Developing Pokedex Restful API with Node.js, Express and MongoDB

Backend is technology which works behind the scene to power websites, yet doing it modestly without any fanbase. Allowing people to browse through their favourite websites without having a clue of how much effort it goes into the Backend.

Today we are going to talk about Node.js and its library express which is goto development language for industry leaders including NASA, Airbnb, PayPal, LinkedIn, Netflix, Uber, and Walmart. Surprisingly 98% of fortune 500 companies have adopted NodeJS.

Node.js is javascript runtime built on chrome's V8 engine using which we develop Restful APIs. API is a short form of “Application Programming Interface” which is a piece of code which interacts between multiple software intermediaries analogous to real world examples of a waiter who brings food from the kitchen to the customer and delivers the request of the customer back to the kitchen.

Our Pokedex API

In this blog, we will learn the basics of Node.js development by building a pokedex application. We will build an API which will be able to create, edit,delete and list various pokemons. We will start by creating a simple web server and then connecting the app to the database to store the information.

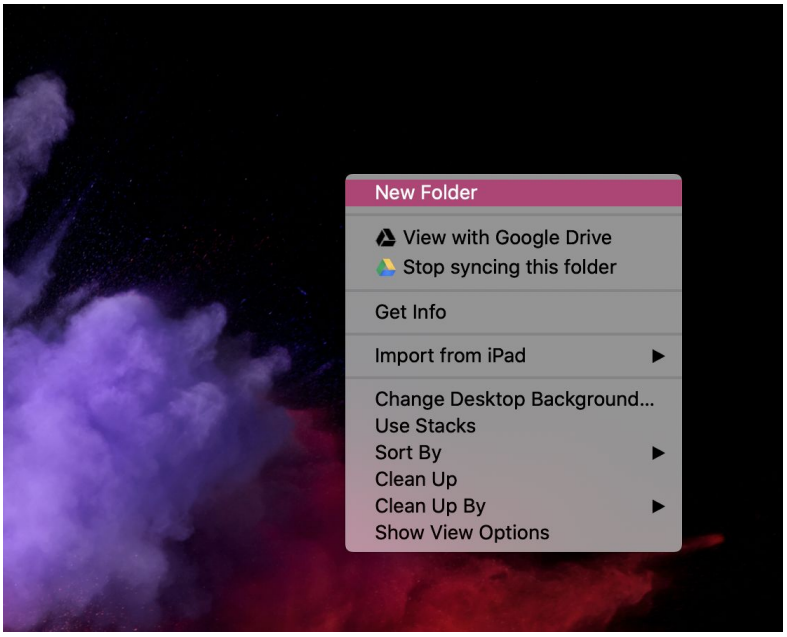
We will make use of various inbuilt features of Javascript, so i would recommend a quick revision of

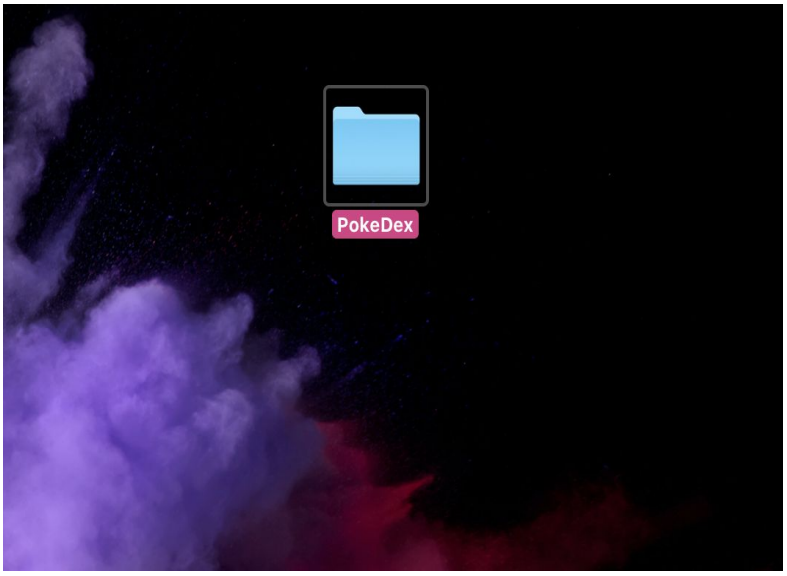
javascript for beginners. Moreover, we will make use of Postman for http requests.

Prerequisites: Node.js(for installation follow https://nodejs.org/en/download/package-manager/)

Lets tickle our brains

1.Create a folder on desktop for the project, all our files would be stored here and it will be our root directory.





