codeSTACKr
Posted on Apr 19, 2022 • Originally published at mongodb.com

Getting Started with MongoDB & Mongoose

#beginners #database #mongodb #javascript

In this article, we'll learn how Mongoose, a third-party library for MongoDB, can help you to structure and access your data with ease.

What is Mongoose?

Many who learn MongoDB get introduced to it through the very popular library, Mongoose. Mongoose is described as "elegant MongoDB object modeling for Node.js."

Mongoose is an ODM (Object Data Modeling) library for MongoDB. While you don't need to use an Object Data Modeling (ODM) or Object Relational Mapping (ORM) tool to have a great experience with MongoDB, some developers prefer them. Many Node.js developers choose to work with Mongoose to help with data modeling, schema enforcement, model validation, and general data manipulation. And Mongoose makes these tasks effortless.

> If you want to hear from the maintainer of Mongoose, Val Karpov, give this episode of the MongoDB Podcast a listen!

Why Mongoose?

By default, MongoDB has a flexible data model. This makes MongoDB databases very easy to alter and update in the future. But a lot of developers are accustomed to having rigid schemas.

Mongoose forces a semi-rigid schema from the beginning. With Mongoose, developers must define a Schema and Model.

What is a schema?

A schema defines the structure of your collection documents. A Mongoose schema maps directly to a MongoDB collection.

```
const blog = new Schema({
  title: String,
  slug: String,
  published: Boolean,
  author: String,
  content: String,
  tags: [String],
  createdAt: Date,
  updatedAt: Date,
  comments: [{
    user: String,
    content: String,
    votes: Number
  }]
});
```

With schemas, we define each field and its data type. Permitted types are:

- String
- Number
- Date
- Buffer
- Boolean
- Mixed
- ObjectId
- Array
- Decimal128
- Map

What is a model?

Models take your schema and apply it to each document in its collection.

Models are responsible for all document interactions like creating, reading, updating, and deleting (CRUD).

> An important note: the first argument passed to the model should be the singular form of your collection name. Mongoose automatically changes this to the plural form, transforms it to lowercase, and uses that for the database collection name.

```
const Blog = mongoose.model('Blog', blog);
```

In this example, `Blog` translates to the `blogs` collection.

Environment setup

Let's set up our environment. I'm going to assume you have Node.js installed already.

We'll run the following commands from the terminal to get going:

```
mkdir mongodb-mongoose
cd mongodb-mongoose
npm init -y
npm i mongoose
npm i -D nodemon
code .
```

This will create the project directory, initialize, install the packages we need, and open the project in VS Code.

Let's add a script to our `package.json` file to run our project. We will also use ES Modules instead of Common JS, so we'll add the module `type` as well. This will also allow us to use top-level `await`.

```
...
  "scripts": {
    "dev": "nodemon index.js"
  },
  "type": "module",
...
```

Connecting to MongoDB

Now we'll create the `index.js` file and use Mongoose to connect to MongoDB.

```
import mongoose from 'mongoose'

mongoose.connect("mongodb+srv://<username>:<password>@cluster0.eyhty.mongodb.net/myFir
```

You could connect to a local MongoDB instance, but for this article we are going to use a free MongoDB Atlas cluster. If you don't already have an account, it's easy to sign up for a free MongoDB Atlas cluster here.

And if you don't already have a cluster set up, follow our guide to get your cluster created.

After creating your cluster, you should replace the connection string above with your connection string including your username and password.

> The connection string that you copy from the MongoDB Atlas dashboard will reference the `myFirstDatabase` database. Change that to whatever you would like to call your database.

Creating a schema and model

Before we do anything with our connection, we'll need to create a schema and model.

Ideally, you would create a schema/model file for each schema that is needed. So we'll create a new folder/file structure: `model/Blog.js`.

```
import mongoose from 'mongoose';
const { Schema, model } = mongoose;

const blogSchema = new Schema({
  title: String,
  slug: String,
  published: Boolean,
  author: String,
  content: String,
  tags: [String],
  createdAt: Date,
  updatedAt: Date,
  comments: [{
    user: String,
    content: String,
    votes: Number
  }]
});
```

```
const Blog = model('Blog', blogSchema);
export default Blog;
```

Inserting data // method 1

Now that we have our first model and schema set up, we can start inserting data into our database.

