An Introduction to Mongoose for MongoDB and Node.js

by Jamie Munro 9 Oct 2017

Length: Long Languages: English

Node.js NoSQL MongoDB

This post is part of a series called An Introduction to Mongoose for MongoDB and Node.js.

Bulk Import a CSV File Into MongoDB Using Mongoose With Node.js

Mongoose is a JavaScript framework that is commonly used in a Node.js application with a MongoDB database. In this article, I am going to introduce you to Mongoose and MongoDB, and more importantly where these technologies fit in to your application.

What Is MongoDB?

Let's start with MongoDB. MongoDB is a database that stores your data as documents. Most commonly these documents resemble a JSON-like structure:

```
{
  firstName: "Jamie",
```

```
  lastName: "Munro"
}
```

A document then is placed within a collection. As an example, the above document example defines a `user` object. This `user` object then would typically be part of a collection called `users`.

One of the key factors with MongoDB is its flexibility when it comes to structure. Even though in the first example, the `user` object contained a `firstName` and `lastName` property, these properties are not required in every `user` document that is part of the `users` collection. This is what makes MongoDB very different from a SQL database like MySQL or Microsoft SQL Server that requires a strongly-defined database schema of each object it stores.

The ability to create dynamic objects that are stored as documents in the database is where Mongoose comes into play.

What Is Mongoose?

Mongoose is an Object Document Mapper (ODM). This means that Mongoose allows you to define objects with a strongly-typed schema that is mapped to a MongoDB document.

Mongoose provides an incredible amount of functionality around creating and working with schemas. Mongoose currently contains eight SchemaTypes that a property is saved as when it is persisted to MongoDB. They are:

1. String
2. Number
3. Date
4. Buffer
5. Boolean
6. Mixed
7. ObjectId
8. Array

Each data type allows you to specify:

- a default value
- a custom validation function
- indicate a field is required
- a get function that allows you to manipulate the data before it is returned as an object
- a set function that allows you to manipulate the data before it is saved to the database
- create indexes to allow data to be fetched faster

Further to these common options, certain data types allow you to further customize how the data is stored and retrieved from the database. For example, a `String` data type also allows you to specify the following additional options:

- convert it to lowercase
- convert it to uppercase
- trim data prior to saving
- a regular expression that can limit data allowed to be saved during the validation process
- an enum that can define a list of strings that are valid

The `Number` and `Date` properties both support specifying a minimum and maximum value that is allowed for that field.

Most of the eight allowed data types should be quite familiar to you. However, there are several exceptions that may jump out to you, such as `Buffer`, `Mixed`, `ObjectId`, and `Array`.

The `Buffer` data type allows you to save binary data. A common example of binary data would be an image or an encoded file, such as a PDF document.

The `Mixed` data type turns the property into an "anything goes" field. This field resembles how many developers may use MongoDB because there is no defined structure. Be wary of using this data type as it loses many of the great features that Mongoose provides,

such as data validation and detecting entity changes to automatically know to update the property when saving.

The `ObjectId` data type commonly specifies a link to another document in your database. For example, if you had a collection of books and authors, the book document might contain an `ObjectId` property that refers to the specific author of the document.

The `Array` data type allows you to store JavaScript-like arrays. With an Array data type, you can perform common JavaScript array operations on them, such as push, pop, shift, slice, etc.

Quick Recap

Before moving on and generating some code, I just wanted to recap what we just learned. MongoDB is a database that allows you to store documents with a dynamic structure. These documents are saved inside a collection.

Mongoose is a JavaScript library that allows you to define schemas with strongly typed data. Once a schema is defined, Mongoose lets you create a Model based on a specific schema. A Mongoose Model is then mapped to a MongoDB Document via the Model's schema definition.

Once you have defined your schemas and models, Mongoose contains many different functions that allow you to validate, save, delete, and query your data using common MongoDB functions. I'll talk about this more with the concrete code examples to follow.

Installing MongoDB

Before we can begin creating our Mongoose schemas and models, MongoDB must be installed and configured. I would suggest visiting MongoDB's Download page. There are several different options available to install. I have linked to the Community Server. This allows you to install a version specific to your operating system. MongoDB also offers an Enterprise Server and a cloud support installation. Since entire books could be written on

installing, tuning, and monitoring MongoDB, I am going to stick with the Community Server.

Once you've downloaded and installed MongoDB for your operating system of choice, you will need to start the database. Rather than reinventing the wheel, I would suggest visiting MongoDB's documentation on how to install the MongoDB Community Edition.