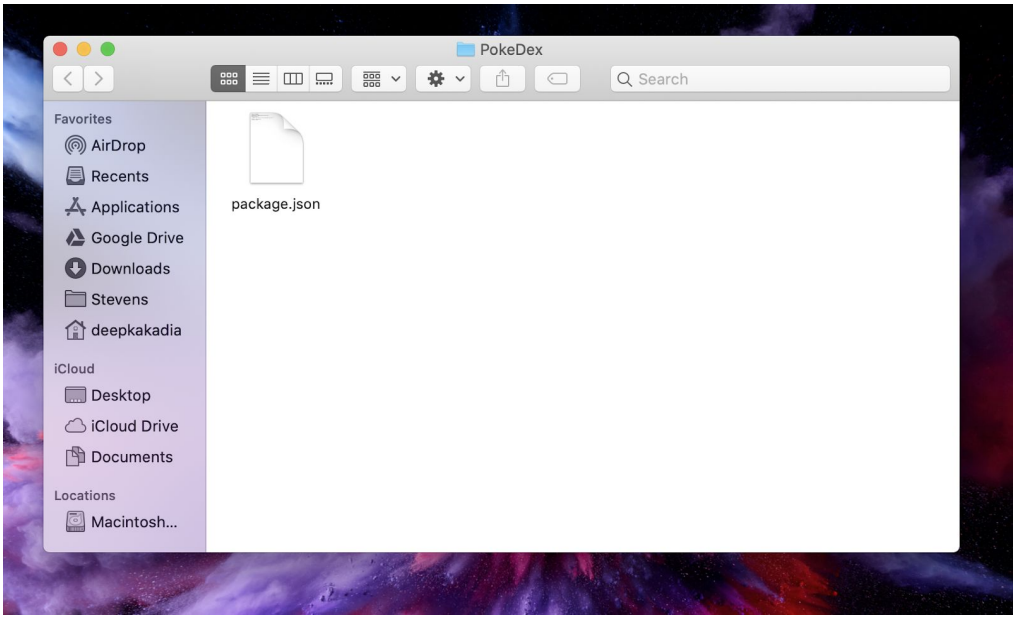
2.Now open terminal at folder, and we will initialize package.json for our pokedex which is a file which contains metadata for our pokedex.

```
cd Poke
$ npm init
```

3.After initializing you will be able to see the following, for time being we just write description and author.Now press enter.

```
Last login: Wed May  6 22:56:12 on console

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(base) Deeps-MBP:PokeDex deepkakadia$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (pokedex)
version: (1.0.0)
description: Pokedex-Gotta Catch'em All
entry point: (index.js)
test command:
git repository:
keywords:
author: Deep Kakadia
license: (ISC)
About to write to /Users/deepkakadia/Desktop/PokeDex/package.json:

{
  "name": "pokedex",
  "version": "1.0.0",
  "description": "Pokedex-Gotta Catch'em All",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Deep Kakadia",
  "license": "ISC"
}


Is this OK? (yes)
```

4. After the above step you will be able to see that it created a package.json file. I'll be using Visual Code as IDE. You can also use different IDEs such as IntelliJ IDEA and Sublime Text.

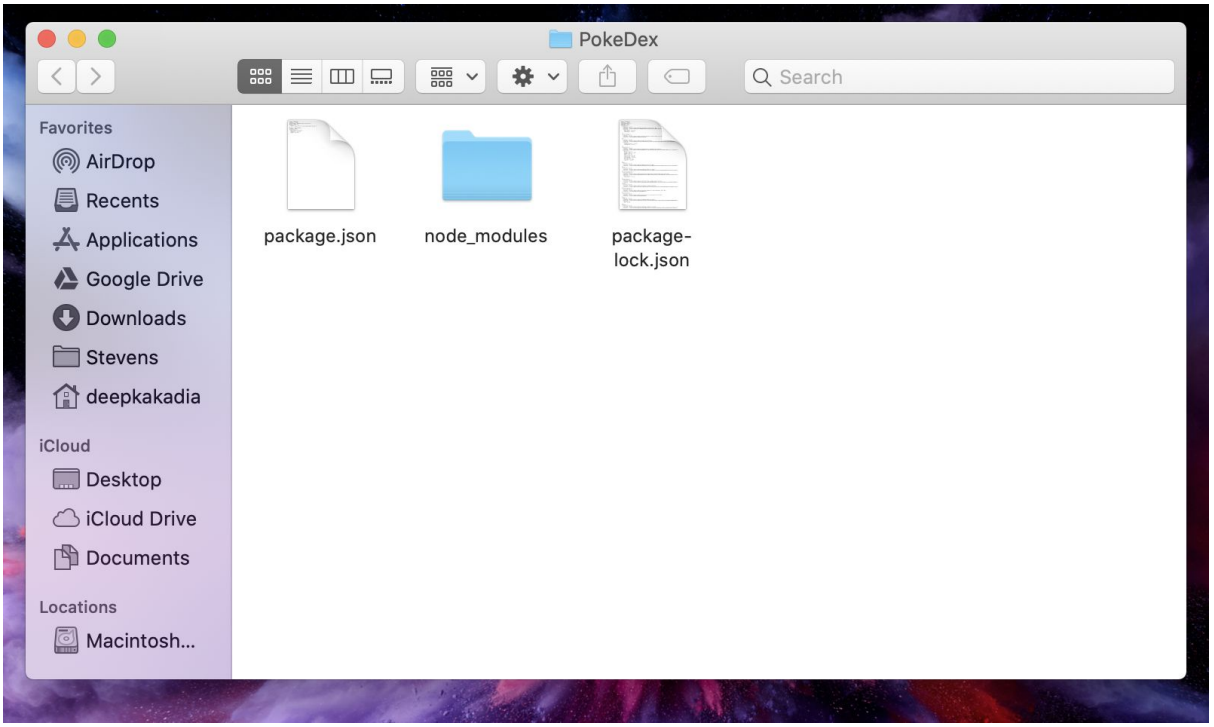


5. Now we will install following dependencies which we will require for the development:

1. Express
2. MongoDB
3. Body-parser

Type the following code in the terminal, it will create a node_modules folder where our dependencies are downloaded and saved.

```
npm install express mongodb body-parser
```



5. Let's start working on our server now as we have environment setup now. Create app.js in the root directory with the following content.

```
const express = require("express");
const bodyParser= require("body-parser");
//creating instance of express
const app = express();

//For importing our routes
const configRoutes = require("./routes");
app.use(bodyParser.urlencoded({extended:true}));

//for parsing data from the browser
app.use(bodyParser.json());
app.use(express.json());
configRoutes(app);

app.listen(3000, () => {
  console.log("We've now got a server!");
      console.log("Your   routes   will   be   running   on
```

```
http://localhost:3000");
```

Create a folder inside the root folder named “routes” which will have two files “index.js” and “pokedexRoutes.js” having the following contents respectively.