Back in the `index.js` file, let's insert a new blog article.

```
import mongoose from 'mongoose';
import Blog from './model/Blog';

mongoose.connect("mongodb+srv://mongo:mongo@cluster0.eyhty.mongodb.net/myFirstDatabase

// Create a new blog post object
const article = new Blog({
  title: 'Awesome Post!',
  slug: 'awesome-post',
  published: true,
  content: 'This is the best post ever',
  tags: ['featured', 'announcement'],
});

// Insert the article in our MongoDB database
await article.save();
```

We first need to import the `Blog` model that we created. Next, we create a new blog object and then use the `save()` method to insert it into our MongoDB database.

Let's add a bit more after that to log what is currently in the database. We'll use the `findOne()` method for this.

```
// Find a single blog post
const firstArticle = await Blog.findOne({});
console.log(firstArticle);
```

Let's run the code!

```
npm run dev
```

You should see the document inserted logged in your terminal.

> Because we are using `nodemon` in this project, every time you save a file, the code will run again. If you want to insert a bunch of articles, just keep saving. 😄

Inserting data // method 2

In the previous example, we used the `save()` Mongoose method to insert the document into our database. This requires two actions: instantiating the object, and then saving it.

Alternatively, we can do this in one action using the Mongoose `create()` method.

```
// Create a new blog post and insert into database
const article = await Blog.create({
  title: 'Awesome Post!',
  slug: 'awesome-post',
  published: true,
  content: 'This is the best post ever',
  tags: ['featured', 'announcement'],
});

console.log(article);
```

This method is much better! Not only can we insert our document, but we also get returned the document along with its `_id` when we console log it.

Update data

Mongoose makes updating data very convenient too. Expanding on the previous example, let's change the `title` of our article.

```
article.title = "The Most Awesomest Post!!";
await article.save();
console.log(article);
```

We can directly edit the local object, and then use the `save()` method to write the update back to the database. I don't think it can get much easier than that!

Finding data

Let's make sure we are updating the correct document. We'll use a special Mongoose method, `findById()`, to get our document by its ObjectId.

```
const article = await Blog.findById("62472b6ce09e8b77266d6b1b").exec();
console.log(article);
```

> Notice that we use the `exec()` Mongoose function. This is technically optional and returns a promise. In my experience, it's better to use this function since it will prevent some head-scratching issues. If you want to read more about it, check out this note in the Mongoose docs about promises.

There are many query options in Mongoose. View the full list of queries.

Projecting document fields

Just like with the standard MongoDB Node.js driver, we can project only the fields that we need. Let's only get the `title`, `slug`, and `content` fields.

```
const article = await Blog.findById("62472b6ce09e8b77266d6b1b", "title slug content")
console.log(article);
```

The second parameter can be of type `Object|String|Array<String>` to specify which fields we would like to project. In this case, we used a `String`.

Deleting data

Just like in the standard MongoDB Node.js driver, we have the `deleteOne()` and `deleteMany()` methods.

```
const blog = await Blog.deleteOne({ author: "Jesse Hall" })
console.log(blog)

const blog = await Blog.deleteMany({ author: "Jesse Hall" })
console.log(blog)
```

Validation

Notice that the documents we have inserted so far have not contained an `author`, dates, or `comments`. So far, we have defined what the structure of our document should look like, but we have not defined which fields are actually required. At this point any field can be omitted.

Let's set some required fields in our `Blog.js` schema.

```
const blogSchema = new Schema({
  title:  {
    type: String,
    required: true,
  },
```

```
  slug:  {
    type: String,
    required: true,
    lowercase: true,
  },
  published: {
    type: Boolean,
    default: false,
  },
  author: {
    type: String,
    required: true,
  },
  content: String,
  tags: [String],
  createdAt: {
    type: Date,
    default: () => Date.now(),
    immutable: true,
  },
  updatedAt: Date,
  comments: [{
    user: String,
    content: String,
    votes: Number
  }]
});
```

When including validation on a field, we pass an object as its value.

> `value: String` is the same as `value: {type: String}`.

There are several validation methods that can be used.

We can set `required` to true on any fields we would like to be required.

For the `slug`, we want the string to always be in lowercase. For this, we can set `lowercase` to true. This will take the slug input and convert it to lowercase before saving the document to the database.

For our `created` date, we can set the default buy using an arrow function. We also want this date to be impossible to change later. We can do that by setting `immutable` to true.

> Validators only run on the create or save methods.

Other useful methods

Mongoose uses many standard MongoDB methods plus introduces many extra helper methods that are abstracted from regular MongoDB methods. Next, we'll go over just a few of them.

