**TRADER INTERACTIVE - SKILL EXERCISE**

Hariharan Thiyagarajan

**Tech stack for constructing the API**

REST API using

* Node.js
* Express.js
* Mongo DB

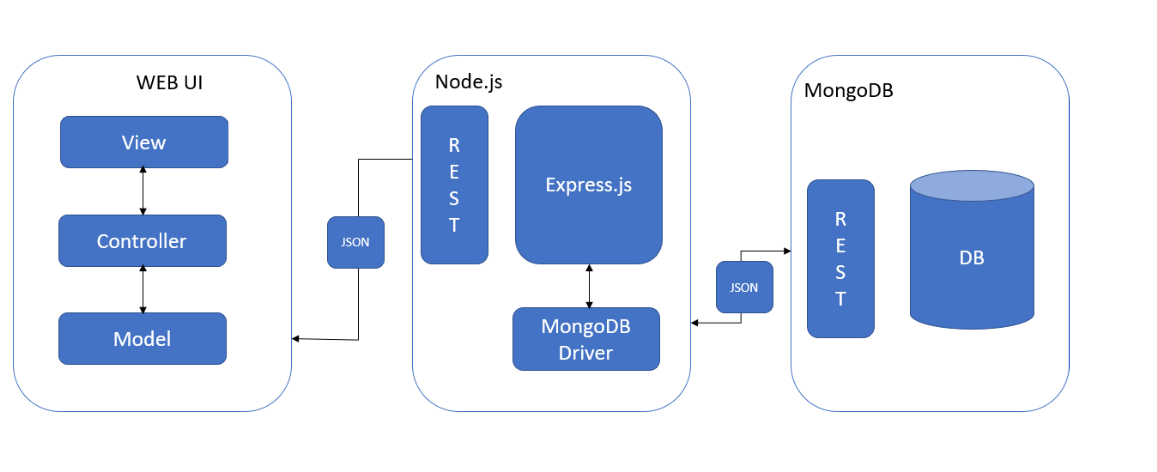
Why REST?

REST API is always independent of the type of the platform or languages. the REST API always adapts to the type of syntax or platforms being used, which gives considerable freedom when changing or testing new environments within the development. Added to that, reliability and scalability is high.

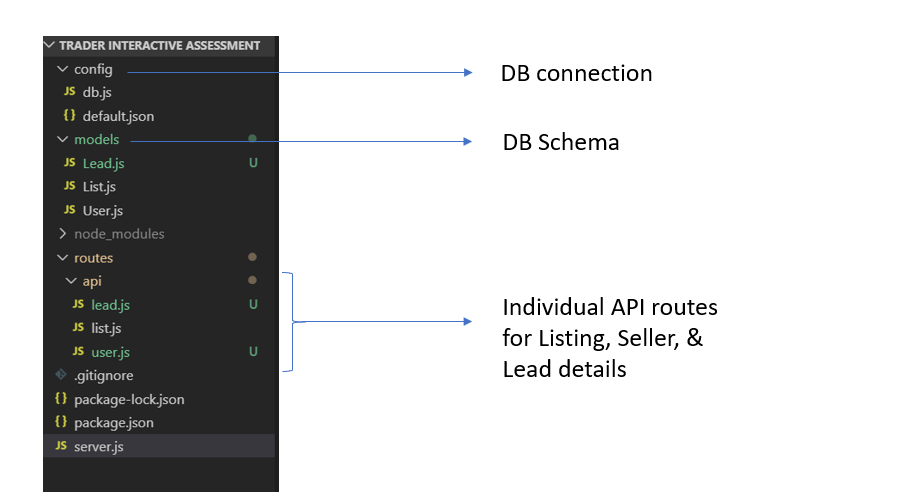
Why Node, Express, Mongo?

* Node.js & Express.js – Using Express framework along with Node.js provide easy way to declare API, handle incoming parameters, errors, transformation to JSON, streaming and sending response.
* MongoDB – In our scenario vehicle data is huge, and the specifications are subjected to change, then schema less & flexible database becomes a natural choice

**Architecture of REST API using Node.js, Express.js & MongoDB**



**Folder Structure**



**API Workflow diagram**

**Data Model**

I’m using Mongo DB to design the database for vehicle listing and selling. It is a NoSQL database and hence the flexibility is more.

