CONTENTS

TUTORIAL

CRUD Operations with Mongoose and MongoDB Atlas

MongoDB

By Joshua Hall
Published on July 7, 2019 51.2k

RELATED
What is MongoDB? Tutorial
How To Import and Export a MongoDB Database on Ubuntu 20.04 Tutorial

While this tutorial has content that we believe is of great benefit to our community, we have not yet tested or edited it to ensure you have an error-free learning experience. It's on our list, and we're working on it! You can help us out by using the "report an issue" button at the bottom of the tutorial.

Mongoose is one of the fundamental tools for manipulating data for a Node.js/MongoDB backend. In this article we'll be looking into some of the basic ways of using Mongoose and even using it with the MongoDB Atlas remote database.

SCROLL TO TOP

Prerequisites

Since we'll be using `Express` to set up a basic server, I would recommend checking out this article on express and the official Express docs if you're not very comfortable with it just yet.

We'll also be setting up a very basic REST Api without any authentication, you can learn a bit more about that here, but the routing will be asynchronous and we'll be using `async/await` functions, which you can freshen up on here.

Installation

Of course, we'll also need to have `express` installed to set up our server.

```
$ npm i express mongoose
```

File Setup

We'll only need three files and two folders to get started. This folder structure works well for the sake of organization if you want to add more to your API in the future:

```
models 📂
  FoodModel.js
routes 📂
  FoodRoutes.js
server.js
```

MongoDB Atlas Setup

We'll need to get setup with MongoDB Atlas. Here's a summary of the steps to get started:

- Setup an account
- Hit `Build a Cluster`
- Go to Database Access and hit `Add New User`. Add a username and password, if you autogenerate a password make sure you copy it, we'll need it later.
- Go to Network Access, hit `Add IP Address`, and hit `Add Current IP Address`, then confirm.
- Go to `Clusters`, if your cluster build is done then hit `Connect`, `Connect Your Application`, and copy the line of code it gives you

Everything else with MongoDB Atlas will be handled on our end with in Node.js.

Postman Setup

We'll be using Postman to help manage our requests for us. Once you get it downloaded and set up, create a new collection called whatever you want (mine will be called Gator Diner). On the bottom right of the new collection, there are three little dots that give you an 'Add Request' option. Every time we make a new API endpoint we'll be setting up another request for it. This will help you manage everything so you don't have to copy/paste HTTP requests everywhere.

Server Setup

Here we'll set up our basic Express server on port 3000, connect to our Atlas database, and import our future routes. `mongoose.connect()` will take the line of code we copied from Atlas and an object of configuration options (the only option we'll be using is `useNewUrlParser`, which will just get rid of an annoying terminal error we get saying that the default is deprecated, this doesn't change the behavior or the usage).

In the MongoDB Atlas code provided there is a section like `UserName:<password>`, replace with the username and whatever password you set up in Atlas (make sure to remove the greater/less than signs).

server.js

```
const express = require('express');
const mongoose = require('mongoose');
const foodRouter = require('./routes/foodRoutes.js');

const app = express();
app.use(express.json()); // Make sure it comes back as json

mongoose.connect('mongodb+srv://UserName:<password>@cluster0-8vkls.mongodb.net/test?retryWrites=true&w=majo
  useNewUrlParser: true
});

app.use(foodRouter);

app.listen(3000, () => { console.log('Server is running...') });
```

There we go, now we're all hooked up to our backend and we can start actually learning mongoose.

Schemas

First we need to have a pattern to structure our data onto, and these patterns are referred to as schemas. Schemas allow us to decide exactly what data we want, and what options we want the data to have as an object.

With that basic pattern in place we'll use the `mongoose.model` method to make it usable with actual data and export it as a variable we can use in `foodRoutes.js`.

Options

- `type` Sets whether it is a `String`, `Number`, `Date`, `Boolean`, `Array`, or a `Map` (an object).
- `required` (Boolean) Return an error if not provided.
- `trim` (Boolean) Removes any extra whitespace.
- `uppercase` (Boolean) Converts to uppercase.
- `lowercase` (Boolean) Converts to lowercase.
- `validate` Sets a function to determine if the result is acceptable.
- `default` Sets the default if no data is given.