I'll wait here while you configure MongoDB. When you're ready, we can move on to setting up Mongoose to connect to your newly installed MongoDB database.

Setting Up Mongoose

Mongoose is a JavaScript framework, and I am going to use it in a Node.js application. If you already have Node.js installed, you can move on to the next step. If you do not have Node.js installed, I suggest you begin by visiting the Node.js Download page and selecting the installer for your operating system.

With Node.js set up and ready to go, I am going to create a new application and then install the Mongoose NPM Package.

With a command prompt that is set to where you wish your application to be installed, you can run the following commands:

```
mkdir mongoose_basics
cd mongoose_basics
npm init
```

For the initialization of my application, I left everything as their default values. Now I'm going to install the mongoose package as follows:

```
npm install mongoose --save
```

With all the prerequisites configured, let's connect to a MongoDB database. I've placed the following code inside an index.js file because I chose that as the starting point for my application:

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

mongoose.connect('mongodb://localhost/mongoose_basics');
```

The first line of code includes the `mongoose` library. Next, I open a connection to a database that I've called `mongoose_basics` using the `connect` function.

The `connect` function accepts two other optional parameters. The second parameter is an object of options where you can define things like the username and password, if required. The third parameter, which can also be the second parameter if you have no options, is the callback function after attempting to connect. The callback function can be used in one of two ways:

```
mongoose.connect(uri, options, function(error) {

// Check error in initial connection. There is no 2nd param to the callback.

});

// Or using promises

mongoose.connect(uri, options).then(

() => { /** ready to use. The `mongoose.connect()` promise resolves to undefined. */ },

err => { /** handle initial connection error */ }

);
```

To avoid a potential introduction to JavaScript Promises, I will use the first way. Below is an updated index.js file:

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

mongoose.connect('mongodb://localhost/mongoose_basics', function (err) {

   if (err) throw err;

   console.log('Successfully connected');

});
```

If an error occurs when connecting to the database, the exception is thrown and all further processing is stopped. When no error occurs, I have logged a success message to the console.

Mongoose is now set up and connected to a database called `mongoose_basics`. My MongoDB connection is using no username, password, or custom port. If you need to set these options or any other option during connection, I suggest reviewing the Mongoose Documentation on connecting. The documentation provides detailed explanations of the many options available as well as how to create multiple connections, connection pooling, replicas, etc.

With a successful connection, let's move on to define a Mongoose Schema.

Defining a Mongoose Schema

During the introduction, I showed a `user` object that contained two properties: `firstName` and `lastName`. In the following example, I've translated that document into a Mongoose Schema:

```
var userSchema = mongoose.Schema({
    firstName: String,
    lastName: String
});
```

This is a very basic Schema that just contains two properties with no attributes associated with it. Let's expand upon this example by converting the first and last name properties to be child objects of a `name` property. The `name` property will comprise both the first and last name. I'll also add a `created` property that is of type `Date`.

```
var userSchema = mongoose.Schema({
    name: {
        firstName: String,
    lastName: String
    },
    created: Date
});
```

As you can see, Mongoose allows me to create very flexible schemas with many different possible combinations of how I am able to organize my data.

In this next example, I am going to create two new schemas that will demonstrate how to create a relationship to another schema: `author` and `book`. The `book` schema will contain a reference to the `author` schema.

```
var authorSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    name: {
            firstName: String,
        lastName: String
    },
    biography: String,
    twitter: String,
    facebook: String,
    linkedin: String,
    profilePicture: Buffer,
    created: {
        type: Date,
        default: Date.now
    }
});
```

Above is the `author` schema that expands upon the concepts of the `user` schema that I created in the previous example. To link the Author and Book together, the first property of the `author` schema is an `_id` property that is an `ObjectId` schema type. `_id` is the common syntax for creating a primary key in Mongoose and MongoDB. Then, like the `user` schema, I've defined a `name` property containing the author's first and last name.

Expanding upon the `user` schema, the `author` contains several other `String` schema types. I've also added a `Buffer` schema type that could hold the author's profile picture. The final property holds the created date of the author; however, you may notice it is created slightly differently because it has defined a default value of "now". When an author is persisted to the database, this property will be set to the current date/time.

