

Making REST API

04 October 2023 12:02 PM

Creating server with express
npm i express, npm i nodemon
Let us make app.js file now,

```
app.js x
app.js > ...
1 const express = require("express");
2 const app = express();
3
4 const PORT = process.env.PORT || 5000;
```

```
const PORT = process.env.PORT || 5000;

app.get("/", (req, res) => {
  res.send("Hi, I am live ");
});

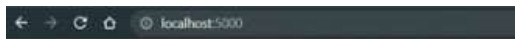
const start = async () => {
  try {
    app.listen(PORT, () => {
      console.log(`${PORT} Yes I am connected`);
    });
  } catch (error) {
    console.log(error);
  }
};

start();
```

Add scripts in package.json

```
Debug
"scripts": {
  "start": "node app.js",
  "run": "nodemon app.js"
},
```

Now we run and we get

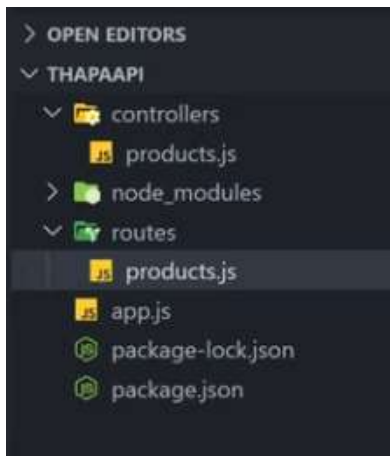


Hi, I am live

Let us now setup routes and controllers,
In bigger apps, people use routers and controllers.

We make folder route and controllers folder

Inside routes we make products.js



Now we make routes inside it and define our routes.

```
routes > products.js > ...  
1  const express = require("express");  
2  const router = express.Router();  
3  
4  router.route("/").get(getAllProducts);  
5  router.route("/testing").get(getAllProductsTesting);  
6  
7  module.exports = router;  
8
```

Inside Controllers we define getAllProducts and getAllProductsTesting function

```

controllers > products.js > ...
1  const getAllProducts = async (req, res) => {
2    res.status(200).json({ msg: "I am getAllProducts" });
3  };
4
5  const getAllProductsTesting = async (req, res) => {
6    res.status(200).json({ msg: "I am getAllProductsTesting" });
7  };
8
9  module.exports = { getAllProducts, getAllProductsTesting };
10

```

Now we require these in our routes

```

const {
  getAllProducts,
  getAllProductsTesting,
} = require("../controllers/products");

router.route("/").get(getAllProducts);
router.route("/testing").get(getAllProductsTesting);

module.exports = router;

```

Now we import our routes inside app.js
For that we use middleware

Inside app.js

```

const products_routes = require("../routes/products");

```

```

// middleware or to set router
app.use("/api/products", products_routes);

```

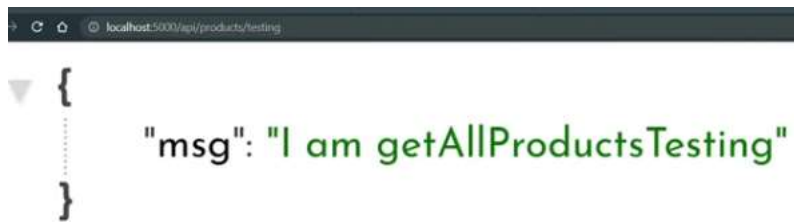
Now inside localhost:5000/api/products we get data of getAllProducts function.
As our routes looks like

```

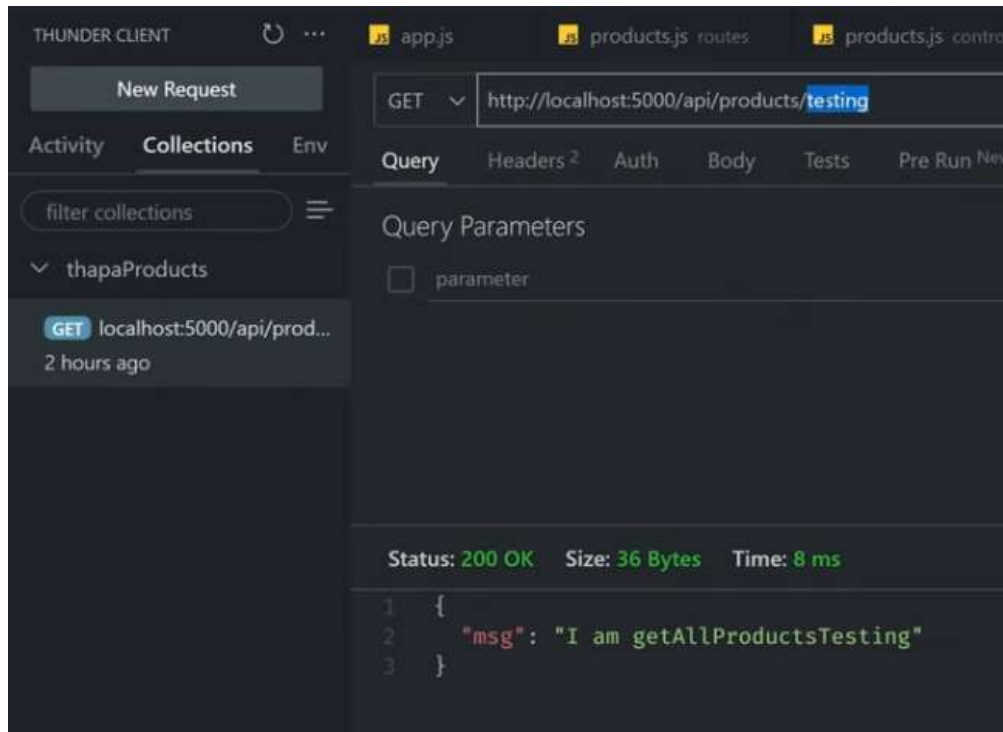
router.route("/").get(getAllProducts);
router.route("/testing").get(getAllProductsTesting);

```

If we use localhost:5000/api/products/testing we get content of getAllProductsTesting function