`exists()`

The `exists()` method returns either `null` or the ObjectId of a document that matches the provided query.

```
const blog = await Blog.exists({ author: "Jesse Hall" })
console.log(blog)
```

`where()`

Mongoose also has its own style of querying data. The `where()` method allows us to chain and build queries.

```
// Instead of using a standard find method
const blogFind = await Blog.findOne({ author: "Jesse Hall" });

// Use the equivalent where() method
const blogWhere = await Blog.where("author").equals("Jesse Hall");
console.log(blogWhere)
```

Either of these methods work. Use whichever seems more natural to you.

You can also chain multiple `where()` methods to include even the most complicated query.

`select()`

To include projection when using the `where()` method, chain the `select()` method after your query.

```
const blog = await Blog.where("author").equals("Jesse Hall").select("title author")
console.log(blog)
```

Multiple schemas

It's important to understand your options when modeling data.

If you're coming from a relational database background, you'll be used to having separate tables for all of your related data.

Generally, in MongoDB, data that is accessed together should be stored together.

You should plan this out ahead of time if possible. Nest data within the same schema when it makes sense.

If you have the need for separate schemas, Mongoose makes it a breeze.

Let's create another schema so that we can see how multiple schemas can be used together.

We'll create a new file, `User.js`, in the model folder.

```
import mongoose from 'mongoose';
const {Schema, model} = mongoose;

const userSchema = new Schema({
  name: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    minLength: 10,
    required: true,
    lowercase: true
  },
});

const User = model('User', userSchema);
export default User;
```

For the `email`, we are using a new property, `minLength`, to require a minimum character length for this string.

Now we'll reference this new user model in our blog schema for the `author` and `comments.user`.

```
import mongoose from 'mongoose';
const { Schema, SchemaTypes, model } = mongoose;

const blogSchema = new Schema({
  ...,
  author: {
    type: SchemaTypes.ObjectId,
    ref: 'User',
```

```
    required: true,
  },
  ...,
  comments: [{
    user: {
      type: SchemaTypes.ObjectId,
      ref: 'User',
      required: true,
    },
    content: String,
    votes: Number
  }];
});
...
```

Here, we set the `author` and `comments.user` to `SchemaTypes.ObjectId` and added a `ref`, or reference, to the user model.

This will allow us to "join" our data a bit later.

And don't forget to destructure `SchemaTypes` from `mongoose` at the top of the file.

Lastly, let's update the `index.js` file. We'll need to import our new user model, create a new user, and create a new article with the new user's `_id`.

```
...
import User from './model/User.js';

...

const user = await User.create({
  name: 'Jesse Hall',
  email: 'jesse@email.com',
});

const article = await Blog.create({
  title: 'Awesome Post!',
  slug: 'Awesome-Post',
  author: user._id,
  content: 'This is the best post ever',
  tags: ['featured', 'announcement'],
});

console.log(article);
```

Notice now that there is a `users` collection along with the `blogs` collection in the MongoDB database.

You'll now see only the user `_id` in the author field. So, how do we get all of the info for the author along with the article?

We can use the `populate()` Mongoose method.

```
const article = await Blog.findOne({ title: "Awesome Post!" }).populate("author");
console.log(article);
```

Now the data for the `author` is populated, or "joined," into the `article` data. Mongoose actually uses the MongoDB `$lookup` method behind the scenes.

Middleware

In Mongoose, middleware are functions that run before and/or during the execution of asynchronous functions at the schema level.

Here's an example. Let's update the `updated` date every time an article is saved or updated. We'll add this to our `Blog.js` model.

```
blogSchema.pre('save', function(next) {
  this.updated = Date.now(); // update the date every time a blog post is saved
  next();
});
```

Then in the `index.js` file, we'll find an article, update the title, and then save it.

```
const article = await Blog.findById("6247589060c9b6abfa1ef530").exec();
article.title = "Updated Title";
await article.save();
console.log(article);
```

Notice that we now have an `updated` date!

Besides `pre()`, there is also a `post()` mongoose middleware function.

Next steps

I think our example here could use another schema for the `comments`. Try creating that schema and testing it by adding a few users and comments.

There are many other great Mongoose helper methods that are not covered here. Be sure to check out the official documentation for references and more examples.

Conclusion

I think it's great that developers have many options for connecting and manipulating data in MongoDB. Whether you prefer Mongoose or the standard MongoDB drivers, in the end, it's all about the data and what's best for your application and use case.

I can see why Mongoose appeals to many developers and I think I'll use it more in the future.