To complete the schema examples, let's create a `book` schema that contains a reference to the author by using the `ObjectId` schema type:

```
var bookSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
```

```
    title: String,
    summary: String,
    isbn: String,
    thumbnail: Buffer,
    author: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Author'
    },
    ratings: [
        {
            summary: String,
            detail: String,
            numberOfStars: Number,
            created: {
                type: Date,
                default: Date.now
            }
        }
    ],
    created: {
        type: Date,
        default: Date.now
    }
});
```

The `book` schema contains several properties of type `String`. As mentioned above, it contains a reference to the `author` schema. To further demonstrate the powerful schema definitions, the `book` schema also contains an `Array` of `ratings`. Each rating consists of a `summary`, `detail`, `numberOfStars`, and `created` date property.

Mongoose allows you the flexibility to create schemas with references to other schemas or, as in the above example with the `ratings` property, it allows you to create an `Array` of child properties that could be contained in a related schema (like book to author) or inline as in the above example (with book to a ratings `Array`).

Creating and Saving Mongoose Models

Since the `author` and `book` schemas demonstrate Mongoose's schema flexibility, I am going to continue using those schemas and derive an `Author` and `Book` model from them.

```
var Author = mongoose.model('Author', authorSchema);

var Book = mongoose.model('Book', bookSchema);
```

A Mongoose Model, when saved, creates a Document in MongoDB with the properties as defined by the schema it is derived from.

To demonstrate creating and saving an object, in this next example, I am going to create several objects: an `Author` Model and several `Book` Models. Once created, these objects will be persisted to MongoDB using the `save` method of the Model.

```
var jamieAuthor = new Author {
    _id: new mongoose.Types.ObjectId(),
    name: {
        firstName: 'Jamie',
        lastName: 'Munro'
    },
    biography: 'Jamie is the author of ASP.NET MVC 5 with Bootstrap and Knockout.js.',
    twitter: 'https://twitter.com/endyourif',
    facebook: 'https://www.facebook.com/End-Your-If-194251957252562/'
};

jamieAuthor.save(function(err) {
    if (err) throw err;

    console.log('Author successfully saved.');

    var mvcBook = new Book {
            _id: new mongoose.Types.ObjectId(),
            title: 'ASP.NET MVC 5 with Bootstrap and Knockout.js',
            author: jamieAuthor._id,
            ratings:[{
                summary: 'Great read'
            }]
    };

    mvcBook.save(function(err) {
        if (err) throw err;

        console.log('Book successfully saved.');
    });

    var knockoutBook = new Book {
            _id: new mongoose.Types.ObjectId(),
            title: 'Knockout.js: Building Dynamic Client-Side Web Applications',
            author: jamieAuthor._id
    };

    knockoutBook.save(function(err) {
        if (err) throw err;

        console.log('Book successfully saved.');
    });
});
```

In the above example, I've shamelessly plugged a reference to my two most recent books. The example starts by creating and saving a `jamieObject` that is created from an `Author` Model. Inside the `save` function of the `jamieObject`, if an error occurs, the application will output an exception. When the save is successful, inside the `save` function, the two book objects are created and saved. Similar to the `jamieObject`, if an error occurs when saving, an error is outputted; otherwise, a success message is outputted in the console.

To create the reference to the Author, the book objects both reference the `author` schema's `_id` primary key in the `author` property of the `book` schema.

Validating Data Before Saving

It's quite common for the data that will end up creating a model to be populated by a form on a webpage. Because of this, it's a good idea to validate this data prior to saving the Model to MongoDB.

In this next example, I've updated the previous author schema to add validation on the following properties: `firstName`, `twitter`, `facebook`, and `linkedin`.

```
var authorSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    name: {
        firstName: {
            type: String,
            required: true
        },
        lastName: String
    },
    biography: String,
    twitter: {
        type: String,
        validate: {
            validator: function(text) {
                return text.indexOf('https://twitter.com/') === 0;
            },
            message: 'Twitter handle must start with https://twitter.com/'
        }
    },
    facebook: {
        type: String,
        validate: {
            validator: function(text) {
```

```
                return text.indexOf('https://www.facebook.com/') === 0;
            },
            message: 'Facebook must start with https://www.facebook.com/'
        }
    },
    linkedin: {
        type: String,
        validate: {
            validator: function(text) {
                return text.indexOf('https://www.linkedin.com/') === 0;
            },
            message: 'LinkedIn must start with https://www.linkedin.com/'
        }
    },
    profilePicture: Buffer,
    created: {
        type: Date,
        default: Date.now
    }
});
```

The `firstName` property has been attributed with the `required` property. Now when I call the `save` function, Mongoose will return an error with a message indicating the `firstName` property is required. I chose not to make the `lastName` property required in case Cher or Madonna were to be authors in my database.