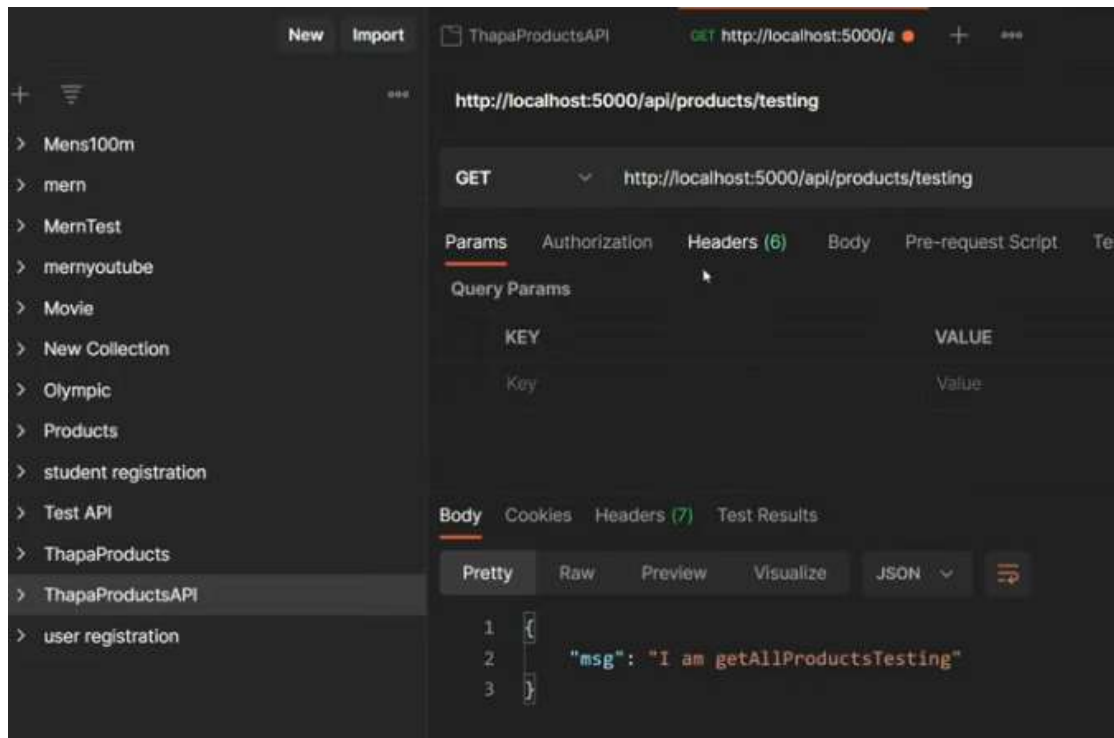
Now Let us see how to use **Postman and ThunderBolt**



Inside **Postman**

Make new collection

Write endpoints and save them



Add MongoDB in our project

Login in MongoDB

Keep all options as default while making cluster

Give name to the cluster

The screenshot shows the 'Cluster Name' form in the MongoDB Atlas interface. The form includes a text input field with the value 'ThapaAPI' and a dropdown menu that also displays 'ThapaAPI'. A note below the input field states: 'One time only: once your cluster is created, you won't be able to change its name.' Below the input field, a warning message reads: 'Cluster names can only contain ASCII letters, numbers, and hyphens.'

Now create User

Save password safely.

Vined's Org ... Access Manager Billing

Project 0 Atlas App Services Charts

DEPLOYMENT

Database

Data Lake **PREVIEW**

DATA SERVICES

Triggers

Data API

Data Federation

SECURITY

Quickstart

Database Access

Network Access

Advanced

New On Atlas 2

privilege by default. You can update these permissions and/or create additional users later. Ensure these credentials are different to your MongoDB Cloud username and password. You can manage existing users via the [Database Access Page](#).

Username

thapa

Password

Autogenerate Secure Password Copy

Create User Success! Please keep your credentials to connect to your cluster.

Username	Authentication Type	
thapa	Password	EDIT REMOVE

2 Where would you like to connect from?

2 Where would you like to connect from?

Enable access for any network(s) that need to read and write data to your cluster.

My Local Environment

Use this to add network IP addresses to the IP Access List. This can be modified at any time.

Cloud Environment

Use this to configure network access between Atlas and your cloud or on-premise environment. Specifically, set up IP Access Lists, Network Peering, and Private Endpoints.

Add entries to your IP Access List

Add your IP address

Add entries to your IP Access List

Only an IP address you add to your Access List will be able to connect to your project's clusters. You can manage existing IP entries via the [Network Access Page](#).

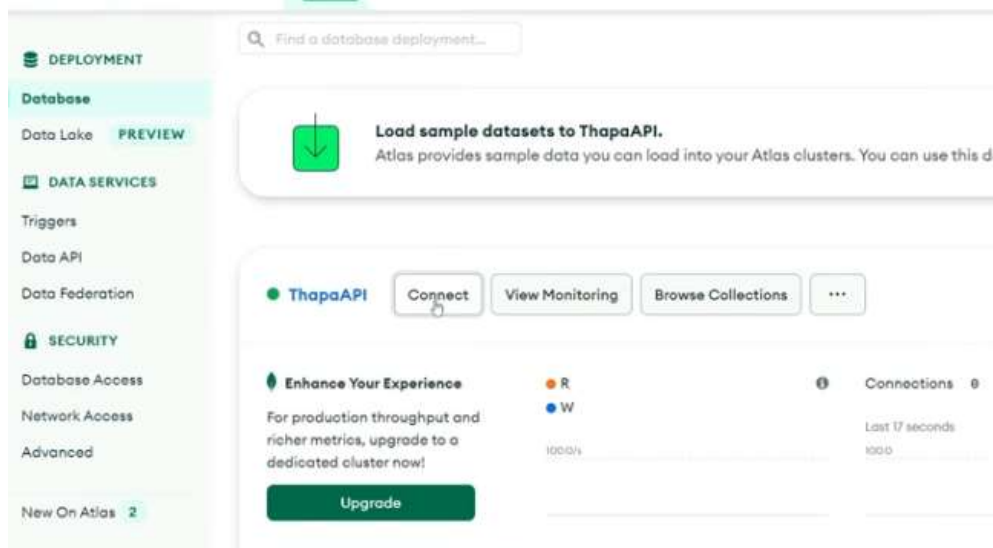
IP Address	Description	
Enter IP Address	Enter description	Add My Current IP Address
Add Entry		