Index.js:

```
const pokedexRoutes = require("./pokedexRoutes");

const constructorMethod = app => {
   app.use("/api", pokedexRoutes);


   app.use("*", (req, res) => {
       res.status(404).json({ error: "Not found" });
   });
};

module.exports = constructorMethod;
```

pokedexRoutes.js:

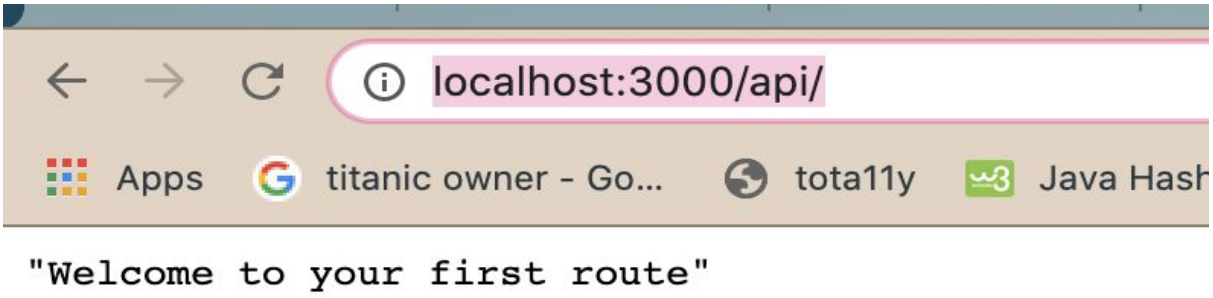
```
const express = require("express");
const router = express.Router();

router.get("/", async (req, res) => {
   try {
       res.json("Welcome to your first route")
   }
   catch (e) {
       res.status(404).json({ message: e })
   }
})

module.exports = router;
```

Here we import the express framework which we will be using to build the API and body-parser is a module which will help to get contents in the form of “req.body”.

Next we created an express app and used middleware functions to parse the requests. Also, we created a constructorMethod which would help in our routing and passed an instance of express app to it, any url starting with “localhost:3000/api” will be directed to pokedexRoutes else any flawed routed will be returned with 404 error.

We have created a simple GET route currently to check our work till now, and we have all this setup on port 3000.

Finally, check your code until now by typing “http://localhost:3000/api/” in your browser. You will be able to see the message.



6. Now let's set up our database, for beginners it's a place where our data will be stored and we can fetch it anytime. For this, we will make use of MongoDB. We create a folder in the root directory named “data” which will have two files namely “mongoConnection.js” used for

creating database and connecting to it and “mongoCollection.js” which will have our collection.

mongoConnection.js:

```
const MongoClient = require("mongodb").MongoClient;

//initial variables for storing the session
let _connection = undefined;
let _db = undefined;

//exporting a async function which connects and creates a database if already not present
module.exports = async () => {
   if (!_connection) {
                                _connection    =    await MongoClient.connect("mongodb://localhost:27017/",       {useNewUrlParser: true, useUnifiedTopology: true });
       _db = await _connection.db("PokeDex_BIT");
   }

   return _db;
};
```

mongoCollection.js:

```
const dbConnection = require("./mongoConnection");


// This will allow you to have one reference to each collection per app
const getCollectionFn = collection => {
   let _col = undefined;

   return async () => {
       if (!_col) {
```

```
            const db = await dbConnection();
            _col = await db.collection(collection);
        }

        return _col;
    };
};

// Now, you can list your collections here:
module.exports = {
    pokemons: getCollectionFn("pokemons"),
};
```

Open mongodb and press the connect button, that’s it for now.

7. Let's start writing our data functions, which will help to perform CRUD(Create,Read, Update, Delete) operations on our database. Make “pokemonFunctions.js” with the following content in the “Data” folder which we created in the last step.

pokemonFunctions.js:

```
const mongoCollections = require('./mongoCollections');
const pokemons = mongoCollections.pokemons;
const { ObjectId } = require("mongodb").ObjectId;

async function createPokemon(name, rank, photo = "null")
{
    // name and rank are required fields whereas photo
isn't
    if (!name || typeof (name) !== "string") throw "You
must provide a name for your pokemon";
    if (!rank || typeof (rank) !== "number") throw "You
must provide rank for your pokemon";
```

```
    //awaiting the collection "pokemon" from mongodb
    const pokemonCollection = await pokemons();

    let newpokemon = {
        name: name,
        rank: rank,
        photo: photo
    };
                    const       insertInfo       =       await
pokemonCollection.insertOne(newpokemon);
     if (insertInfo.insertedCount === 0) throw "Could not
add pokemon";

    const newId = insertInfo.insertedId;
    const poke = await getPokemon(newId);
    return poke;
}

async function getAll() {
    const pokemonCollection = await pokemons();
       //mongo  query  which  returns  all  entries  in  the
collection
    const a1 = await pokemonCollection.find({}).toArray();
    //error check to see if there are no current pokemons
    if (!a1) throw "No pokemons are currently present";
    return a1;
}

async function getPokemon(id) {
    const pokemonCollection = await pokemons();
     const  user  =  await  pokemonCollection.findOne({  _id:
ObjectId(id) });
    //error check if no pokemon with that id
    if (!user) throw 'Pokemon not found';
    return user;
}
```

```
async function removePokemon(id) {
    if (!id) throw 'You must provide an id for deleting
Pokemon';
   if (!ObjectId.isValid(id)) throw "Please input a valid
Id of Pokemon"
   const pokemonCollection = await pokemons();
                const      deletionInfo      =      await
pokemonCollection.removeOne({ _id: ObjectId(id) });
   if (deletionInfo.deletedCount === 0) {
       throw `Could not delete pokemon with id of ${id}`;
   }
   else {
       return {
           deleted: true,
       }
   }
}

async function updatePokemon(id, newName, newRank) {
   if (!id) throw 'You must provide an id for Pokemon';
   if (!ObjectId.isValid(id)) throw "Please input a valid
Id"
    if (!newName || typeof (newName) !== "string") throw
"You must provide new name";
    if (!newRank || typeof (newRank) !== "string") throw
"You must provide new rank";
   const pokemonCollection = await pokemons();
   const updatedpokemon = {
       name: newName,
       rank: newRank,
   };
   const updatedInfo = await
pokemonCollection.updateOne({  _id:  ObjectId(id)  },  {
$set: updatedpokemon });
   if (updatedInfo.modifiedCount === 0) {
       throw 'could not update animal successfully';
   }
```

```
    return await this.getAnimalById(id);
}

module.exports = {
    getPokemon,
    getAll,
    createPokemon,
    updatePokemon,
    removePokemon
}
```