There will be three collections, one for listing of vehicles, one for user details and one more for lead details.

**Listing Collection**

****

**User Collection**

****

**Leads Collection**

****

**Relationship between user and listing**

   [ {

        "\_id": "5e3d0e5cd1b083258437f004",

        "year": 2015,

        "make": "Merc",

        "model": "Benz",

        "description": "Good",

        "price": 32000,

        "vehicletype": "Car",

        "meta": {

            "color": "black",

            "transmission": "Gearless"

        },

"images": [

             {

                "\_id": "5e3dcd11e9495041f4a4e474",

                "url": "www.adsasd.com",

                "name": "logo"

             },

             {

                "\_id": "5e3dcd11e9495041f4a4e475",

                "url": "www.asddsd.com",

                "name": "picture2"

             }

        ],

 "user": {

        "user\_id": 1,

        "user\_type": "dealer",

        "name": "rick",

        "address": "Norfolk",

        "phonenumber": "123123123123",

        "email": "rick@gmail.com",

        "website": "merc.com",

"reviews": [

            "good",

            "bad"

        ], },

    }]

**API functional prototype**

**All vehicle listing**

End point: [**http://localhost:3000/api/list**](http://localhost:3000/api/list)

HTTP request: **GET**

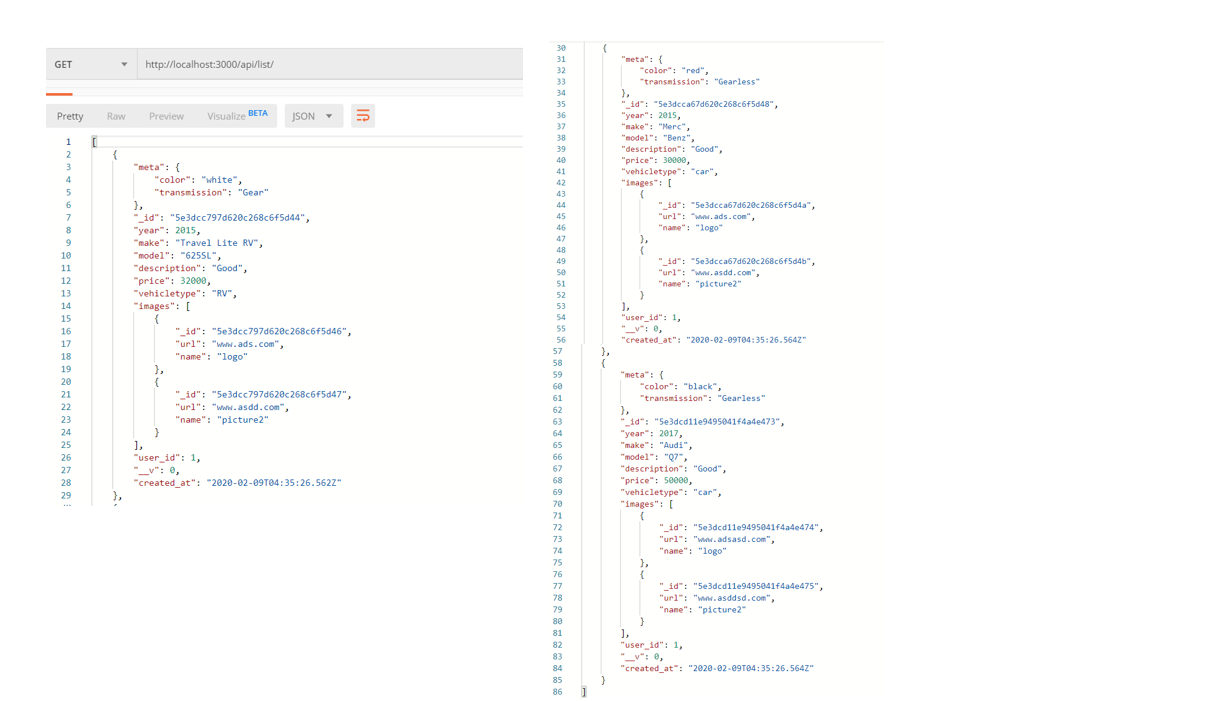
router.get('/', async (req, res) => {

  try {

    const list = await List.find().populate();

    res.json(list);

  }



**User search based on Vehicle Type, Keyword & Price range**

When the user searches for following details in the form

* Vehicle type
* Keyword
  + It can contain anything from model to metadata of the vehicle
* Price range

On form submission, it will hit the following end point.