IP Access List	Description	
103.135.203.118/32	My IP Address	EDIT REMOVE

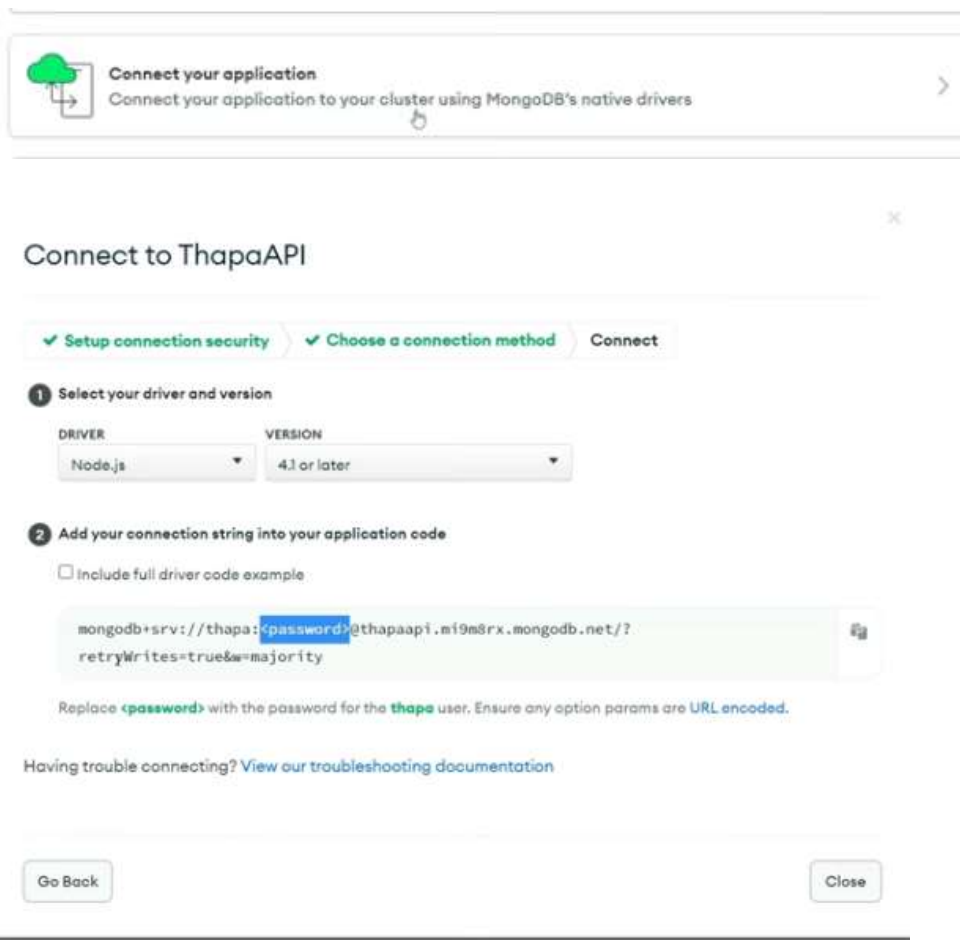
Finish and Close

Go to Network access and Database Access and make sure your have read/write permission in database access and in network access, your connection has the access.

Now we connect the database



Connect to application to connect MongoDB to express



Now add password and Database in the URL

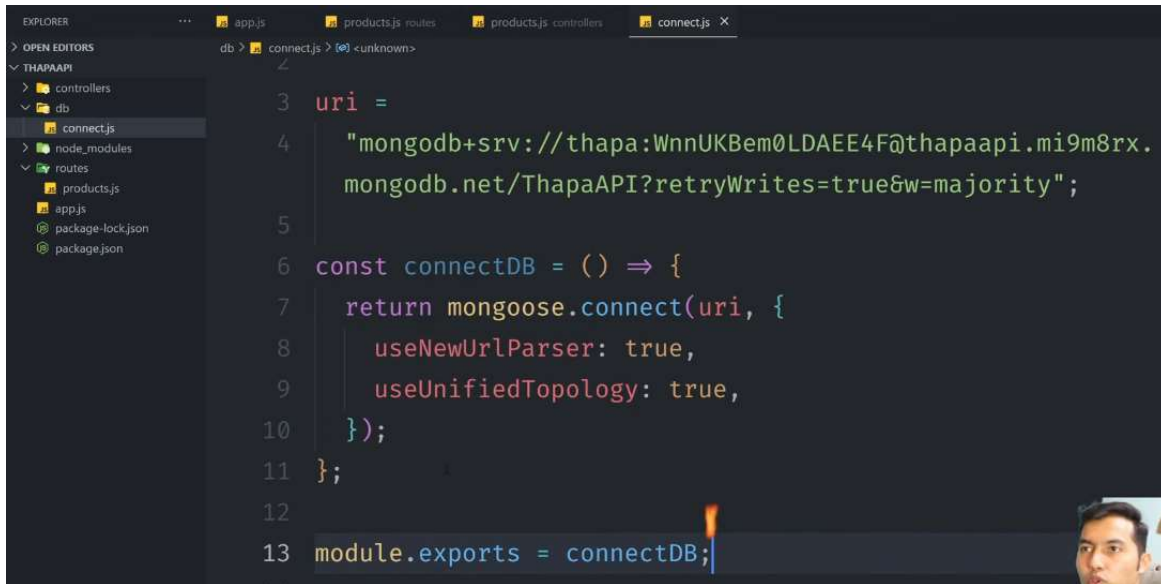
Put DB name between / and ? In URL and write your password in <password>

Now we connect our express app

Make folder db, make connect.js inside db folder

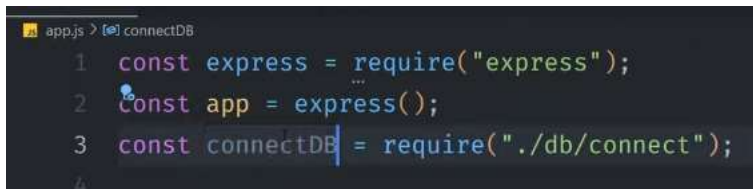
Install mongoose by npm | mongoose

Now inside connect.js



```
3 uri =
4   "mongodb+srv://thapa:WnnUKBem0LDAEE4F@thapaapi.mi9m8rx.
   mongodb.net/ThapaAPI?retryWrites=true&w=majority";
5
6 const connectDB = () => {
7   return mongoose.connect(uri, {
8     useNewUrlParser: true,
9     useUnifiedTopology: true,
10  });
11 };
12
13 module.exports = connectDB;
```

Inside app.js we connect it to the DB



```
1 const express = require("express");
2 const app = express();
3 const connectDB = require("../db/connect");
```

connectDB() returns a promise so we use await

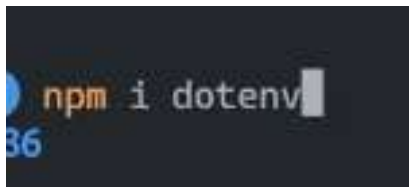


```
const start = async () => {
  try {
    await connectDB();
    app.listen(PORT, () => {
      console.log(`${PORT} Yes I am connected`);
    });
  } catch (error) {
    console.log(error);
  }
}
```

Using ENV file

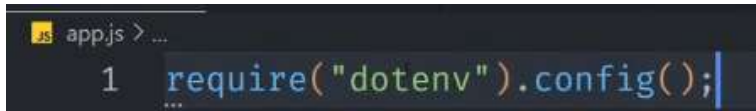
We need to protect our important credentials.