8. Now we will integrate our data functions with their corresponding routes. So, that we can test our API. Put the the following content in the pokedexRoutes.js

pokedexRoutes.js:

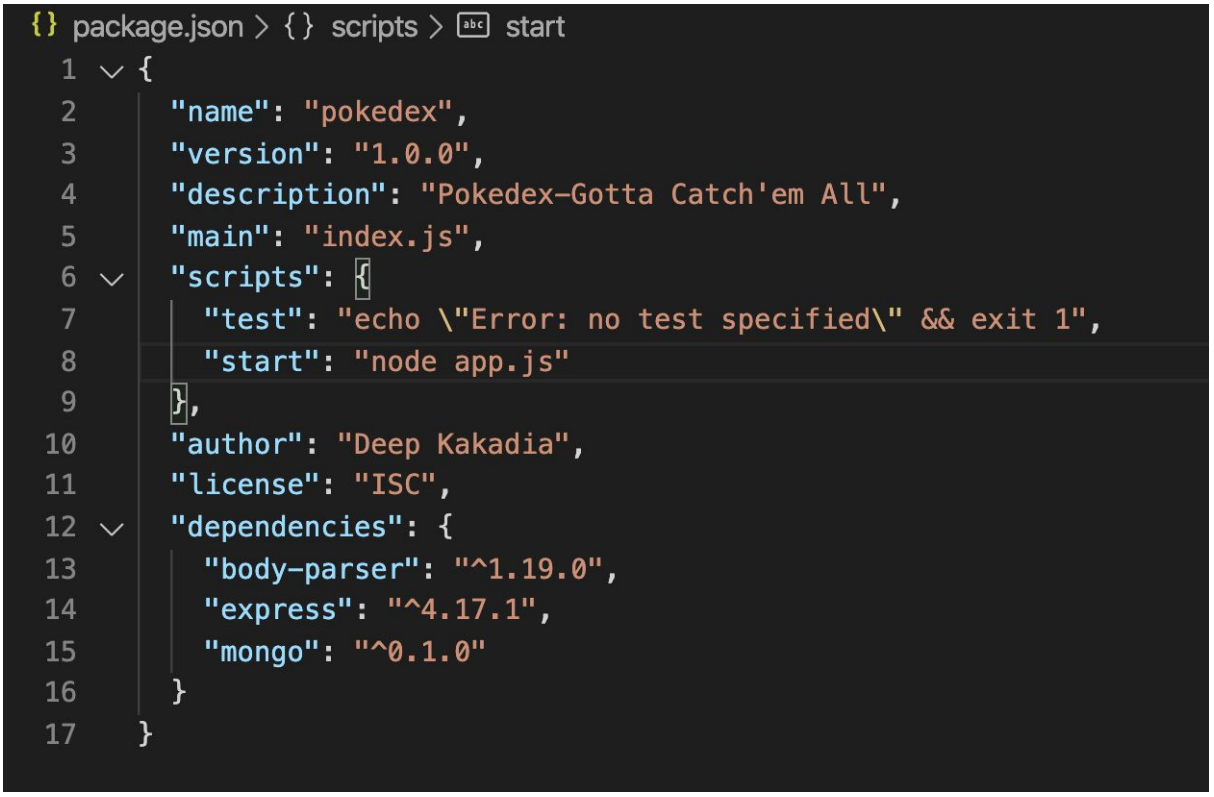
```
const express = require("express");
const router = express.Router();
const pokemonData = require("../data/pokemonFunctions");

router.get("/", async (req, res) => {
    try {
        const pokemon = await pokemonData.getAll();
        res.json(pokemon)
    }
    catch (e) {
        res.status(404).json({ message: e })
    }
})

router.get("/:id", async (req, res) => {
    try {
                                const    poke1    =    await
pokemonData.getPokemon(req.params.id);
        res.json(poke1);
```

```
    } catch (e) {
        res.status(404).json({ message: "not found!" });
    }
});

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

    if (!pokeInfo) {
          res.status(400).json({ error: 'You must provide
data to update animal' });
        return;
    }

    if (!pokeInfo.name) {
          res.status(400).json({ error: 'You must provide
name' });
        return;
    }
    if (!pokeInfo.rank) {
          res.status(400).json({ error: 'You must provide
rank' });
        return;
    }

    try {
        await pokemonData.getPokemon(req.params.id);
    } catch (e) {
         res.status(404).json({ error: 'Pokemon not found'
});
        return;
    }
    try {
                        const   updatedpokemon   =   await
pokemonData.updatePokemon(req.params.id,   pokeInfo.name,
pokeInfo.rank);
        res.json(updatedpokemon);
```

```
    } catch (e) {
        res.status(400).json({ error: e });
    }
});

router.delete("/:id", async (req, res) => {
    try {
                                const    poke1    =    await
pokemonData.removePokemon(req.params.id);
        res.json(poke1);
    } catch (e) {
        res.status(404).json({ message: "not found!" });
    }
});

router.post("/", async (req, res) => {
    let pokeInfo = req.body;

    if (!pokeInfo.name) {
          res.status(400).json({ error: 'You must provide
name' });
        return;
    }

    if (!pokeInfo.rank) {
          res.status(400).json({ error: 'You must provide
rank' });
        return;
    }

    try {
                            const    newpokemon    =    await
pokemonData.createPokemon(pokeInfo.name, pokeInfo.rank);
        res.json(newpokemon);
    } catch (e) {
        res.sendStatus(500);
    }
```

```
});

module.exports = router;
```