The `twitter`, `facebook`, and `linkedin` properties all have very similar custom validators applied to them. They each ensure that the values begin with the social networks' respective domain name. These fields are not required, so the validator will only be applied when data is supplied for that property.

Searching for and Updating Data

An introduction to Mongoose wouldn't be complete without an example of searching for a record and updating one or more properties on that object.

Mongoose provides several different functions to find data for a specific Model. The functions are `find`, `findOne`, and `findById`.

The `find` and `findOne` functions both accept an object as input allowing for complex searches, whereas `findById` accepts just a single value with a callback function (an

example will follow shortly). In this next example, I am going to demonstrate how to find all books that contain the string "mvc" in their title.

```
Book.find({
    title: /mvc/i
}).exec(function(err, books) {
    if (err) throw err;

    console.log(books);
});
```

Inside the `find` function, I am searching for the case insensitive string "mvc" on the `title` property. This is accomplished using the same syntax for searching a string with JavaScript.

The find function call also be chained to other query methods, such as `where`, `and`, `or`, `limit`, `sort`, `any`, etc.

Let's expand upon the previous example to limit our results to the first five books and sort on the created date descending. This will return up to the five most recent books containing "mvc" in the title.

```
Book.find({
    title: /mvc/i
}).sort('-created')
.limit(5)
.exec(function(err, books) {
    if (err) throw err;

    console.log(books);
});
```

After applying the `find` function, the order of the other functions is not important because all of the chained functions are compiled together into a single query and *not* executed until the `exec` function is called.

As I mentioned earlier, the `findById` is executed a bit differently. It executes immediately and accepts a callback function, instead of allowing for a chain of functions. In this next example, I am querying a specific author by their `_id`.

```
Author.findById('59b31406beefa1082819e72f', function(err, author) {
    if (err) throw err;

    console.log(author);
});
```

The `_id` in your case might be slightly different. I copied this `_id` from a previous `console.log` when finding a list of books with "mvc" in their title.

Once an object has been returned, you can modify any of its properties to update it. Once you have made the necessary changes, you call the `save` method, just like when you were creating the object. In this next example, I will extend the `findbyId` example and update the `linkedin` property on the author.

```
Author.findById('59b31406beefa1082819e72f', function(err, author) {
    if (err) throw err;

    author.linkedin = 'https://www.linkedin.com/in/jamie-munro-8064ba1a/';

    author.save(function(err) {
        if (err) throw err;

        console.log('Author updated successfully');
    });
});
```

After the author is successfully retrieved, the `linkedin` property is set and the `save` function is called. Mongoose is able to detect that the `linkedin` property was changed, and it will send an update statement to MongoDB on only the properties that have been modified. If an error occurred when saving, an exception will be thrown and will halt the application. When successful, a success message is logged to the console.

Mongoose also offers two additional functions that make finding an object and saving it in a single step with the appropriately named functions: `findByIdAndUpdate` and `findOneAndUpdate`. Let's update the previous example to use the `findByIdAndUpdate`.

```
Author.findByIdAndUpdate('59b31406beefa1082819e72f',
    { linkedin: 'https://www.linkedin.com/in/jamie-munro-8064ba1a/' },
    function(err, author) {
        if (err) throw err;

        console.log(author);

```

```
});
```

In the previous example, the properties to update are supplied as an object to the second parameter of the `findByIdAndUpdate` function. The callback function is now the third parameter. When the update is successful, the `author` object returned contains the updated information. This is logged to the console to see the updated author's properties.

Final Sample Code

Throughout this article, I provided small snippets of code identifying a very specific action, such as creating a schema, creating a model, etc. Let's put it all together in a full example.

Firstly, I've created two additional files: `author.js` and `book.js`. These files contain their respective schema definitions and the model creation. The final line of code makes the model available for use in the `index.js` file.

Let's start with the author.js file:

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

var authorSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    name: {
        firstName: {
            type: String,
            required: true
```