./models/food.js

```
const mongoose = require('mongoose');

const FoodSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    trim: true,
    lowercase: true
  },
  calories: {
    type: Number,
    default: 0,
    validate(value) {
      if (value < 0) throw new Error("Negative calories aren't real.");
    }
  },
});

const Food = mongoose.model("Food", FoodSchema);
module.exports = Food;
```

Query Functions

While there are many querying functions available, which you can find here, here are the ones we'll be using in this instance:

- `find()` Returns all objects with matching parameters so `.find({ name: fish })` would return every object named fish and an empty object will return everything.
- `save()` Save it to our Atlas database.
- `findByIdAndDelete()` Takes the objects id and removes from the database.
- `findByIdAndUpdate` Takes the objects id and an object to replace it with.
- `deleteOne()` and `deleteMany()` Removes the first or all items from the database.

Read All

Once we have our data model set up we can start setting up basic routes to use it.

We'll start by getting all foods in the database, which should just be an empty array right now. Since Mongoose functions are asynchronous, we'll be using `async/await`.

Once we have the data we'll use a try/catch block to send it back to and so that we can see that things are working using Postman.

./routes/foodRoutes.js

```
const express = require('express');
const foodModel = require('../models/food');
const app = express();

app.get('/foods', async (req, res) => {
  const foods = await foodModel.find({});

  try {
    res.send(foods);
  } catch (err) {
    res.status(500).send(err);
  }
});

module.exports = app
```

In Postman, we'll create a new Get All request, set the url to `localhost:3000/foods`, and hit send. It should make our HTTP GET request for us and return our empty array.

Create

This time we'll be setting up a post request to `'/food'` and we'll create a new food object with our model and pass it the request data from Postman.

Once we have a model with data in place, we can use `.save()` to save it to our database.

```
app.post('/food', async (req, res) => {
  const food = new foodModel(req.body);

  try {
    await food.save();
    res.send(food);
  } catch (err) {
    res.status(500).send(err);
  }
});
```

In Postman we'll make another request called Add New, give it the URL like before to `localhost:3000/food`, and change it from a GET to a POST request.

Over in `Body`, select raw and JSON (application/json) from the drop down. Here we can add whatever new food item we want to save to our database.

```
{
  "name": "snails",
  "calories": "100"
}
```

If you run the Get All request again, our once empty array should now contain this object.

Delete

Every object created with Mongoose is given its own `_id`, and we can use this to target specific items with a `DELETE` request. We can use the `.findByIdAndDelete()` method to easily remove it from the database, it obviously just needs the id, which is stored on our request object.

```
app.delete('/food/:id', async (req, res) => {
  try {
    const food = await foodModel.findByIdAndDelete(req.params.id)

    if (!food) res.status(404).send("No item found")
    res.status(200).send()
  } catch (err) {
    res.status(500).send(err)
  }
})
```

In Postman we can use the Get All request to get the id of one of our items, and add that to the end of our url in our new Remove request, it'll look something like `localhost:3000/food/5d1f6c3e4b0b88fb1d257237`.

Run Get All again and it should be gone.

Update

Just like with the delete request, we'll be using the `_id` to target the correct item. `.findByIdAndUpdate()` just takes the target's id, and the request data you want to replace it with.

```
app.patch('/food/:id', async (req, res) => {
  try {
    await foodModel.findByIdAndUpdate(req.params.id, req.body)
    await foodModel.save()
    res.send(food)
  } catch (err) {
    res.status(500).send(err)
  }
})
```

Conclusion

With Mongoose methods we can quickly make and manage our backend data in a breeze. While there's still much more to learn, hopefully this little introduction will help give you a brief insight into Mongoose and help you decide if it's the right tool for you and your future projects.

Was this helpful? Yes No

Report an issue

About the authors

Joshua Hall
is a Community author on DigitalOcean.

Still looking for an answer?

Ask a question

Search for more help

Comments

2 Comments

Leave a comment...

Sign In to Comment

LukeAvedon September 11, 2020
1
Great tutorial!

Think I may have found one small error:
In "File Setup" you refer to the file ./models/FoodModel.js but later refer to the same file as ./models/food.js. Perhaps, I missed something obvious? Thanks for the very helpful tutorial.

Reply Report

Yacie November 21, 2020
0
Gooday sir
Do you have a git repo for this.
I wud much appreciate if you have it hosted too
Or u want me to figure it out too

The challenge I been having with these tutorials is transitioning from 'localhost' to remote host. That's why I asked you the above questions

Reply Report