8. Lastly add the start script as in Line 8 in your package.json file to get things working.

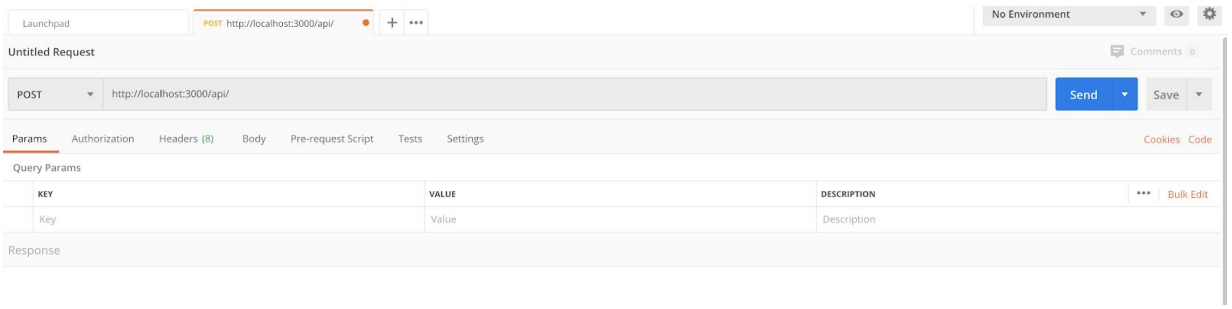


If you made it this far, you have created your first working API. Congratulations! Now let's test the API.

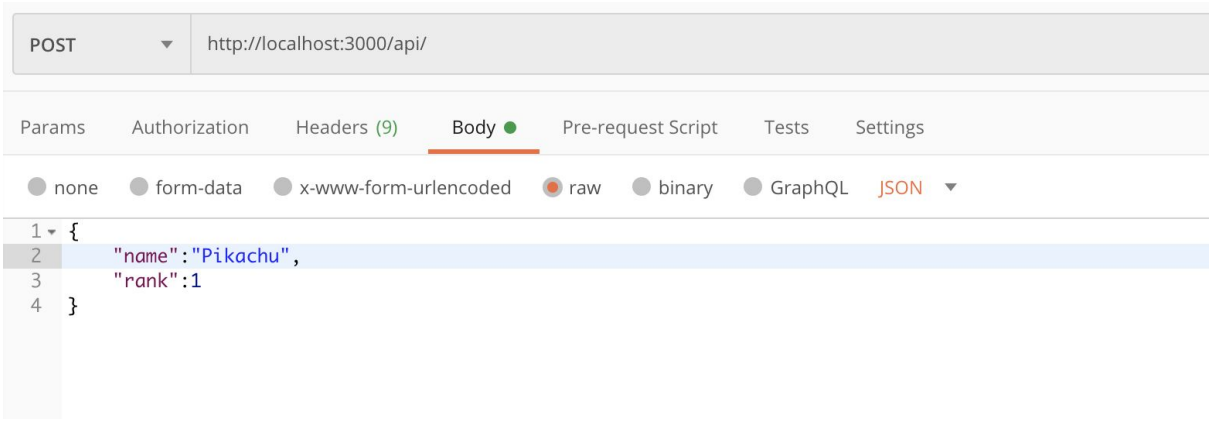
Testing Our API:

If you have not Postman installed on your system you can use this link https://www.postman.com/downloads/. We will be using Postman for testing our routes.

