**MODULE**

**Week 6: Advanced MongoDB**

*Nawfal Ali*

*Updated 1 September 2019*

**Question**: MongoDB is very flexible. It allows me to insert documents in collections with different shapes. But, can I have strongly-typed documents? and is there any way to validate the input data?

**Answer**: Using MongooseJS, you can have strongly-typed schemas, data validation, data pre-processing, and many other features.

**Question**: I am developing a web app that stores books and each book has one Author. How can I create a one to one relationship?

**Approach 1**: the first way to do that is to store the author’s object as a property of the book

For example:

1. let Author = {
2. \_id: 1234,
3. firstName: 'Tim',
4. lastName: 'John',
5. age: 35
6. };
7. let book1 = {
8. \_id: 789,
9. title: 'FIT2095 Book',
10. author: {
11. id: 1234,
12. firstName: 'Tim',
13. lastName: 'John',
14. age: 35
15. },
16. isbn: 9876
17. }

But this approach has several problems. I have to iterate through all books documents each time I need to update an author.

**Approach 2**: Let’s save the author’s ID as a property of his book.

1. let Author = {
2. \_id: 1234,
3. firstName: 'Tim',
4. lastName: 'John',
5. age: 35
6. };
7. let book1 = {
8. \_id: 789,
9. title: 'FIT2095 Book',
10. author\_id: 1234,
11. isbn: 9876
12. }

**Question**: Is there any tool or package that helps to create such a relation?

**Answer**: Yes, Mongoose.

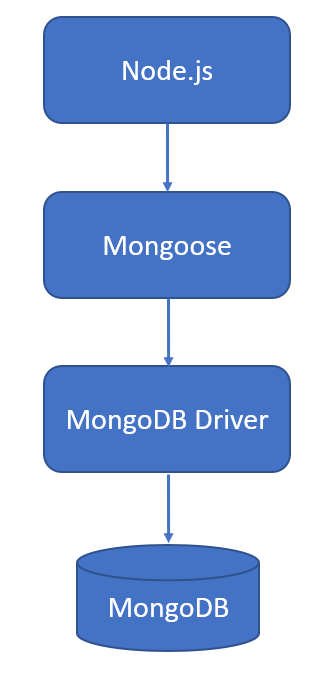
**What is Mongoose?**

Sources:[2,3,4]

**Mongoose** is an object data modelling (ODM) library that provides a modeling environment for your collections. It enforces structure as needed while still keeping the flexibility and scalability of MongoDB.

**Mongoose** is a JavaScript framework that is commonly used in a Node.js application with a MongoDB database. Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in typecasting, validation, query building, business logic hooks and more, out of the box.

It uses MongoDB driver to interact with MongoDB storage.

[](https://www.alexandriarepository.org/wp-content/uploads/20180825131435/mongoose.png)

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
9. Decimal128
10. Map

SchemaTypes handle definition of path

1. defaults
2. validation
3. getters
4. setters
5. field selection defaults for queries

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

We will discuss them in details soon.

**In a nutshell, we will use Mongoose functions and options and perform all the CRUD (Create, Retrieve, Update, Delete) operations on MongoDB database.**

**Schema in Mongoose**

Let’s start by creating a database that consists of two collections (Authors and Books). We will implement a simple relationship between them which is One-To-One. In other words, each book has an author.

Everything in Mongoose starts with a Schema. Each schema maps to a MongoDB collection and defines the shape of the documents within that collection.

1. var mongoose = require('mongoose');

Initially, get a reference from Mongoose package

Create your schema:

1. var authorSchema = mongoose.Schema({});

Now let’s add our fields:

1. var authorSchema = mongoose.Schema({
2. \_id: mongoose.Schema.Types.ObjectId,
3. name: {
4. firstName:String,
5. lastName:String,
6. },
7. age: Number,
8. created: Date
9. });

Using the above schema, each document will get four items, which are: \_id, name, age, and created.

Now, I need to add some roles to my schema:

* Field name is mandatory (i.e. required)

1. name: {
2. firstName: {
3. type: String,
4. required: true
5. },
6. lastName: String
7. }

From the above, to make a field required, set the field an object with two properties: the **type** and **required** which needs a boolean value.