End point: [**http://localhost:3000/api/list/searchby**](http://localhost:3000/api/list/searchby)

HTTP request: **GET**

In the backend, it will check the query parameter **?type= &keyword= &fromrange= &torange=** selected in the form and finds the corresponding result from the lists collections in DB.

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

  try {

    const list = await List.find({

      vehicletype: req.query.type,

      $or: [

        { year: req.query.keyword },

        { make: req.query.keyword },

        { model: req.query.keyword },

        { description: req.query.keyword },

        { 'meta.color': req.query.keyword },

        { 'meta.transmission': req.query.keyword }

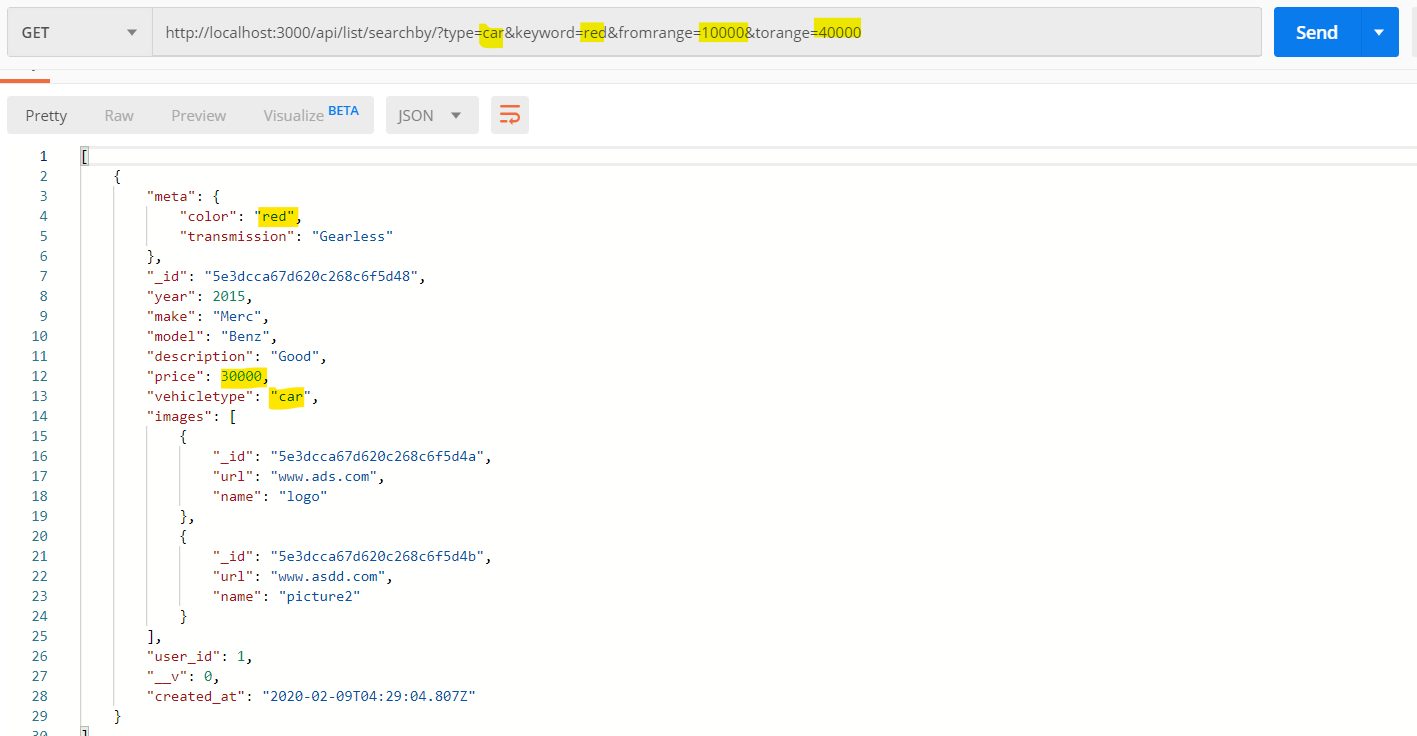
      ],

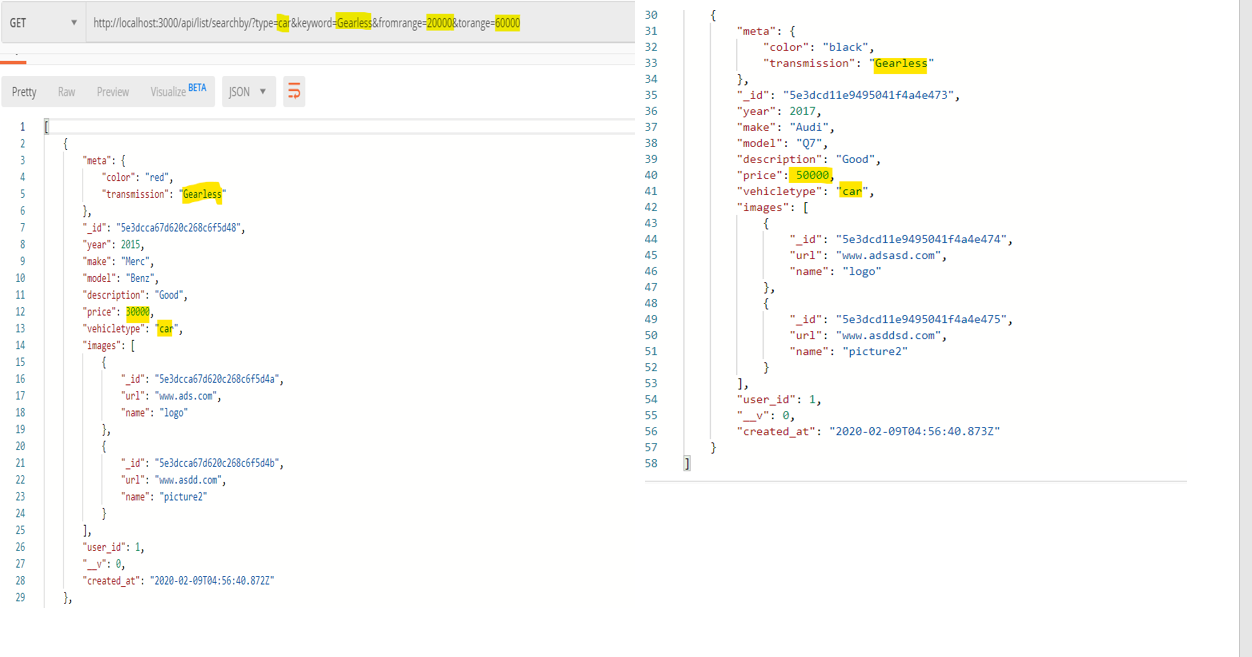
      price: { $gt: req.query.fromrange, $lt: req.query.torange }

    }).populate();

    res.json(list);

The below screenshot provides result for **type=car&keyword=red&fromrange=10000&torange=40000**

****



**Handling pagination in the API**

Since I’m working with dummy data, I’m not able to show the functional prototype of how pagination works in the API. The logical explanation of pagination results as follows:

Endpoint: [**http://localhost:3000/list/**](http://localhost:3000/list/)

The query parameter will be **?pagenum=**

Based on the number entered we can compute the following in the backend

let page\_size = 10;

skips = page\_size \* (req.query.pagenum - 1);

//for page 1

// skips =10 \*(1-1);

//skips =0

// It will fetch first 10 records

//for page 2

// skips =10 \*(2-1);

//skips =10

// It will skip the first 10 records fetches from 10-20

const list = await List.find()

  .skip(skips)

  .limit(page\_size).populate;

Each and every listing result based on the pagination contains data and metadata which can be rendered in the frontend accordingly.

Since multiple results will be populated, in order to access the specific resource, it should hit another end point which carries the unique id of it.