We use .env environment

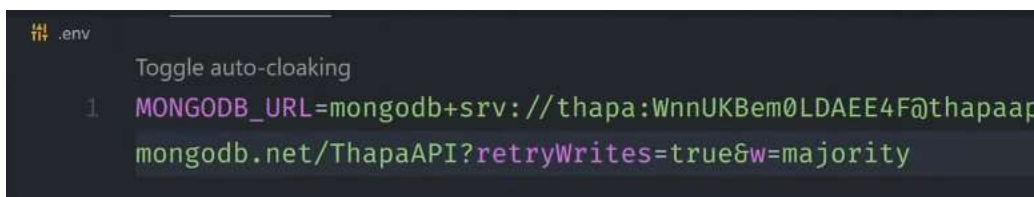
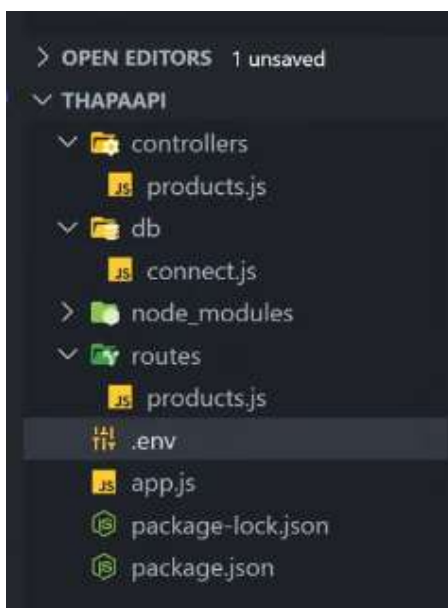


We install dotenv

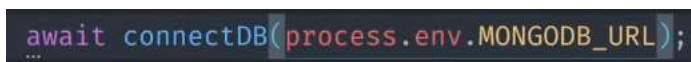
In app.js we require it



Now we make **.env** file



Now we pass URL as parameter from app.js



We do process.env.mongo_URL to get it.

```

db > connect.js > connectDB
1  const mongoose = require("mongoose");
2
3  const connectDB = (uri) => {
4    // console.log("connect db");
5    return mongoose.connect(uri, {
6      useNewUrlParser: true,
7      useUnifiedTopology: true,
8    });
9  };
10
11 module.exports = connectDB;
12

```

Making Collection & Schema, Model

In our API response we have some fields in the document.

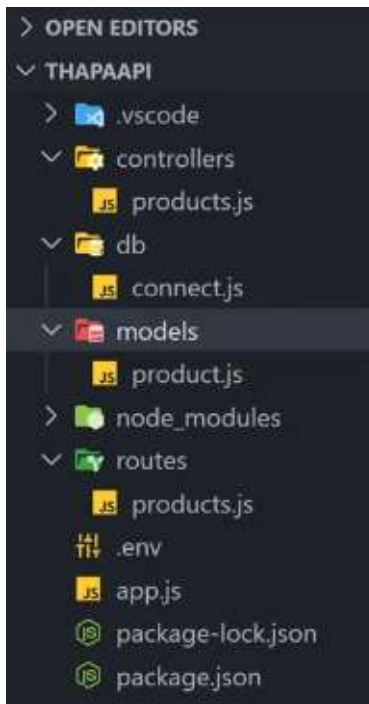
```

{
  "_id": "63761d609a739351d4648bb1",
  "name": "iphone",
  "price": 154,
  "featured": false,
  "rating": 4.8,
  "createdAt": "2022-11-17T11:39:09.640Z",
  "company": "apple",
  "__v": 0
}

```

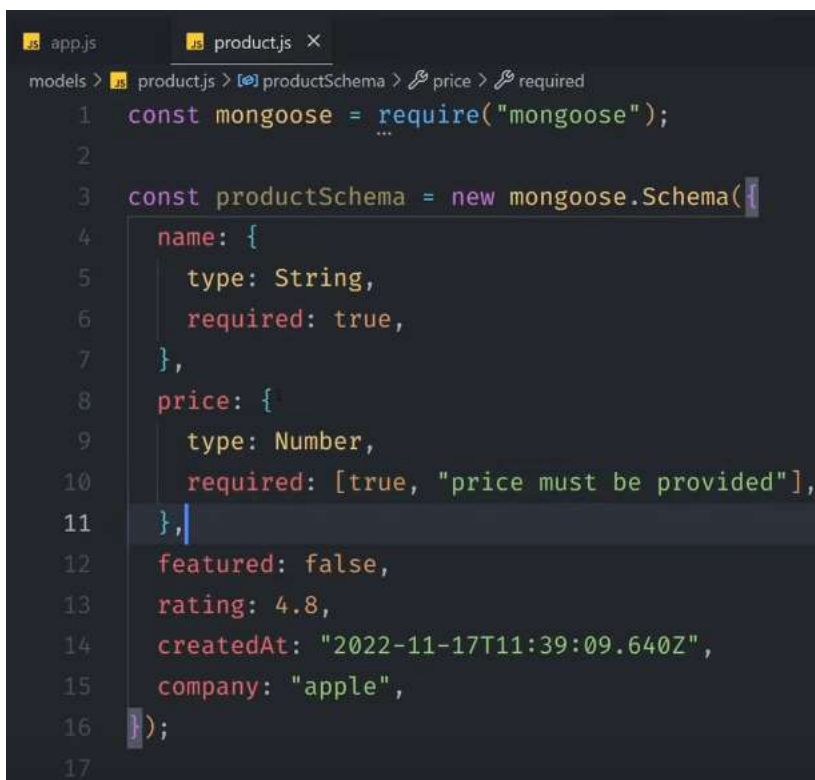
This structure is called model

We make model folder and make products.js inside it



We require mongoose and createSchema using it.

Let us create Schema for product.



Required = true means it's a required field and we can give the message using " "

We can give default value also.

To get the date we do date.now

If we want only specific values for a field we define enum

```

    createdAt: {
      type: Date,
      default: Date.now(),
    },
    company: {
      type: String,
      enum: {
        values: ["apple", "samsung", "dell", "mi"],
        message: `{VALUE} is not supported`,
      },
    },
  },
});

```

Now we create Table

"Product" should always be singular

```

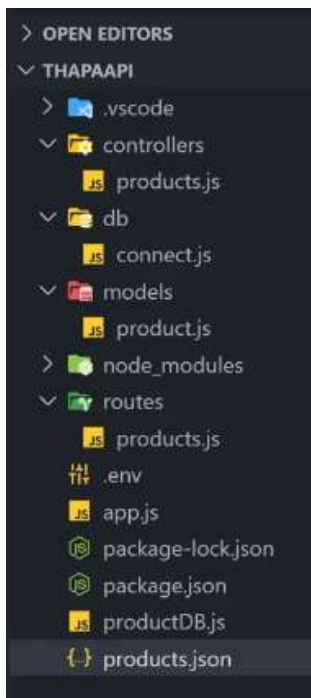
module.exports = mongoose.model("Product", productSchema);