* Field age should be a number between 10 and 110

1. age: {
2. type: Number,
3. validate: {
4. validator: function (ageValue) {
5. **return** ageValue >= 10 && ageValue <= 110;
6. },
7. message: 'Age should be a number between 10 and 110'
8. }
9. }

Similar to name, age becomes an object with two properties: the type and validate which has a boolean function that takes an input represents the field’s (age) value and returns true if the value is valid (based on our logic) and false otherwise.  The **message** property is a string represents the output that will be printed out when an invalid value is detected.

There is another way to validate the minimum and maximum values of numeric fields by using the **min** and **max** properties:

1. age: { type: Number, min: 5, max: 20 }

* Field created has a default value

1. created: {
2. type: Date,
3. default: Date.now
4. }

The field created is an object with two properties: the type and the default value of the field if no value is provided.

So, the schema in one piece:

1. const mongoose = require('mongoose');
2. let authorSchema = mongoose.Schema({
3. \_id: mongoose.Schema.Types.ObjectId,
4. name: {
5. firstName: {
6. type: String,
7. required: true
8. },
9. lastName: String
10. },
11. age: {
12. type: Number,
13. validate: {
14. validator: function (ageValue) {
15. **return** ageValue >= 10 && ageValue <= 110;
16. },
17. message: 'Age should be a number between 10 and 110'
18. }
19. },
20. created: {
21. type: Date,
22. default: Date.now
23. }
24. });

Now its time to create the Book schema, which has to have the following fields: \_id (ObjectId), title (String and required), isdn (String), author (ObjectId, reference wit Author Schema), and created(Date with the default value to the current date/time).

* field **\_id**:

1. \_id: mongoose.Schema.Types.ObjectId,

* field **title:**

1. title: {
2. type: String,
3. required: true
4. },

* field **isbn**:

1. isbn: String,

* field **author**:

1. author: {
2. type: mongoose.Schema.Types.ObjectId,
3. ref: 'Author'
4. },

Field **author** is an object with two properties: **type** and **ref**. The type is ObjectId because it will only have the Id of the author’s document. the **ref** indicates the name of the schema, which is, in this case, the **Author**.

* field **created**:

1. created: {
2. type: Date,
3. default: Date.now
4. }

So, the Book schema in one piece:

1. const mongoose = require('mongoose');
2. let bookSchema = mongoose.Schema({
3. \_id: mongoose.Schema.Types.ObjectId,
4. title: {
5. type: String,
6. required: true
7. },
8. isbn: String,
9. author: {
10. type: mongoose.Schema.Types.ObjectId,
11. ref: 'Author'
12. },
13. created: {
14. type: Date,
15. default: Date.now
16. }
17. });

**Mongoose Models**

‘Models’ are higher-order constructors that take a schema and create an instance of a document equivalent to records in a relational database. A Mongoose model is a wrapper on the Mongoose schema. A Mongoose schema defines the structure of the document, default values, validators, etc., whereas a Mongoose model provides an interface to the database for creating, querying, updating, deleting records, etc. [4]

So, in order to export a model, we need to invoke the **model** constructor and pass it a string represents **the name of the collection** and a reference to the schema.

For example, the **Author** schema:

1. const mongoose = require('mongoose');
2. let authorSchema = mongoose.Schema({
3. \_id: mongoose.Schema.Types.ObjectId,
4. name: {
5. firstName: {
6. type: String,
7. required: true
8. },
9. lastName: String
10. },
11. age: {
12. type: Number,
13. validate: {
14. validator: function (ageValue) {
15. **return** ageValue >= 10 && ageValue <= 110;
16. },
17. message: 'Age should be a number between 10 and 110'
18. }
19. },
20. // age : { type: Number, min: 5, max: 20 },
21. created: {
22. type: Date,
23. default: Date.now
24. }
25. });
26. module.exports = mongoose.model('Author', authorSchema);

and the **Book** model:

1. const mongoose = **require**('mongoose');
2. let bookSchema = mongoose.**Schema**({
3. \_id: mongoose.Schema.Types.ObjectId,
4. title: {
5. type: String,
6. required: true
7. },
8. isbn: String,
9. author: {
10. type: mongoose.Schema.Types.ObjectId,
11. ref: 'Author'
12. },
13. created: {
14. type: Date,
15. default: Date.now
16. }
17. });
18. module.exports = mongoose.**model**('Book', bookSchema);

Now, our database has two collections ‘Author’ and ‘Book’.