**Access specific resource based on URI**

Endpoint: [**http://localhost:3000/list/searchby/:id**](http://localhost:3000/list/searchby/:id)

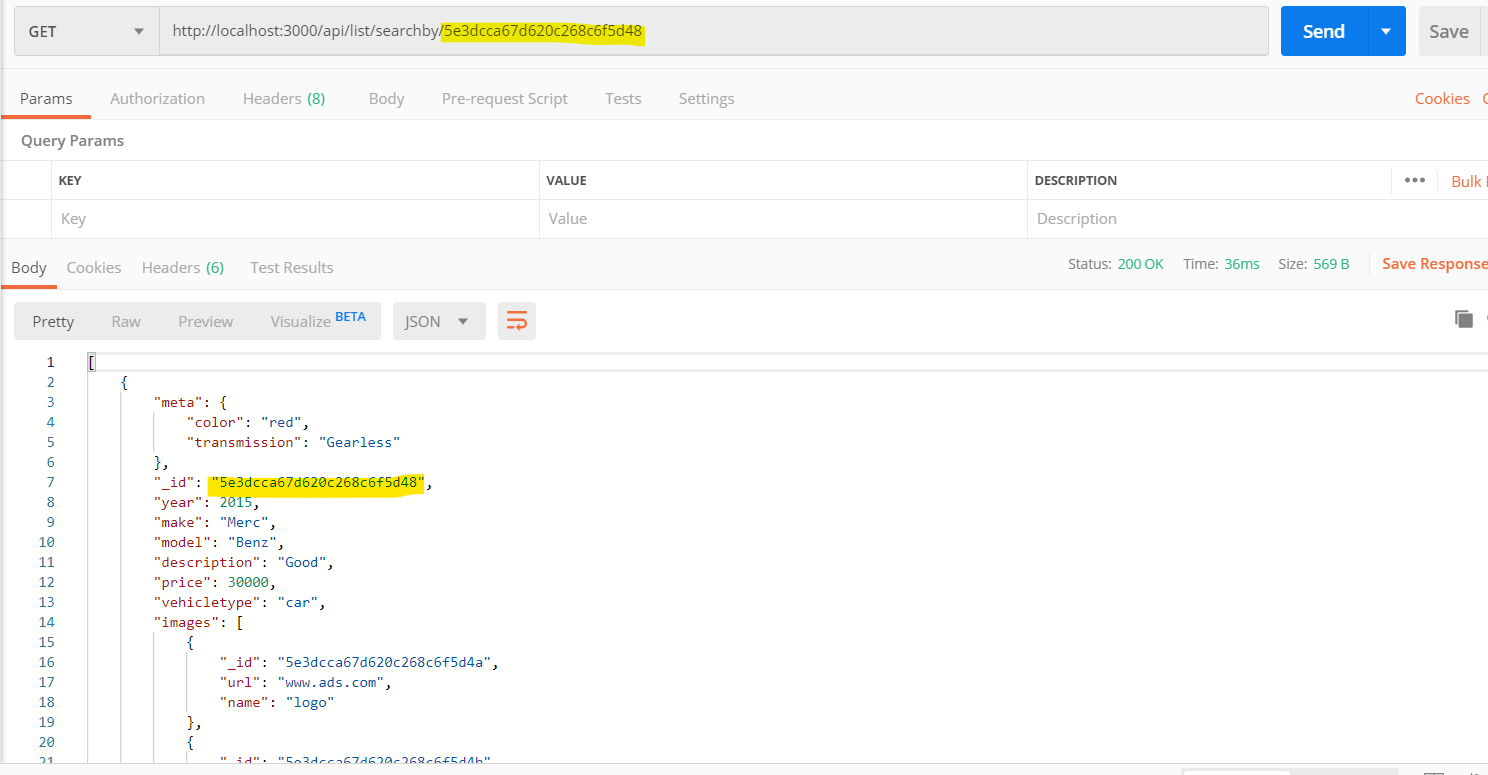
Since ID is unique it will access only that item from DB

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

  try {

    const list = await List.find({ \_id: req.params.id }).populate();

    res.json(list);



**Accessing reviews of the seller**

Each listing will have a user id who posted it, when buyer wants to access the reviews of an user, it will be taken as parameter to fetch the necessary details.