```

Storing JSON file in Database

We create a JSON file products.json and productDB.js

Inside products.json we paste some data



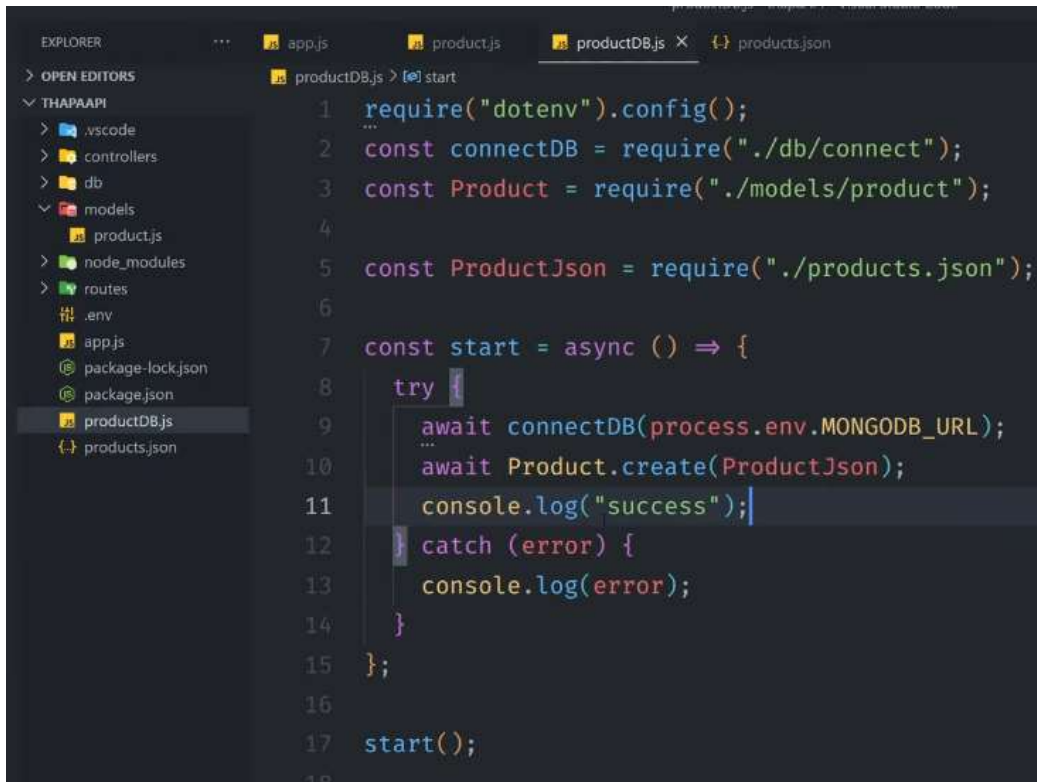
```

[
  {
    "name": "iphone",
    "price": 154,
    "feature": true,
    "company": "apple"
  },
  {
    "name": "iphone10",
    "price": 1154,
    "feature": true,
    "company": "apple"
  },
  {
    "name": "watch",
    "price": 204,
    "company": "apple"
  }
]

```

Now we add this JSON inside our MongoDB in ProductDB.js

We do not need to export JSON files, we can automatically import them using their path.



```

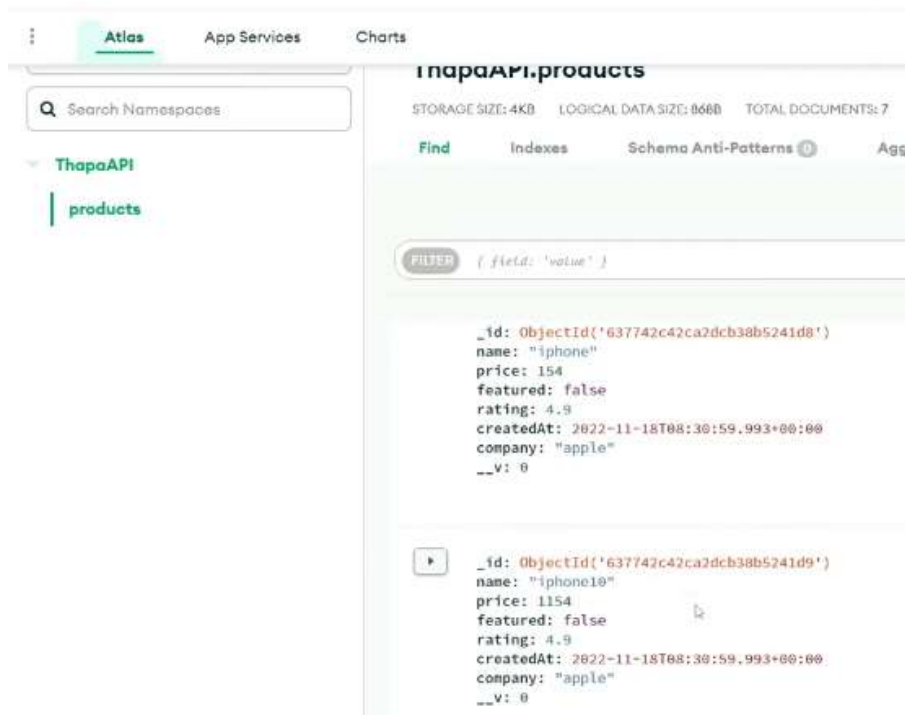
EXPLORER
  THAPAAPI
    .vscode
    controllers
    db
    models
      product.js
    node_modules
    routes
    .env
    app.js
    package-lock.json
    package.json
    productDB.js
    products.json

productDB.js > [0] start
1  require("dotenv").config();
2  const connectDB = require("../db/connect");
3  const Product = require("../models/product");
4
5  const ProductJson = require("../products.json");
6
7  const start = async () => {
8    try {
9      await connectDB(process.env.MONGODB_URL);
10     await Product.create(ProductJson);
11     console.log("success");
12   } catch (error) {
13     console.log(error);
14   }
15 };
16
17 start();

```

Now if we run productDB.js using "npm run productDB.js"

We see our Data gets added to MongoDB



Read Data from Database

We want if user hits "localhost:5000/api/products" we see our JSON data from MongoDB

Inside app.js

We have called our routes at "/api/products"

On "/api/products" we show getAllProducts()

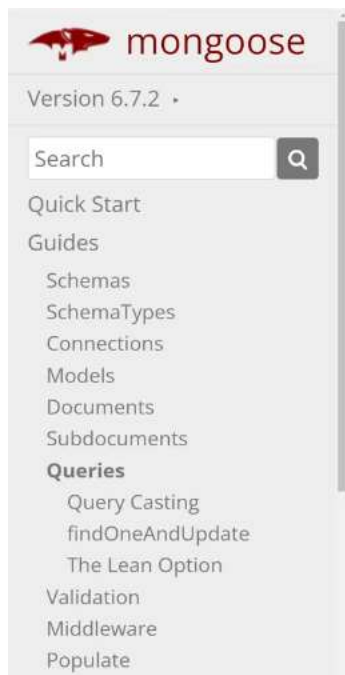
On "/api/products/Testing" we show getAllProductsTesting()

So we somehow need to get our data inside getAllProducts function

We go inside controllers

To do CRUD operations with our Data, we have many methods inside model.