Thanks for reading!

Say Hello! YouTube | Twitter | Instagram

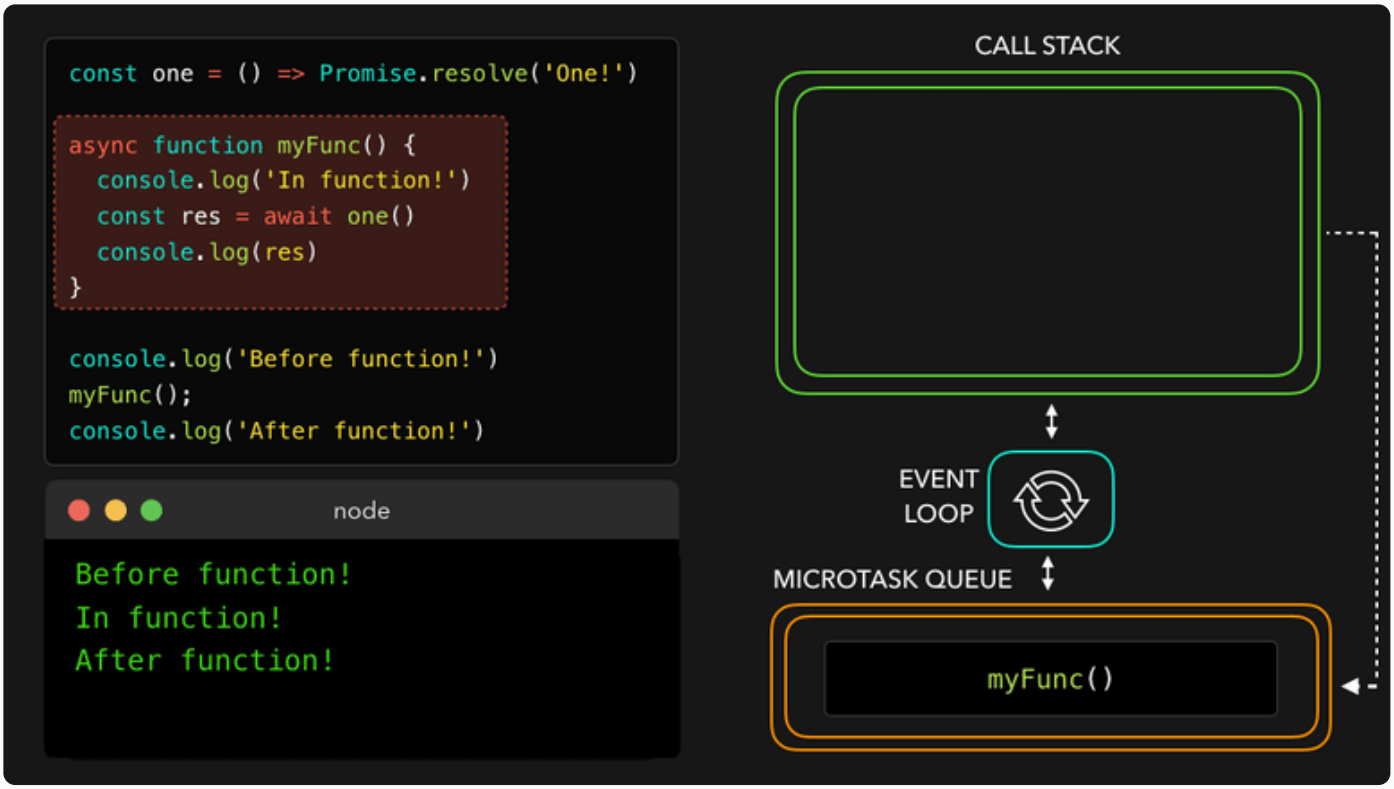
Top comments (1)

Ibrahim Raimi • May 5 '22 • Edited on

This was really helpful, thank you

Code of Conduct • Report abuse

Visualizing Promises and Async/Await 😲



☝ Check out this all-time classic DEV post

codeSTACKr

I create Tutorials for all things CODE.

LOCATION
Texas, US

EDUCATION
self

WORK
Senior Developer Advocate @MongoDB / Full-Stack Developer / Instructor

JOINED
Apr 8, 2020

More from codeSTACKr

Web Development Roadmap 2023
#webdev #beginners #programming #career

Microinteractions: Password Validation Animation
#webdev #beginners #tutorial #ux

Notion + YouTube - A Powerful Combination for Productivity

#productivity #webdev #beginners #tutorial