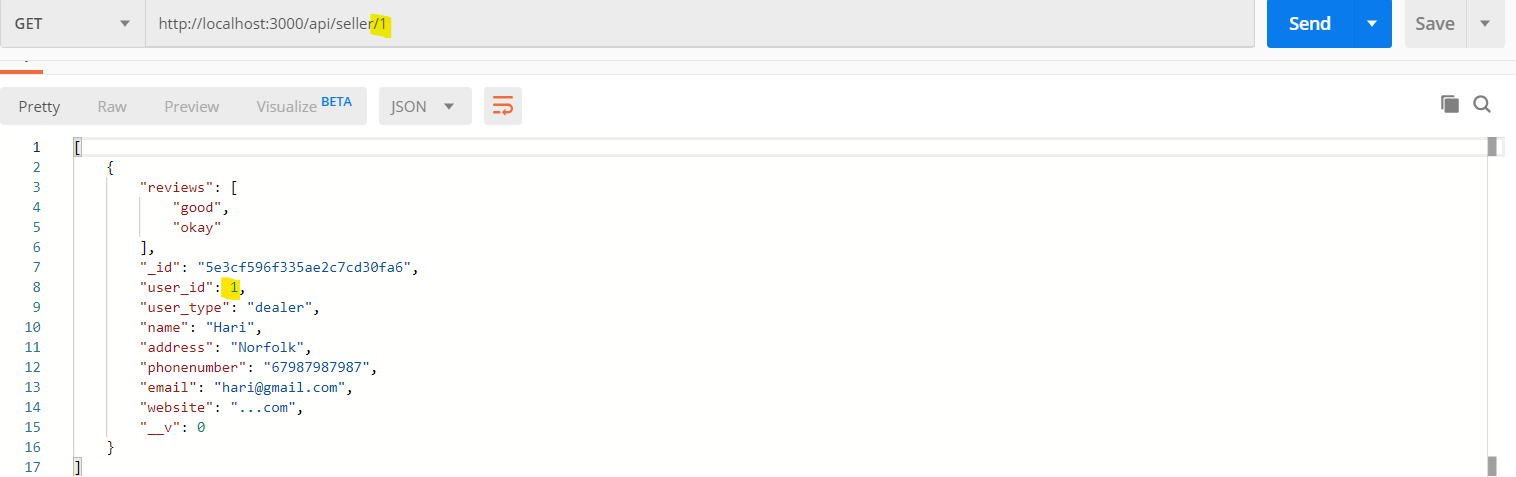
End point: : [**http://localhost:3000/api/seller/:id**](http://localhost:3000/api/seller/:id)

In backend, it will fetch the seller details based on the user\_id

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

  try {

    const seller = await User.find({ user\_id: req.params.id }).populate();



**Lead submission**

Ideally, when the buyer is interested in some vehicle, he/she will submit the following details in the form as a POST request to the API

* Name
* Phone number
* Email
* Comments about the vehicle he/she is interested in

End point: [**http://localhost:3000/api/list**](http://localhost:3000/api/list)

HTTP request: **POST**

Along with this we will be sending vehicle\_id and softdelete option to the dB in the backend, which will be used by the seller to update the status of the lead later.

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

  const { vehicle\_id, name, phonenumber, email, softdelete  } = req.body;

