators to compare the values of the documents. The following table contains the comparison query operators:

**OPERATORS DESCRIPTION**

**$eq** It is used to match the values of the fields that are equal to a specified value.

**$ne** It is used to match all values of the field that are not equal to a specified value.

**$gt** It is used to match the values of the fields that are greater than a specified value.

**$gte** It is used to match the values of the fields that are greater than equal to the specified value.

**$lt** It is used to match values of the fields that are less than a specified value

**$lte** It is used to match values of the fields that are less than equals to the specified value

**$in** It is used to match any of the values specified in an array.

**$nin** It is used to match none of the values specified in an array.

**Example: Find name of those records who have greater than 50 videos**

const getDocument = async () => {

    try{

        const result = await Playlist.find({videos: **{$gt: 50}**}, {name:1});

        console.log(result);

    }catch(err){

        console.log(err);

    }

}

getDocument();

**Example: Find name of those records whose course type is either Front End or Database (use IN)**

const getDocument = async () => {

    try{

        const result = await Playlist.find({ctype: **{$in: ['Front End', 'Database']}**}, {name:1});

        console.log(result);

    }catch(err){

        console.log(err);

    }

}

getDocument();

# MONGODB LOGICAL QUERY OPERATORS USING MONGOOSE AND NODE(EXPRESS JS)

MongoDB supports logical query operators. These operators are used for filtering the data and getting precise results based on the given conditions. The following table contains the comparison query operators:

**OPERATOR DESCRIPTION**

**$and** It is used to join query clauses with a logical AND and return all documents that match the given conditions of both clauses.

**$or** It is used to join query clauses with a logical OR and return all documents that match the given conditions of either clause.

**$not** It is used to invert the effect of the query expressions and return documents that do not match the query expression.

**$nor** It is used to join query clauses with a logical NOR and return all documents that fail to match both clauses.

**Example**: **Find name of those playlist whose either ctype is Front End or author is Aayush Agarwal**

const getDocument = async () => {

    try{

        const result = await Playlist.find(**{$or: [{ctype: 'Front End'}, {author: 'Aayush Agarwal'}]}**, {name:1});

        console.log(result);

    }catch(err){

        console.log(err);

    }

}

getDocument();

# MONGODB SORTING AND COUNT QUERY METHODS USING MONGOOSE AND NODE(EXPRESS JS)

**Example: Count number of records who have greater than 50 videos**

const getDocument = async () => {

    try{

        const result = await Playlist.find({videos: {$gt: 50}}).**countDocuments();**

        console.log(result);

    }catch(err){

        console.log(err);

    }

}

getDocument();

# Example: Sort our records based on name field

const getDocument = async () => {

    try{

        const result = await Playlist.find({videos: {$gt: 50}}, {name: 1}).**sort("name : -1");**

        console.log(result);

    }catch(err){

        console.log(err);

    }

}

getDocument();

# NOTE: 1 represents ascending order whereas -1 represents descending order.

# MONGODB UPDATE THE DOCUMENTS USING MONGOOSE IN EXPRESS APP

# <https://docs.mongodb.com/manual/reference/operator/update/>

# Example: To update the name of playlist from JS to Javascript

const updateDocument = async () => {

    try{

        const result = await Playlist.**updateOne**({\_id: "6208c449162bf2ca30c22d4c"}, {

**$set: { //what to update with field name and its value**

**name: 'Javascript'**

**}**

        });

console.log(result); **//it will show how many number of records are updated**

    }catch(err){

        console.log(err);

    }

}

updateDocument();

# OR

const updateDocument = async () => {

    try{

        const result = await Playlist.**findByIdAndUpdate**({\_id: "6208c449162bf2ca30c22d4c"}, {

            $set: {

                author: 'Aayush Technical'

            }

        }, {

**new: true, //show modified value on console**

**useFindAndModify: false //use with findByIdAndUpdate method**

        });

        console.log(result);

    }catch(err){

        console.log(err);

    }

}

updateDocument();

# MONGODB DELETE THE DOCUMENTS USING MONGOOSE IN EXPRESS APP

# Example: Delete the document based on \_id

const deleteDocument = async () => {

    try{

        const result = await Playlist.**deleteOne**({\_id: "6208c449162bf2ca30c22d4c"});

    }catch(err){

        console.log(err);

    }

}

deleteDocument();

# OR

const deleteDocument = async () => {

    try{

        const result = await Playlist.**findByIdAndDelete**({\_id: "6208c449162bf2ca30c22d4c"});

    }catch(err){

        console.log(err);

    }

}

deleteDocument();

# NOTE: findByIdAndDelete method will show which record is deleted on console whereas deleteOne method will return number of records deleted on console.