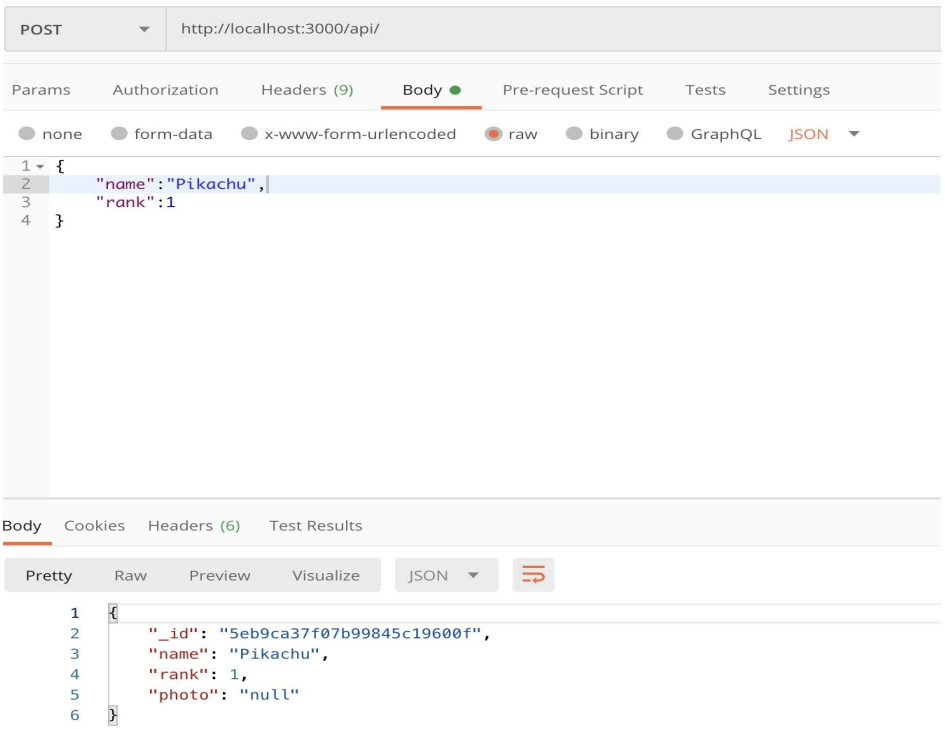
1. First let's start by adding a pokemon to our pokedex, for which we will call POST route, where we will provide name and rank.
Let's write our POST route:

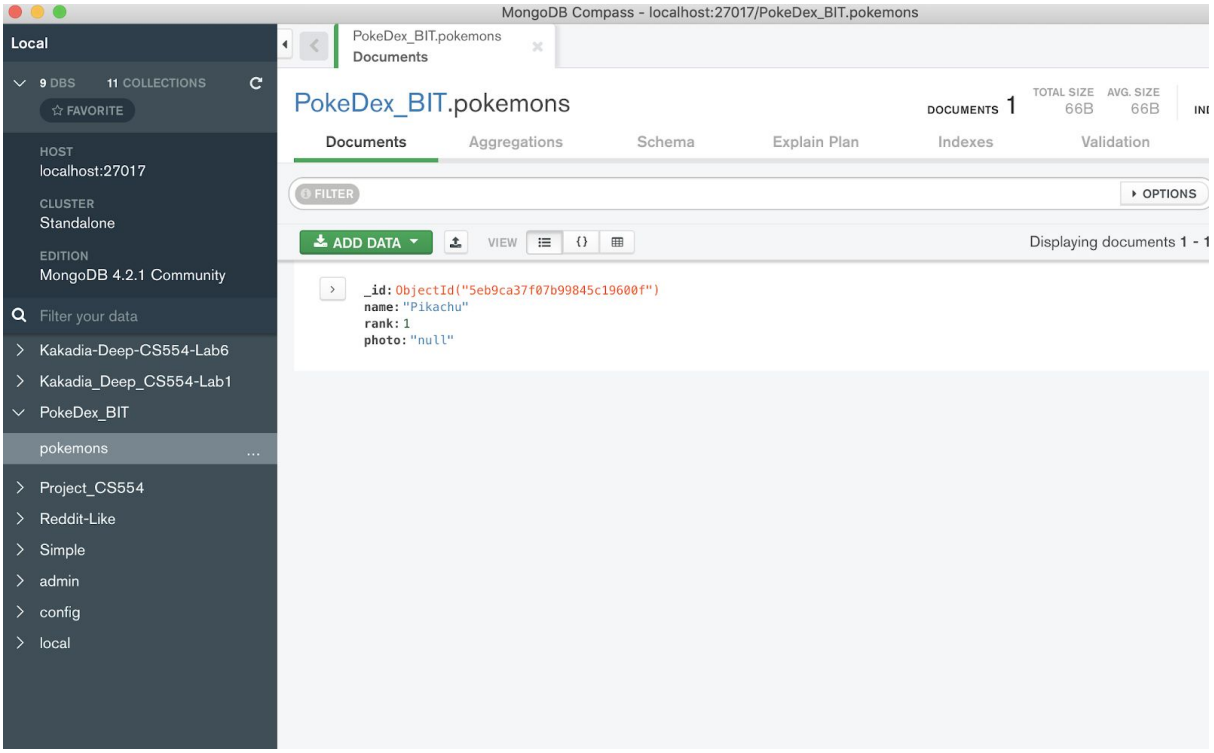


Now click on the Body tab and select raw in that. After that provide name and rank as JSON:

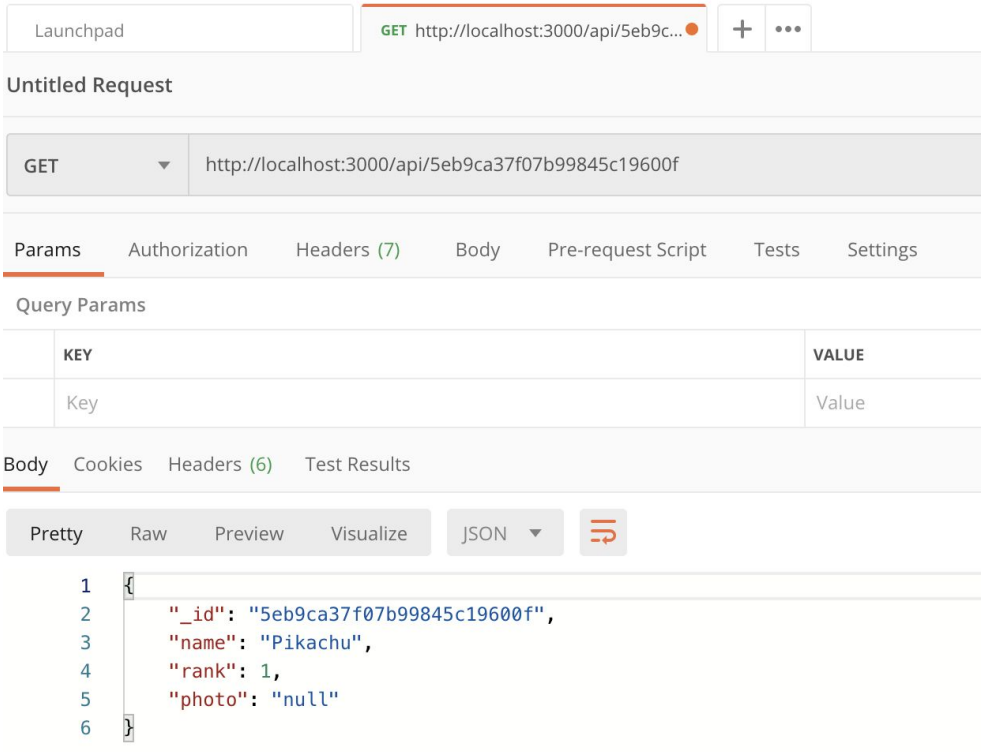


Click Enter and it should create entry in our Database with following details and respond us with same object



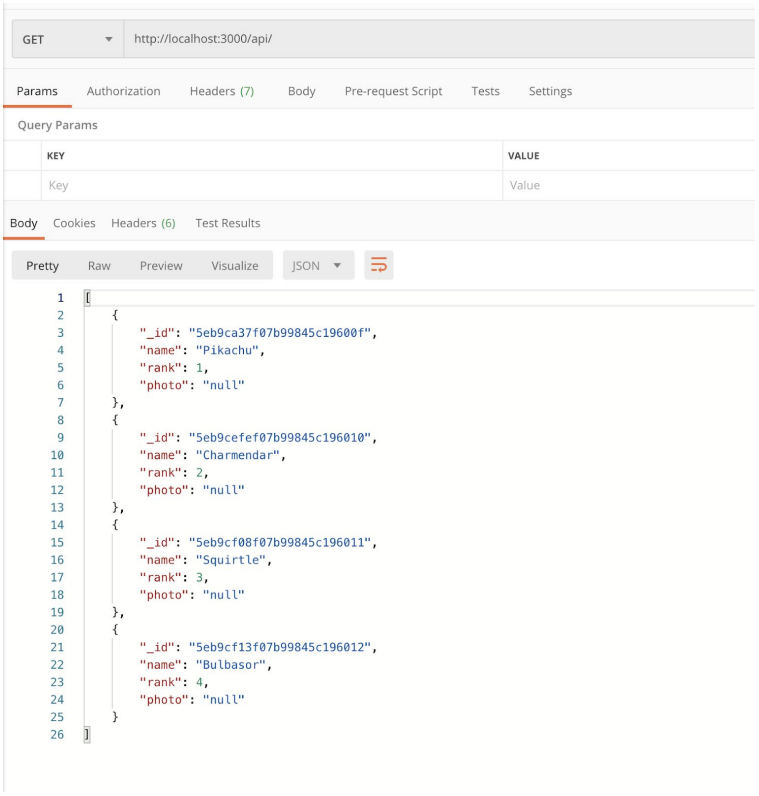


2. Let's try to fetch this pokemon using GET route "localhost:3000/api/:id" which we had written, instead of id here we will pass id of pikachu.

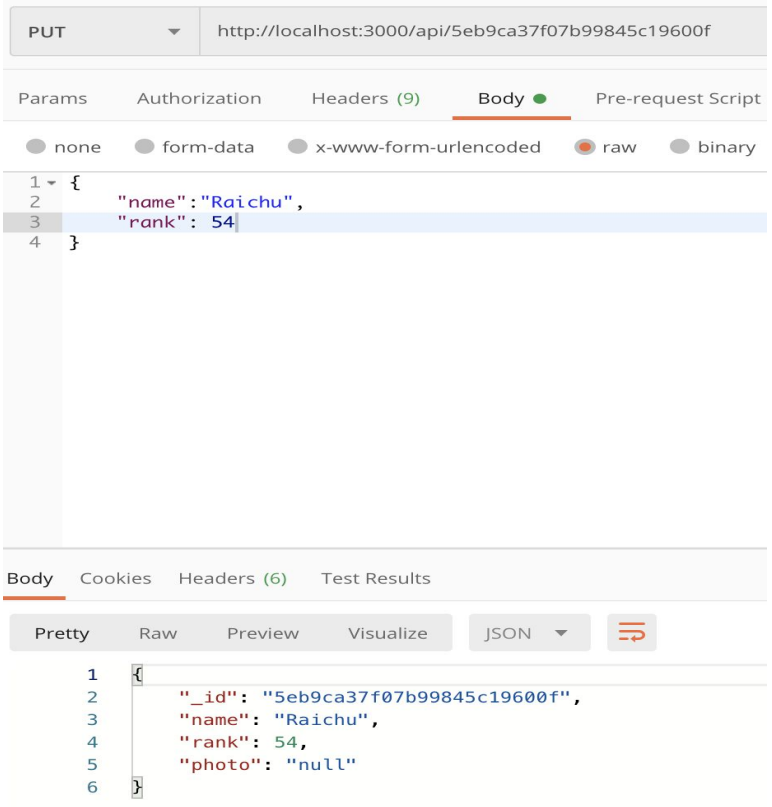


As you can see in the above picture we were able to fetch details using only id. Now i have added a few pokemons to the database and we will try to retrieve all of them.