try {

    let lead = new Lead({

      vehicle\_id,

      name,

      phonenumber,

      email,

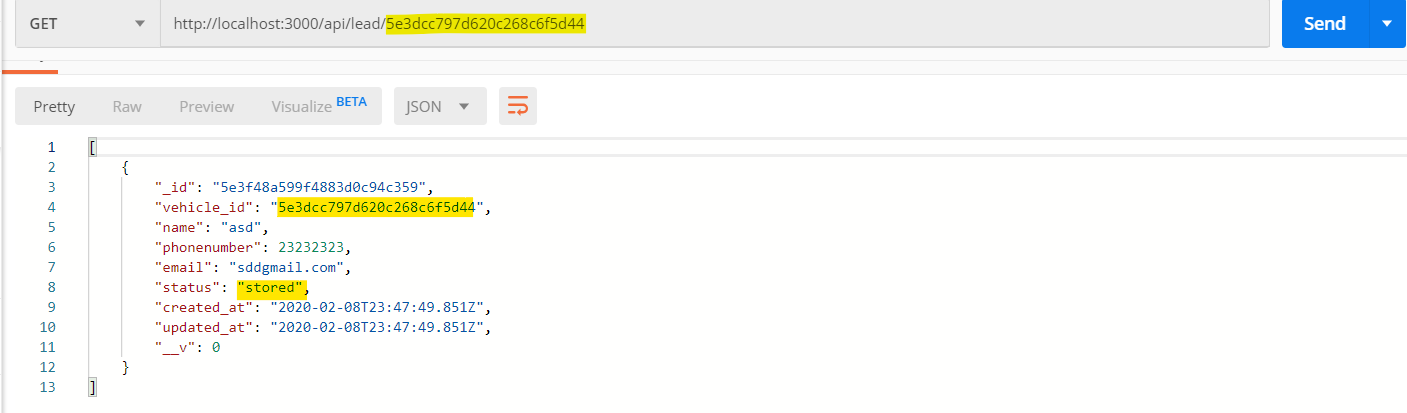
      softdelete

    });

    await lead.save();

  }

The initial status field will be “**stored**” once the buyer submits the request.



Later the status will be changed to “**distributed**” after the verification is done.

In order to change in to “**distributed**”, it should be a **PUT** request which will carry the unique id of each leads

End point: **http://localhost:3000/api/lead/: id**

HTTP request: **PUT**

// change status to distributed

router.put('/:id', async (req, res) => {

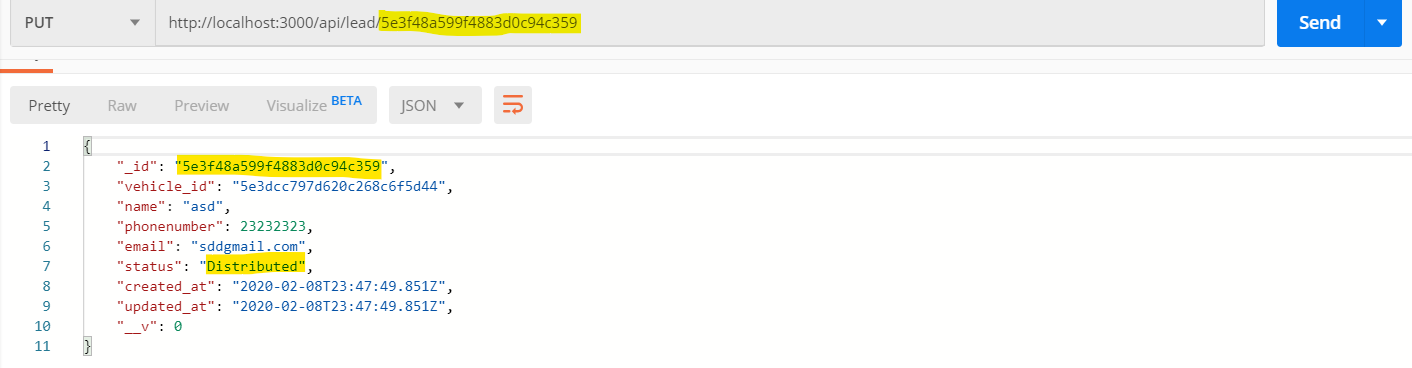
  try {

    let lead = await Lead.findOne({ \_id: req.params.id }).populate();

    lead.status = 'Distributed';

    await lead.save();

    res.json(lead);



**Seller Review of Leads**

When the seller wants to access the leads received, the endpoint should carry the vehicle\_id so that, it will display the leads for the corresponding vehicles