# MONGOOSE BUILT-IN VALIDATION USING MONGODB IN (NODEJS) EXPRESS APP

Before we get into the specifics of validation syntax, please keep the following rules in mind:

* Validation is defined in the SchemaType
* Validation is middleware. Mongoose registers validation as a pre('save') hook on every schema by default.
* You can disable automatic validation before save by setting the validateBeforeSave option
* You can manually run validation using doc.validate(callback) or doc.validateSync()
* You can manually mark a field as invalid (causing validation to fail) by using doc.invalidate(...)
* Validators are not run on undefined values. The only exception is the required validator.
* Validation is asynchronously recursive; when you call Model[#save](https://www.youtube.com/hashtag/save), sub-document validation is executed as well. If an error occurs, your Model[#save](https://www.youtube.com/hashtag/save) callback receives it
* Validation is customizable

<https://mongoosejs.com/docs/validation.html>

##### **All Schema Types**

* required: boolean or function, if true adds a [required validator](https://mongoosejs.com/docs/validation.html#built-in-validators) for this property
* default: Any or function, sets a default value for the path. If the value is a function, the return value of the function is used as the default.
* select: boolean, specifies default [projections](https://docs.mongodb.com/manual/tutorial/project-fields-from-query-results/) for queries
* validate: function, adds a [validator function](https://mongoosejs.com/docs/validation.html#built-in-validators) for this property
* get: function, defines a custom getter for this property using [Object.defineProperty()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty).
* set: function, defines a custom setter for this property using [Object.defineProperty()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty).
* alias: string, mongoose >= 4.10.0 only. Defines a [virtual](https://mongoosejs.com/docs/guide.html#virtuals) with the given name that gets/sets this path.
* immutable: boolean, defines path as immutable. Mongoose prevents you from changing immutable paths unless the parent document has isNew: true.
* transform: function, Mongoose calls this function when you call [Document#toJSON()](https://mongoosejs.com/docs/api/document.html#document_Document-toJSON) function, including when you [JSON.stringify()](https://thecodebarbarian.com/the-80-20-guide-to-json-stringify-in-javascript) a document.

##### **Indexes:** You can also define [MongoDB indexes](https://docs.mongodb.com/manual/indexes/) using schema type options.

* index: boolean, whether to define an [index](https://docs.mongodb.com/manual/indexes/) on this property.
* unique: boolean, whether to define a [unique index](https://docs.mongodb.com/manual/core/index-unique/) on this property.
* sparse: boolean, whether to define a [sparse index](https://docs.mongodb.com/manual/core/index-sparse/) on this property.

##### **String**

* lowercase: boolean, whether to always call .toLowerCase() on the value
* uppercase: boolean, whether to always call .toUpperCase() on the value
* trim: boolean, whether to always call [.trim()](https://masteringjs.io/tutorials/fundamentals/trim-string) on the value
* match: RegExp, creates a [validator](https://mongoosejs.com/docs/validation.html) that checks if the value matches the given regular expression
* enum: Array, creates a [validator](https://mongoosejs.com/docs/validation.html) that checks if the value is in the given array.
* minLength: Number, creates a [validator](https://mongoosejs.com/docs/validation.html) that checks if the value length is not less than the given number
* maxLength: Number, creates a [validator](https://mongoosejs.com/docs/validation.html) that checks if the value length is not greater than the given number
* populate: Object, sets default [populate options](https://mongoosejs.com/docs/populate.html#query-conditions)

##### **Number**

* min: Number, creates a [validator](https://mongoosejs.com/docs/validation.html) that checks if the value is greater than or equal to the given minimum.
* max: Number, creates a [validator](https://mongoosejs.com/docs/validation.html) that checks if the value is less than or equal to the given maximum.
* enum: Array, creates a [validator](https://mongoosejs.com/docs/validation.html) that checks if the value is strictly equal to one of the values in the given array.
* populate: Object, sets default [populate options](https://mongoosejs.com/docs/populate.html#query-conditions)