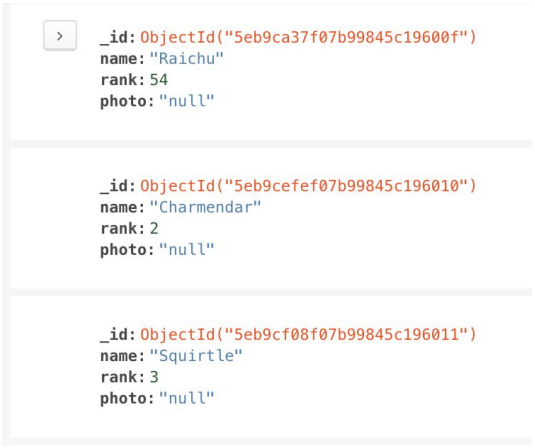
3. Let's try to fetch all the pokemons in the database using our GET “localhost:3000/api/” route.



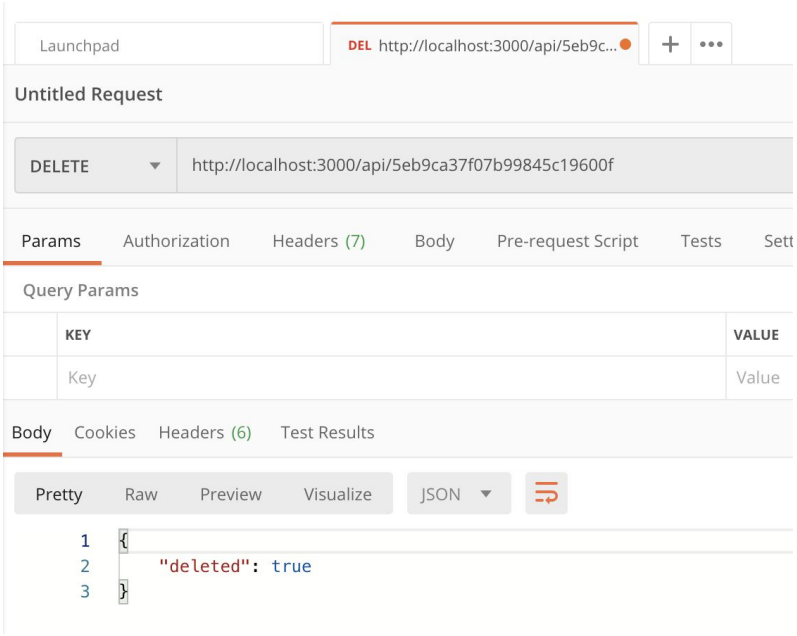
4.Now let's try to update the name of pikachu to raichu using our PUT route “localhost:3000/api/:id” where we will replace id with id of pikachu as stated previously.



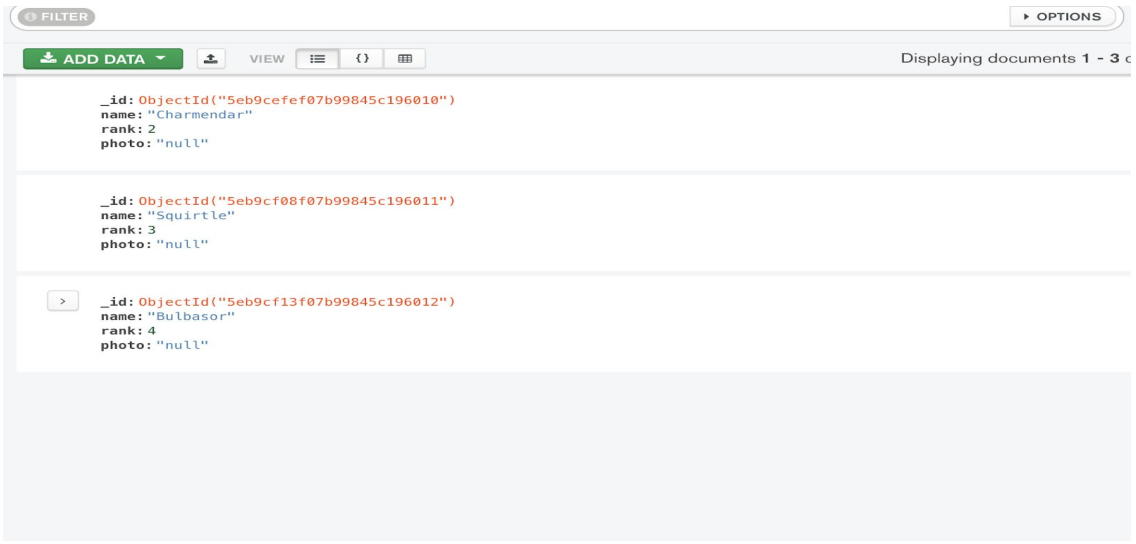
As you can see we have successfully changed the name to raichu and rank to 54. Also it will be changed in the database.



5.Lastly let's try to delete the record of newly created “Raichu” from the database. For this we will use the DELETE route “localhost:3000/api/:id”.



Let's check the same thing in our database, we can see that it has been successfully deleted in the database too.



Conclusion

You have taken your first step towards being a successful backend developer by building this API. You can find the complete code at https://github.com/deepkakadia/Blog-Pokedex. Hope you enjoyed.