End point: **http://localhost:3000/api/lead/: id**

HTTP request: **GET**

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

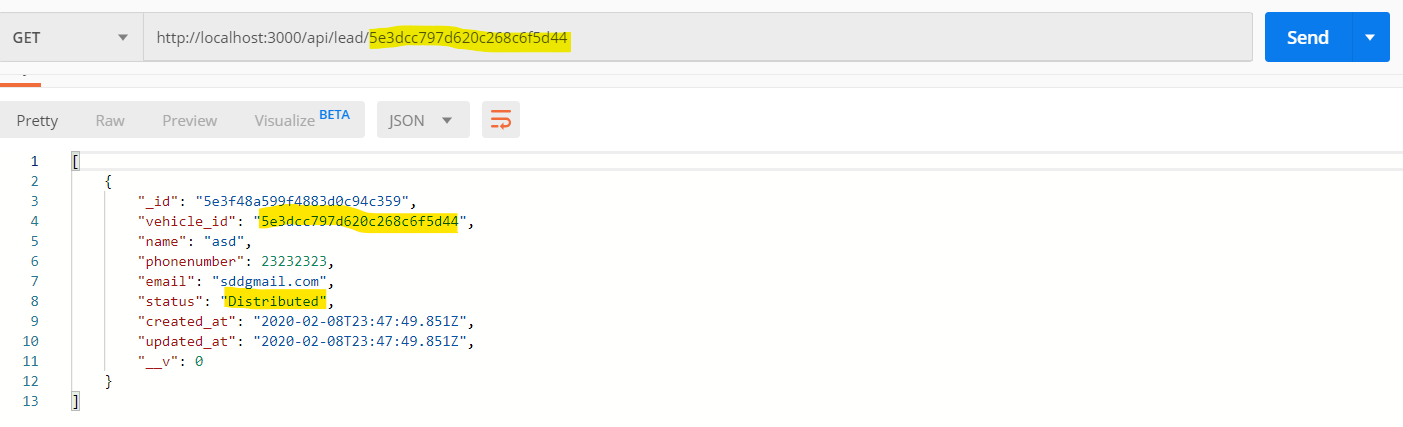
  try {

    const lead = await Lead.find({

      vehicle\_id: req.params.id,

      status: 'Distributed'

    }).populate();

****

The one major drawback is any seller can view anyone’s lead details if they know the endpoint because it is not secured. In order to make it secure, we should use **JWT ( JSON web token)** so when someone else tries to access it, it will throw an error as it won’t match with the secret key.

**Handling Deletion**

If the seller wants to update the status of the lead, he should submit a PUT request to the API

Endpoint: [**http://localhost:3000/api/lead/update/:id**](http://localhost:3000/api/lead/update/:id)

HTTP request: **PUT**

The unique id created when submitting the lead will be taken as a parameter to update the status of the lead.

This request should also be carried out with JWT.

router.put('/update/:id/', async (req, res) => {

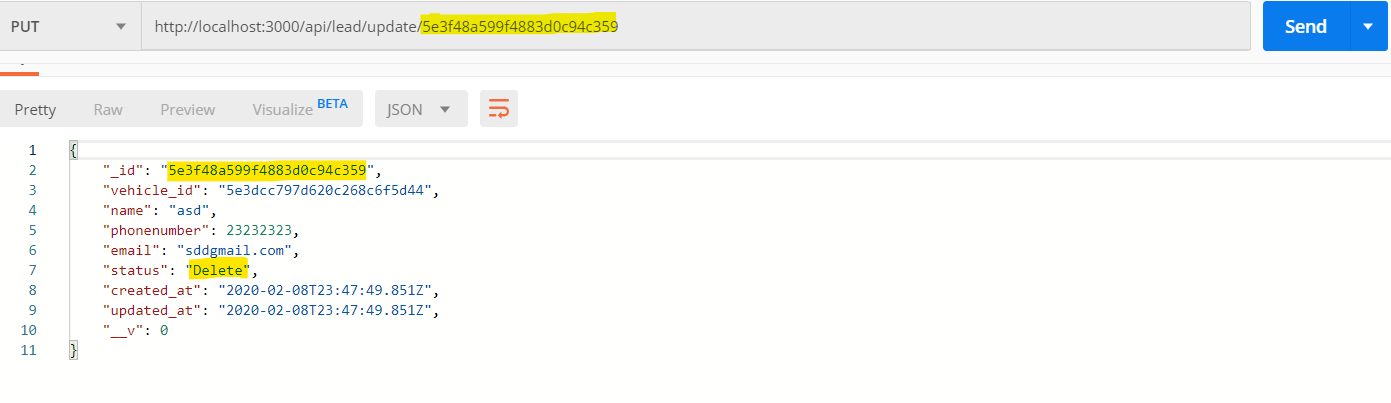
  try {

    let lead = await Lead.findOne({ \_id: req.params.id }).populate();

    lead.status = 'Delete';

    await lead.save();

    res.json(lead);



Source code for all the API routes are available in my GitHub repository

[**https://github.com/hthiyaga/REST-API-using-mongodb-express.js-node.js**](https://github.com/hthiyaga/REST-API-using-mongodb-express.js-node.js)