##### **Date**

* min: Date
* max: Date

##### **ObjectId**

* populate: Object, sets default [populate options](https://mongoosejs.com/docs/populate.html#query-conditions)

**Example**:



# CREATE YOUR OWN CUSTOM VALIDATION USING MONGODB IN (NODEJS) EXPRESS

# Example: Videos should not be less than 0

videos: {

        type: Number,

**validate(value){ //custom validation**

**if(value < 0){**

**throw new Error('Value cannot be less than zero');**

**}**

**}**

    }

**OR**

videos: {

        type: Number,

**validate: { //custom validation**

**validator:function(v){**

**return v.lenght > 0**

**},**

**message: "Videos Count should not be less than zero"**

**}**

    }

# USING NPM VALIDATOR PACKAGE FOR VALIDATION USING MONGODB IN (NODEJS) EXPRESS

# <https://www.npmjs.com/package/validator>

# Install the Validator Package: npm I validator

**Require this package in your project**: const validator = require('validator');

# Example: Email Validator

email: {

        type: String,

        required: true,

        unique: true,

        validate(value){

            if(!**validator.isEmail**(value)){

                throw new Error("Invalid Email");

            }

        }

    }

# REST API POSTMAN

# Postman is an interactive and automatic tool for verifying the APIs of your project. Postman is a Google Chrome app for interacting with HTTP APIs.

# It presents you with a friendly GUI for constructing requests and reading responses. It works on the backend and makes sure that each API is working as intended.

# <https://www.postman.com/downloads/>

# Download the application from above link.

# Create Collection > Add Request > GET API and Test

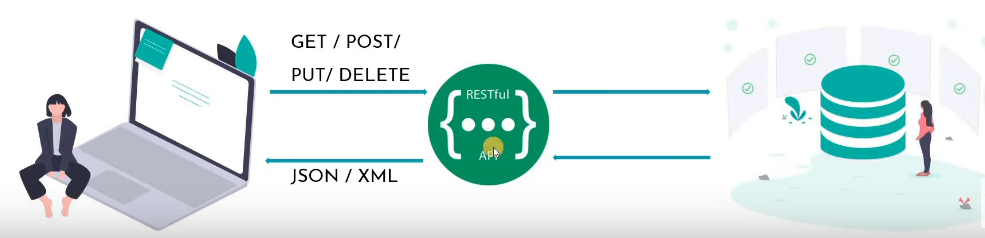
# 

# INTRODUCTION TO RESTFUL API

**Representational State Transfer** is a software architectural style that defines a set of constraints to be used for creating Web services.

Web services that conform to the REST architectural style, called RESTful Web services, provide interoperability between computer systems on the internet.

As per the REST (REpresentational “State” Transfer) architecture, the server does not store any state about the client session on the server-side. This restriction is called Statelessness.





**HTTP Verbs**: POST, GET, PUT, PATCH, DELETE

**RESOURCES**: /api/users

**Resources should not be change**.

# HOW TO CREATE YOUR OWN RESTFUL API (with promises)

# The term REST stands for Representational State Transfer. It is an architectural style that defines a set of rules in order to create Web Services. In a client-server communication, REST suggests to create an object of the data requested by the client and send the values of the object in response to the user.

# 

# Create a folder for your project > Add package.json file by typing: npm init –y > Install Express by typing: npm i express > Install Mongoose by typing: npm i mongoose > Install Validator by typing: npm i validator > Create a folder under your project named as src > Create a file named as app.js under src folder > Code in app.js as:

const express = require('express');

const app = express();

const port = process.env.PORT || 3000;

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

    res.send("Hello");

})

app.post('/students', (req, res) => {

    res.send("Hello");

})

app.listen(port, () => {

    console.log(`Connection is setup at ${port}`);

});

Add these two scripts in package.json file:

**"start": "node src/app.js",**

**"dev": "nodemon src/app.js"**

Run your project as: D:\Learning\resttestApi> **npm run dev**

**Create a new folder under src named as db > Create a new file under db folder named as conn.js > Create a new folder under src named as models > Create a new file under models folder named as students.js**

**NOW START CREATING RESTFUL API (POST data in postman and save in DB)**

**src>db>conn.js (Connection with mongoDB)**

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/students-api')

.then(() => {

    console.log("Connection Successfull");

}).catch((e) => {

    console.log('No Connection');