So we import our model first inside controller folder.



Queries

Mongoose [models](#) provide several static helper functions. [mongoose Query object](#).

- `Model.deleteMany()`
- `Model.deleteOne()`
- `Model.find()`
- `Model.findById()`
- `Model.findByIdAndDelete()`
- `Model.findByIdAndRemove()`
- `Model.findByIdAndUpdate()`
- `Model.findOne()`
- `Model.findOneAndDelete()`
- `Model.findOneAndRemove()`
- `Model.findOneAndReplace()`
- `Model.findOneAndUpdate()`
- `Model.replaceOne()`
- `Model.updateMany()`
- `Model.updateOne()`

```
controllers > . products.js > [get] getAllProducts
1  const Product = require("../models/product");
  ...
```

Now we use `find()`

If we give `{ }` inside `find()` we get whole data

```
const getAllProducts = async (req, res) => {
  const myData = await Product.find({});
  res.status(200).json({ myData });
};
```

Output

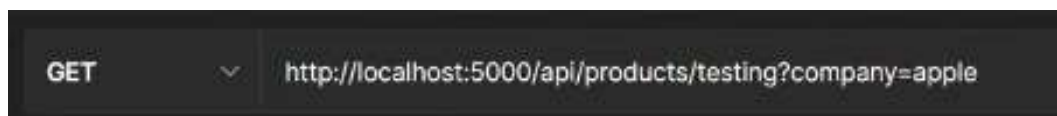


Now if we want only that data which has "iphone"

```
const getAllProducts = async (req, res) => {
  const myData = await Product.find({ name: "iphone" });
  res.status(200).json({ myData });
};
```

Add Filtration & Searching Functionality with Query Props

Now what we want is, if we call below endpoint we want only products of company = apple



Express: req.params, req.query and req.body

These three, req.body, req.query and req.params are part of Express request object.

They are used by the client to send data to the server.

This post outlines their differences and gives examples on how to use them.

1. req.body

Generally used in POST/PUT requests.

Use it when you want to send sensitive data(eg. form data) or super long JSON data to the server.

How to send data in request body

- using axios

```
axios.post('/giraffe', {
  key1: 'value1',
  key2: 'value2'
})
.then(response => {
  ...
})
.catch(error => {
  ...
})
```

How to get data from request body

```
app.get('/giraffe', (req, res) => {
  console.log(req.body.key1) //value1
  console.log(req.body.key2) //value2
})
```

Remember to use express.json() middleware to parse request body else you'll get an error

```
app.use(express.json())
```

2. req.params

These are properties attached to the url i.e named route parameters. You prefix the parameter name with a colon(:) when writing your routes.

For instance,

```
app.get('/giraffe/:number', (req, res) => {  
  console.log(req.params.number)  
})
```

To send the parameter from the client, just replace its name with the value

```
GET http://localhost:3000/giraffe/1
```

3. req.query

req.query is mostly used for searching, sorting, filtering, pagination, e.t.c
Say for instance you want to query an API but only want to get data from page 10, this is what you'd generally use.

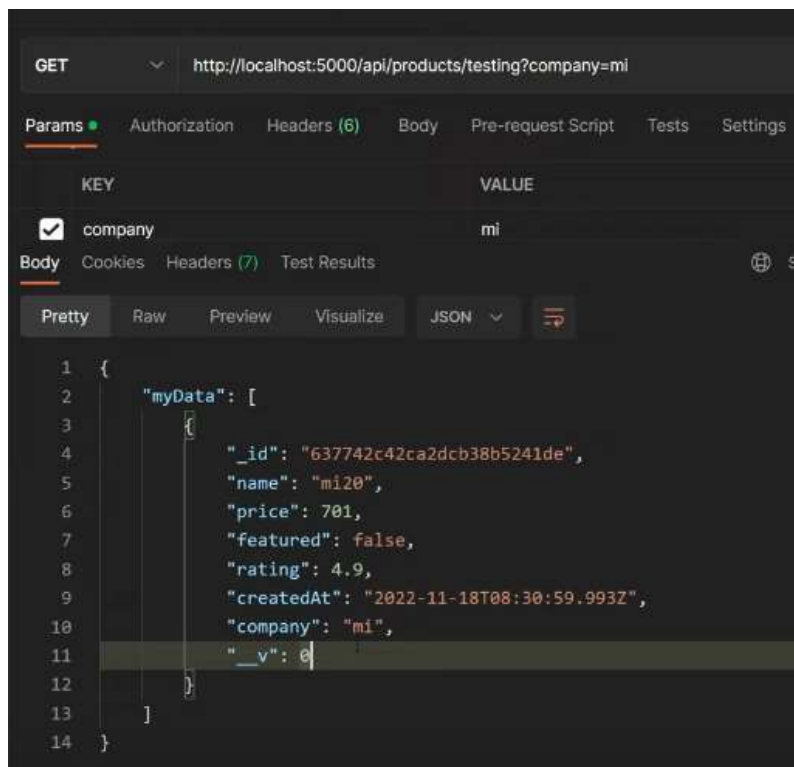
It written as *key=value*

```
GET http://localhost:3000/animals?page=10
```

To access this in your express server is pretty simple too;

```
app.get('/animals', ()=>{  
  console.log(req.query.page) // 10  
})
```

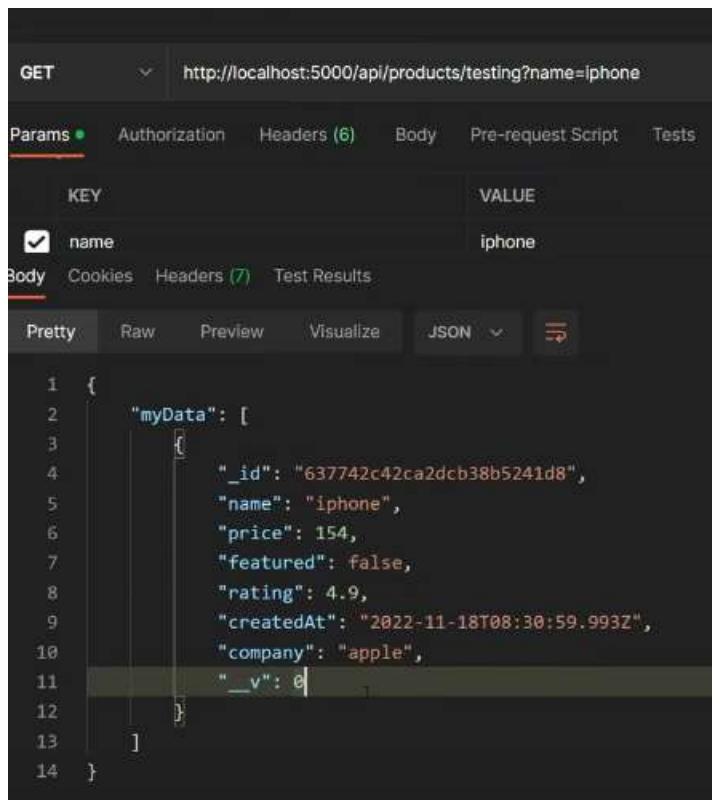
**In endpoint everything after ? Comes under query
Its always in key-value pair.**