```
        },
        lastName: String
    },
    biography: String,
    twitter: {
        type: String,
        validate: {
            validator: function(text) {
                return text.indexOf('https://twitter.com/') === 0;
            },
            message: 'Twitter handle must start with https://twitter.com/'
        }
    },
    facebook: {
        type: String,
        validate: {
            validator: function(text) {
                return text.indexOf('https://www.facebook.com/') === 0;
            },
            message: 'Facebook must start with https://www.facebook.com/'
        }
    },
    linkedin: {
        type: String,
        validate: {
            validator: function(text) {
                return text.indexOf('https://www.linkedin.com/') === 0;
            },
            message: 'LinkedIn must start with https://www.linkedin.com/'
        }
    },
    profilePicture: Buffer,
    created: {
        type: Date,
        default: Date.now
    }
});

var Author = mongoose.model('Author', authorSchema);

module.exports = Author;
```

Next comes the `book.js` file:

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

var bookSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    title: String,
    summary: String,
    isbn: String,
    thumbnail: Buffer,
    author: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Author'
```

```
    },
    ratings: [
        {
            summary: String,
            detail: String,
            numberOfStars: Number,
            created: {
                type: Date,
                default: Date.now
            }
        }
    ],
    created: {
        type: Date,
        default: Date.now
    }
});

var Book = mongoose.model('Book', bookSchema);

module.exports = Book;
```

And finally, the updated `index.js` file:

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

var Author = require('./author');
var Book = require('./book');

mongoose.connect('mongodb://localhost/mongoose_basics', function (err) {
    if (err) throw err;

    console.log('Successfully connected');

    var jamieAuthor = new Author({
        _id: new mongoose.Types.ObjectId(),
        name: {
            firstName: 'Jamie',
            lastName: 'Munro'
        },
        biography: 'Jamie is the author of ASP.NET MVC 5 with Bootstrap and Knockout.js
        twitter: 'https://twitter.com/endyourif',
        facebook: 'https://www.facebook.com/End-Your-If-194251957252562/'
    });

    jamieAuthor.save(function(err) {
        if (err) throw err;

        console.log('Author successfully saved.');

        var mvcBook = new Book({
            _id: new mongoose.Types.ObjectId(),
            title: 'ASP.NET MVC 5 with Bootstrap and Knockout.js',
            author: jamieAuthor._id,
            ratings:[{
```

```
                summary: 'Great read'
            }]
        });

        mvcBook.save(function(err) {
            if (err) throw err;

            console.log('Book successfully saved.');
        });

        var knockoutBook = new Book({
            _id: new mongoose.Types.ObjectId(),
            title: 'Knockout.js: Building Dynamic Client-Side Web Applications',
            author: jamieAuthor._id
        });

        knockoutBook.save(function(err) {
            if (err) throw err;

            console.log('Book successfully saved.');
        });
    });
});
```

In the above example, all of the Mongoose actions are contained within the `connect` function. The `author` and `book` files are included with the `require` function after including the `mongoose` library.

With MongoDB running, you can now run the complete Node.js application with the following command:

```
node index.js
```

After I saved some data to my database, I updated the `index.js` file with the find functions as follows:

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

var Author = require('./author');
var Book = require('./book');

mongoose.connect('mongodb://localhost/mongoose_basics', function (err) {
    if (err) throw err;

    console.log('Successfully connected');

    Book.find({
```

```
        title: /mvc/i
    }).sort('-created')
    .limit(5)
    .exec(function(err, books) {
        if (err) throw err;

        console.log(books);
    });

    Author.findById('59b31406beefa1082819e72f', function(err, author) {
        if (err) throw err;

        author.linkedin = 'https://www.linkedin.com/in/jamie-munro-8064ba1a/';

        author.save(function(err) {
            if (err) throw err;

            console.log('Author updated successfully');
        });
    });

    Author.findByIdAndUpdate('59b31406beefa1082819e72f', { linkedin: 'https://www.linke
        if (err) throw err;

        console.log(author);
    });
});
```

Once again, you can run the application with the command: `node index.js`.

Summary

After reading this article, you should be able to create extremely flexible Mongoose Schemas and Models, apply simple or complex validation, create and update documents, and finally search for the documents that were created.

Hopefully you now feel comfortable using Mongoose. If you are looking to learn more, I would suggest reviewing the Mongoose Guides which delve into more advanced topics such as population, middleware, promises, etc.

Happy hunting (poor Mongoose animal reference)!

Jamie Munro
Software Developer

Jamie Munro is the author of ASP.NET MVC 5 with Bootstrap and Knockout.js, Knockout.js: Building Dynamic Client-Side Web Applications, 20 Recipes for Programming MVC 3, and 20 Recipes for Programming PhoneGap. He has been developing websites and web applications for over 15 years. For the past ten years, Jamie has been acting as a lead developer by mentoring younger developers to enhance their skills. Using his love of mentoring people, Jamie began his writing career on his personal blog back in 2009. As the success of Jamie's blog grew, he turned his writing passion to books about web development in hopes that his many years of experience could be passed on to his readers.

endyourif