})

**src>models>students.js (Schema Definition)**

const mongoose = require('mongoose');

const validator = require('validator');

const studentSchema = new mongoose.Schema({

    name: {

        type: String,

        required: true,

        minlength: 3

    },

    email: {

        type: String,

        required: true,

        unique: [true, 'Email is already present'],

        validate(value){

            if(!validator.isEmail(value)){

                throw new Error ("Invalid Email ID");

            }

        }

        },

        phone: {

            type: Number,

            min: 10,

            required: true,

            unique: true

        },

        address: {

            type: String,

            required: true

        }

});

module.exports = mongoose.model('Student', studentSchema); **//Need to export schema in which you want to use**

**src>app.js (Main File)**

const express = require('express');

require('./db/conn'); **//import database connection**

const Student = require('./models/students'); **//import schema definition**

const app = express();

const port = process.env.PORT || 3000;

**app.use(express.json());**

app.post('/students', (req, res) => {

    console.log(req.body);

    const user = new Student(**req.body**); **//to get details from postman**

    user.save().then(() => {

        res.status(201).send(user);

    }).catch((err) => {

        res.status(400).send(err);

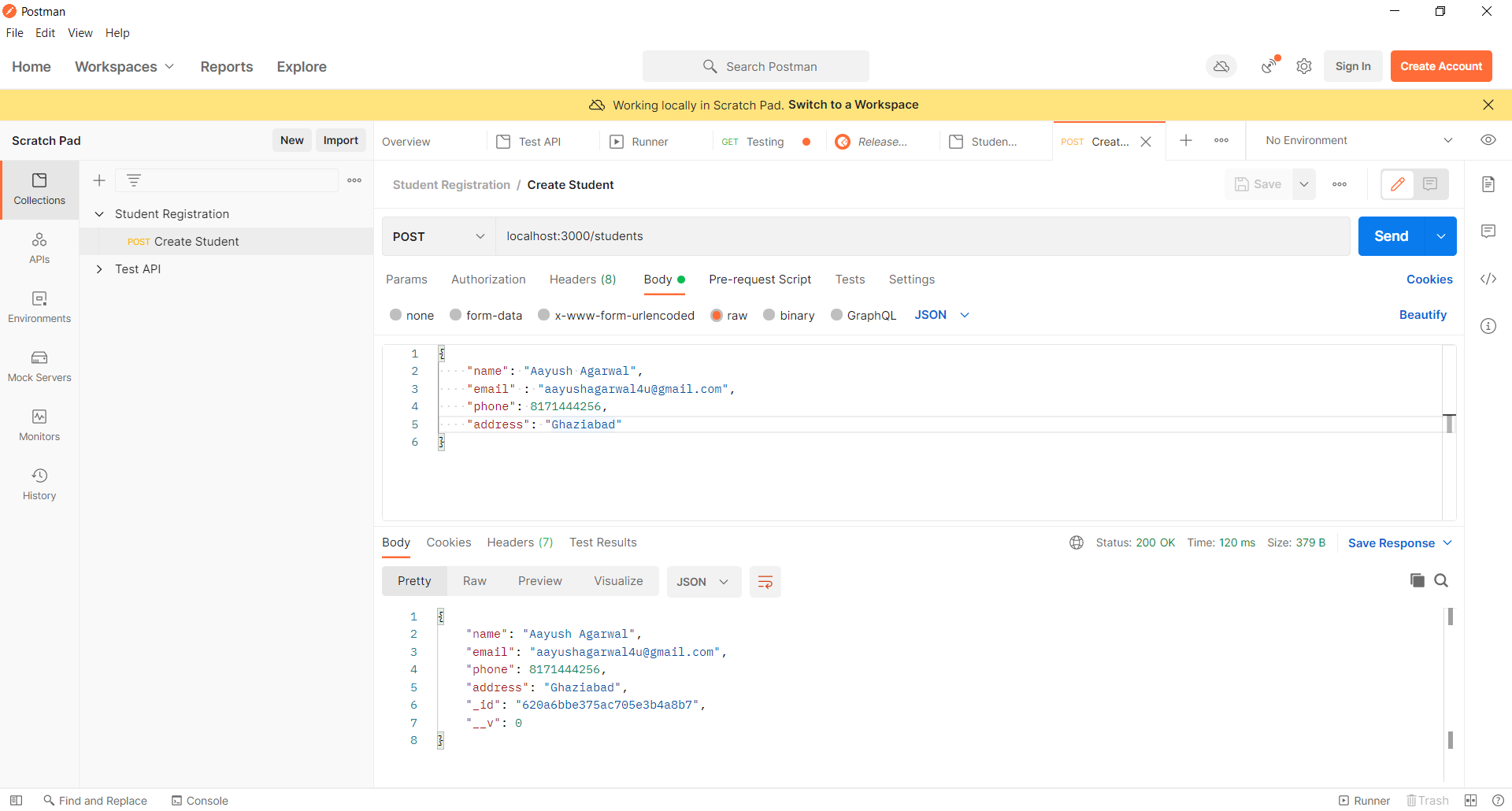
    })