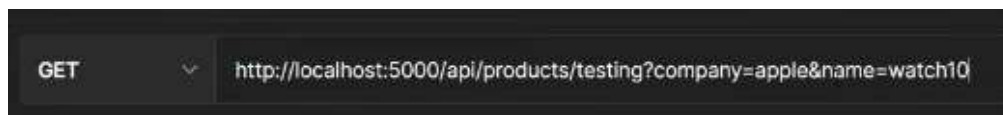
```
const getAllProductsTesting = async (req, res) => {
  const myData = await Product.find(req.query);
  console.log(
    "🔥 ~ file: products.js ~ line 10 ~ getAllProductsTesting ~ req.query",
    req.query
  );

  res.status(200).json({ myData });
};
```

We can put anything inside filter



If there are more than one query, they are separated by &



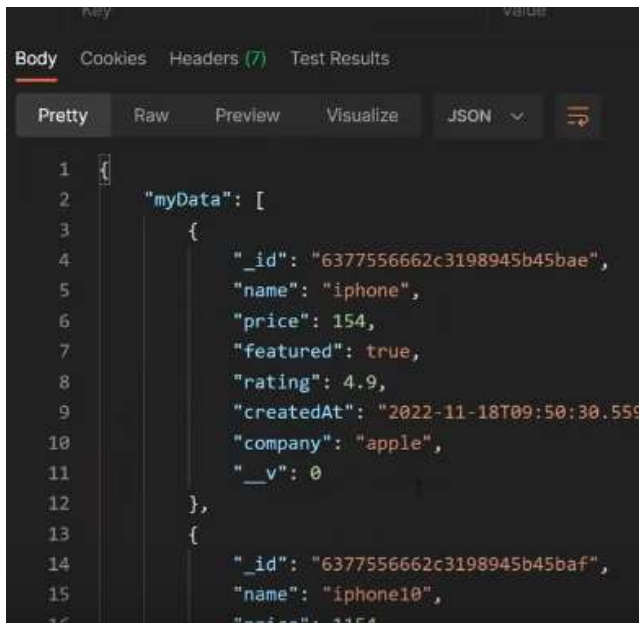
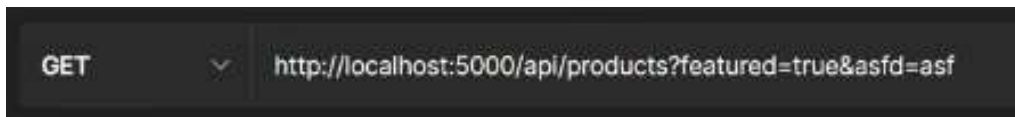
Add Company Filter in API & Make API Work Better

Let say we give 2 things in query, if any one of them is present. Show the result.

Whenever we run our database, it adds the same data again and again so to get away with it. We delete the previous data like using `deleteMany()`

```
const start = async () => {
  try {
    await connectDB(process.env.MONGODB_URL);
    await Product.deleteMany();
    await Product.create(ProductJson);
    console.log("success");
  } catch (error) {
    console.log(error);
  }
};
```

If we write below query inside endpoint and `asfd = asf` is not valid query still we want to show all data where `featured = true`



So if anyone of query is true, show the results.

Inside controller

We take the query paramertes and find It in our DB

```
const getAllProducts = async (req, res) => {
  const { company } = req.query;
  const queryObject = {};

  if (company) {
    queryObject.company = company;
  }

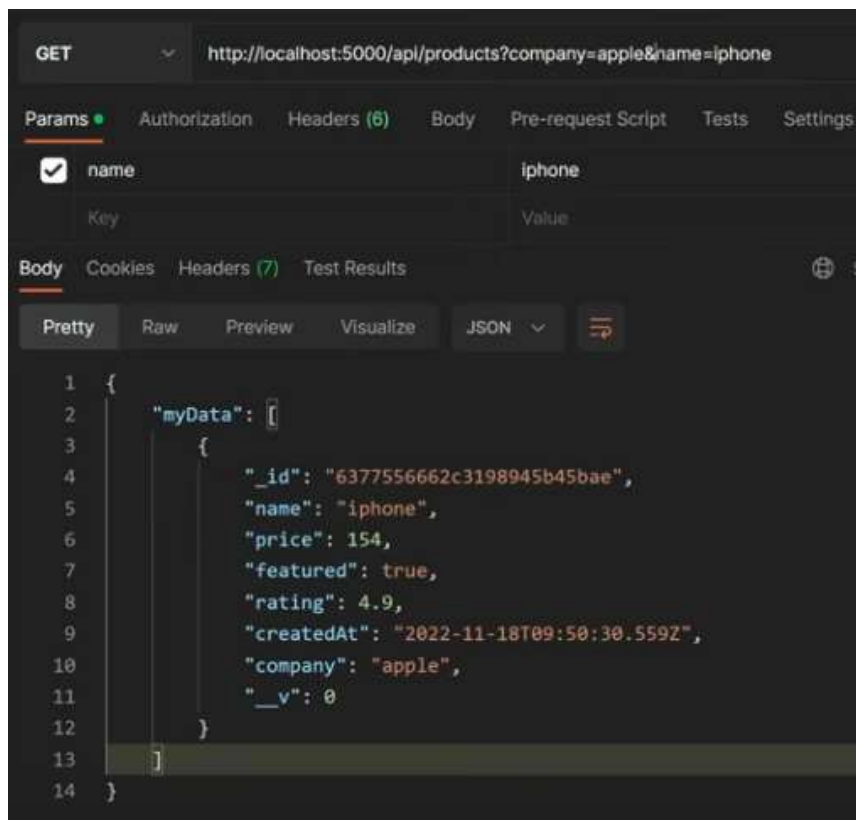
  const myData = await Product.find(queryObject);
  res.status(200).json({ myData });
};
```

Add Advance Search Functionality in our Rest API

Say in the query we give name and company both.

We want to show result matching name and company both.

So we destructure name and company both



```
const getAllProducts = async (req, res) => {
  const { company, name } = req.query;
  const queryObject = {};

  if (company) {
    queryObject.company = company;
  }

  if (name) {
    queryObject.name = name;
  }

  console.log(queryObject);

  const myData = await Product.find(queryObject);
  res.status(200).json({ myData });
};
```

If our query has "name=iphone" it shows only name = iphone
We want to show iphone10 also in the results as it also has iphone in its name
So we want our search to be little elastic

We will use MongoDB regex for this.