})

app.listen(port, () => {

    console.log(`Connection is setup at ${port}`);

});



# HOW TO CREATE YOUR OWN RESTFUL API (USING ASYNC-AWAIT)

# app.js (change only will be in main file as app.js)

app.post('/students', async(req, res) => {

    try{

        console.log(req.body);

        const user = new Student(req.body);

        const createUser = await user.save();

        res.status(201).send(createUser);

    }catch(e){

        res.status(400).send(e);

    }

})

# HANDLING GET REQUEST IN REST API USING NODEJS & MONGODB

# app.js (change only will be in main file as app.js): Get all registered student detail

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

    try{

const studentsData = await Student.find();

res.send(studentsData);

    }catch(e){

        res.send(e);

    }

})

# OUTPUT (Check URL): http://localhost:3000/students

# app.js (change only will be in main file as app.js): Get detail of particular student

app.get('/students/:id', async(req, res) => {

    try{

        const \_id = req.params.id;

        const studentData = await Student.findById(\_id);

        if(!studentData){

            return res.status(404).send();

        }else{

            res.send(studentData);

        }

    }catch(e){

        res.send(e);

    }

})

# HANDLING PUT/ PATCH REQUEST IN REST API USING NODEJS & MONGODB

# app.js (change only will be in main file as app.js): Update detail of particular student

app.patch('/students/:id', async(req, res) => {

    try{

        const \_id = req.params.id;

        const updateData = await Student.findByIdAndUpdate(\_id, req.body);

        res.send(updateData);

    }catch(e){

        res.send(e);

    }

})

# HANDLING DELETE REQUEST IN REST API USING NODEJS & MONGODB

# app.js (change only will be in main file as app.js): Delete detail of particular student

app.delete('/students/:id', async(req, res) => {

    try{

        const \_id = req.params.id;

        const deleteData = await Student.findByIdAndDelete(\_id);

        if(!deleteData){

            return res.status(404).send();

        }else{

            res.send(deleteData);

        }

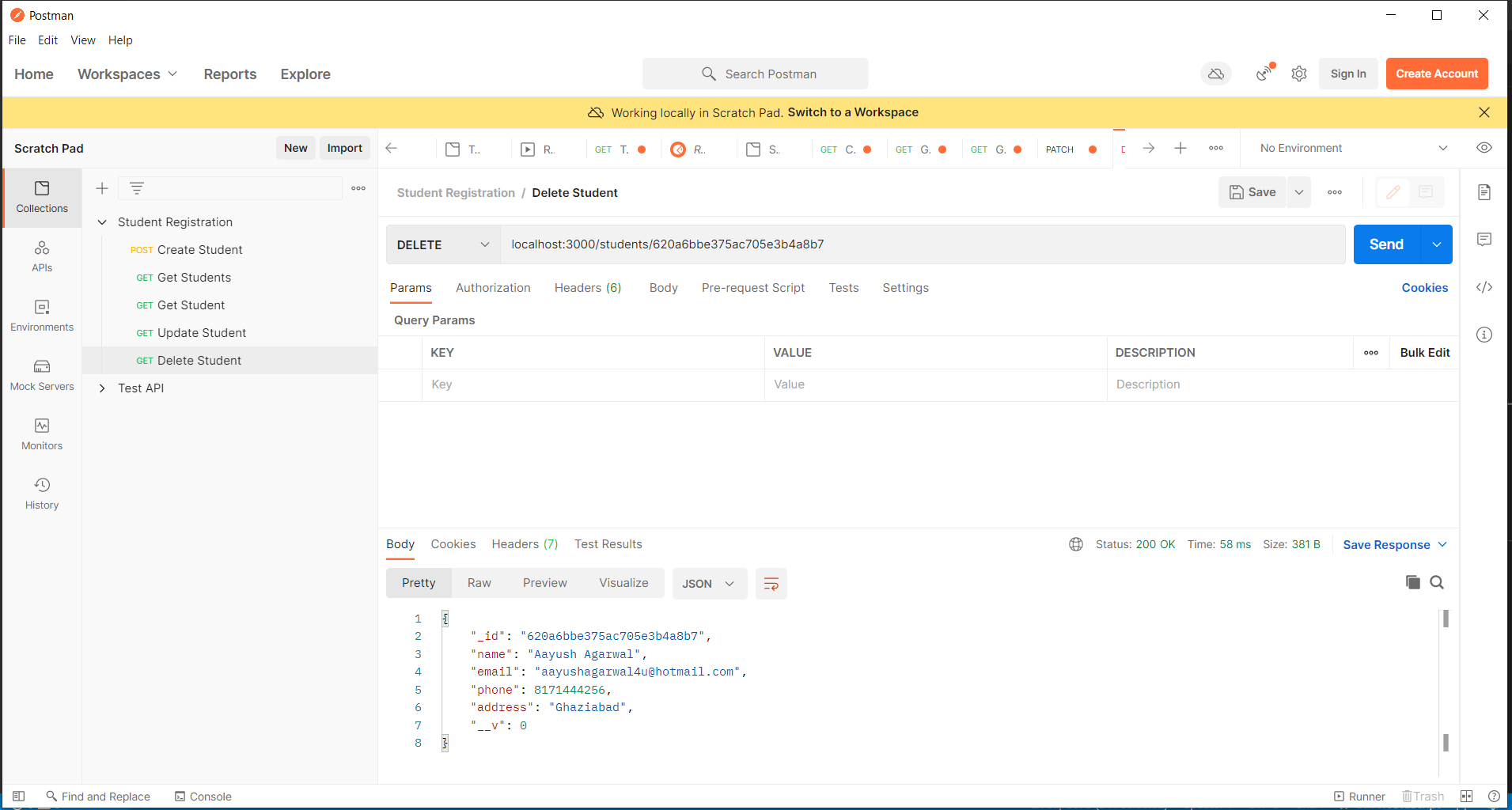
    }catch(e){

        res.send(e);

    }

})

POSTMAN



# ADDING EXPRESS ROUTER IN RESTFUL API

# Create a new folder under src named as routers. Create a new file under routers folder named as student.js

# ./routers/student.js

const express = require('express');

**const router = new express.Router();** **//create a new router**

const Student = require('../models/students');

**router.post**('/students', async(req, res) => { **//define router**

    try{

        console.log(req.body);

        const user = new Student(req.body);

        const createUser = await user.save();

        res.status(201).send(createUser);

    }catch(e){

        res.status(400).send(e);

    }

})

**router.get**('/students', async(req, res) => {

    try{

const studentsData = await Student.find();

res.send(studentsData);

    }catch(e){

        res.send(e);

    }

})

**router.get**('/students/:id', async(req, res) => {

    try{

        const \_id = req.params.id;

        const studentData = await Student.findById(\_id);

        if(!studentData){

            return res.status(404).send();

        }else{

            res.send(studentData);

        }

    }catch(e){

        res.send(e);

    }

})

**router.patch**('/students/:id', async(req, res) => {

    try{

        const \_id = req.params.id;

        const updateData = await Student.findByIdAndUpdate(\_id, req.body);

        res.send(updateData);

    }catch(e){

        res.send(e);

    }

})

**router.delete**('/students/:id', async(req, res) => {

    try{

        const \_id = req.params.id;

        const deleteData = await Student.findByIdAndDelete(\_id);

        if(!deleteData){

            return res.status(404).send();

        }else{

            res.send(deleteData);

        }

    }catch(e){

        res.send(e);

    }

})

**module.exports = router; //export module to app.js**

# app.js

const express = require('express');

require('./db/conn');

const studentRouter = require('./routers/student');

const app = express();

const port = process.env.PORT || 3000;

app.use(express.json());

**app.use(studentRouter);** **//use router**

app.listen(port, () => {

    console.log(`Connection is setup at ${port}`);

});