\$regex

For data hosted on MongoDB Atlas, MongoDB offers a full-text search solution, MongoDB Atlas Search. If you frequently run case-insensitive regex queries (utilizing the `i` option), MongoDB recommends Atlas Search queries that use the `$search` aggregation pipeline stage.

```
if (name) {  
  queryObject.name = { $regex: name, $options: "i" };  
}
```

Let us also destructure feature

```
const getAllProducts = async (req, res) => {  
  const { company, name, featured } = req.query;  
  const queryObject = {};  
  
  if (company) {  
    queryObject.company = company;  
  }  
  
  if (featured) {  
    queryObject.featured = featured;  
  }  
  
  if (name) {  
    queryObject.name = { $regex: name, $options: "i" };  
  }  
  
  console.log(queryObject);  
  
  const myData = await Product.find(queryObject);  
}
```

Add SORT functionality in Rest API (ASC TO DESC, LOW TO HIGH)

To achieve sorting functionality we use `query.sort` provided by mongoose

Example:

```
// sort by "field" ascending and "test" descending
query.sort({ field: 'asc', test: -1 });

// equivalent
query.sort('field -test');

// also possible is to use a array with array key-value pairs
query.sort([[ 'field', 'asc' ]]);
```

If we want descending we use -1

By default it is Ascending

Take sort from query using req.query

```
const getAllProducts = async (req, res) => {
  const { company, name, featured, sort, select } = req.query;
```

```
const getAllProductsTesting = async (req, res) => {
  const myData = await Product.find(req.query).sort("name");

  res.status(200).json({ myData });
};
```

For Descending

```
const getAllProductsTesting = async (req, res) => {
  const myData = await Product.find(req.query).sort("-name");

  res.status(200).json({ myData });
};
```

Whenever user writes in URL he writes like ?sort=name,price

But In our sort function we want it like sort("name price")

So if we try to get name and price from req.query, we get it like name,price

We need to convert "name,price" to "name price"




```

if (sort) {
  let sortFix = sort.replace(",", " ");
  queryObject.sort = sortFix;
}

console.log(queryObject);

const myData = await Product.find(queryObject);
res.status(200).json({ myData });

```

Now we want to show sorted data only if user has used sort in the query otherwise show normal data. For this we do

```

const getAllProducts = async (req, res) => {

  if (name) {
    queryObject.name = { $regex: name, $options: "i" };
  }

  let apiData = Product.find(queryObject);

  if (sort) {
    let sortFix = sort.replace(",", " ");
    apiData = apiData.sort(sortFix);
  }

  console.log(queryObject);

  const myData = await apiData;
  res.status(200).json({ myData });
};

```

Return Specific Document Fields using SELECT in Mongoose

Query.select is used to tell from all the fields in our schema, which fields we want to show/hide.

Example:

```
// include a and b, exclude other fields
query.select('a b');
// Equivalent syntaxes:
query.select(['a', 'b']);
query.select({ a: 1, b: 1 });

// exclude c and d, include other fields
query.select('-c -d');
```

Take select from query using req.query

```
const getAllProducts = async (req, res) => {
  const { company, name, featured, sort, select } = req.query;
```

To separate 2 queries we use comma ","

```
https://storeapi-production.up.railway.app/api/v1/products?select=name,company

{"Products":[{"_id":"63761d609a739351d4648bb1","name":"iphone","company":"apple"},
{"_id":"63761d609a739351d4648bb2","name":"iphone10","company":"apple"},
{"_id":"63761d609a739351d4648bb6","name":"dell gaming","company":"dell"},
{"_id":"63761d609a739351d4648bb5","name":"s20","company":"samsung"},
{"_id":"63761d609a739351d4648bb7","name":"mi20","company":"mi"},
{"_id":"63761d609a739351d4648bb4","name":"watch10","company":"apple"},
{"_id":"63761d609a739351d4648bb3","name":"watch","company":"apple"}], "nbHits":7}
```

We again need to replace , with " "

As user will write like ?select=name,company

We want query.select(name,company)

So we do split from "," and join using " " [SPACE]

```
// (select = name company;
if (select) {
  // let selectFix = select.replace(", ", " ");
  let selectFix = select.split(", ").join(" ");

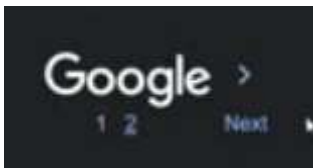
  apiData = apiData.select(selectFix);
}

console.log(queryObject);

const myData = await apiData;
res.status(200).json({ myData });
```

Add Pagination in Rest API using Node & Mongoose

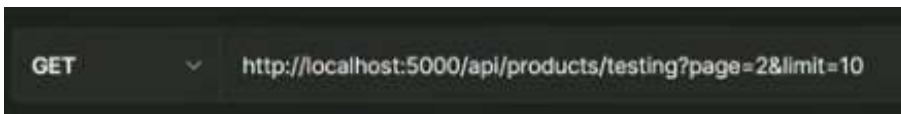
Pagination means



We have a limit of results per page.
Content is fixed, page number are increasing.

Say we have 100 data
We show 10 data per page

In the query we have page and limit to get page number and number of posts in that page.



We get Page and number from req.query. Now we convert it to number
And if page is not given by user, we take by default as 1
And limit = 3

Get page and limit from req.query

```
let page = Number(req.query.page) || 1;
let limit = Number(req.query.limit) || 3;
```

Say our limit = 3 and we have total 8 results.

So we show 3 in one page and next 3 in 2nd page and so on

So we need to pick 3 results and skip other for a particular page so we make a formula for it.

```
let skip = (page - 1) * limit;
```

```
apiData = apiData.skip(skip).limit(limit);
```

Skip(skip) means show me result by skipping first skip results and limit them to 3 only

Skip = 3, limit = 3 means skip first 3 results and show me next 3 results.

Skip = 6, limit = 3 means skip first 6 results and show me next 3 results.

Collect.js skip() Method

Read

Discuss

Courses



The **skip()** method is used to skip the given number of elements from collection and returns the remaining collection elements.

Syntax:

```
collect(array).skip(size)
```

The **limit** method returns the original string if its length is shorter than or equal to the given limit.

Otherwise, the method cuts off the string value and returns a substring with a length of the given character limit.

Host REST API

We can use Railway App.

Push whole code into git first

Configure using github in your Railway app

Choose the github project to deploy

Put environment variables while deploying like MONGO_URL in .env file

Allow Network access from anywhere in MongoDB

Generate domain by going in Settings

Don't push .env file in git

#16: Host REST API Live for Free [🔗](#)