**Inserting Documents**

Now, its time to develop the main app js file and insert new documents into the DB.

Let’s start by referencing Mongoose package

1. const mongoose = require('mongoose');

referencing our schemas:

1. const Author = require('./models/author');
2. const Book = require('./models/book');

Create a Mongoose URL string which has syntax: mongodb**://ServerAddress: Port//DbName**

1. let url='mongodb://localhost:27017/libDB';

and then connect

1. mongoose.**connect**(url, **function** (err) {});

The **err** parameter will get a value if an error occurs.

Let’s create a new author:

1. let author1 = new Author({
2. \_id: new mongoose.Types.ObjectId(),
3. name: {
4. firstName: 'Tim',
5. lastName: 'John'
6. },
7. age: 80
8. });

Now, let’s save it:

1. author1.save(function (err) {});

the save function has a callback that will get executed after the save operation is completed.

To test the data type validation, try to insert a document where the age is a non-numeric value:

1. let author1 = new Author({
2. \_id: new mongoose.Types.ObjectId(),
3. name: {
4. firstName: 'Tim',
5. lastName: 'John'
6. },
7. age: '8a'
8. });

Next, I need to add two books for this author (author1 or ‘Tim John’):

1. var book1 = new Book({
2. \_id: new mongoose.Types.ObjectId(),
3. title: 'FIT2095 Book ',
4. author: author1.\_id,
5. isbn: '123456',
6. });

Line 4, field **author**, which is of type ObjectId and referencing Author Schema (look at line 11 in the Book Schema) is set to the ID of the author we have just created (author1).

Similar to the author, we have to save the new document by calling the function **save**.

1. book1.save(function (err) {
2. **if** (err) **throw** err;
3. console.log('Book1 successfully Added to DB');
4. });

Let’s add another book:

1. var book2 = new Book({
2. \_id: new mongoose.Types.ObjectId(),
3. title: 'MEAN Stack with FIT2095',
4. author: author1.\_id
5. });

and let’s save it

1. book2.save(function (err) {
2. **if** (err) **throw** err;
3. console.log('Book2 successfully add to DB');
4. });

Now, the app.js in one piece:

1. const mongoose = require('mongoose');
2. const Author = require('./models/author');
3. const Book = require('./models/book');
4. mongoose.connect('mongodb://localhost:27017/libDB', function (err) {
5. **if** (err) {
6. console.log('Error in Mongoose connection');
7. **throw** err;
8. }
9. console.log('Successfully connected');
10. let author1 = new Author({
11. \_id: new mongoose.Types.ObjectId(),
12. name: {
13. firstName: 'Tim',
14. lastName: 'John'
15. },
16. age: 80
17. });
18. author1.save(function (err) {
19. **if** (err) **throw** err;
20. console.log('Author successfully Added to DB');
21. var book1 = new Book({
22. \_id: new mongoose.Types.ObjectId(),
23. title: 'FIT2095 Book ',
24. author: author1.\_id,
25. isbn: '123456',
26. });
27. book1.save(function (err) {
28. **if** (err) **throw** err;
29. console.log('Book1 successfully Added to DB');
30. });
31. var book2 = new Book({
32. \_id: new mongoose.Types.ObjectId(),
33. title: 'MEAN Stack with FIT2095',
34. author: author1.\_id
35. });
36. book2.save(function (err) {
37. **if** (err) **throw** err;
38. console.log('Book2 successfully add to DB');
39. });
40. });
41. });

Note: the creating and saving of **book1** and **book2** have been implemented insode the callback of **author1.save** function. Why??

Node.js is asynchrounus and both **book1** and **book2** required the author’s ID (lines 31 and 43). Therefore, we have to create them after the save operation of the author is done.

**.insertMany() function**

This function takes as input an array of documents (objects) and inserts them if they are valid.

**Querying Documents with Mongoose**

Mongoose models have several static functions that can be used for CRUD operations.

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

For more details about all the above functions, please navigate to:<https://mongoosejs.com/docs/queries.html>

**Finding Documents**

Simple Query

1. Book.**find**({ 'name.firstName': 'Tim' }, **function** (err, docs) {
2. // docs is an array
3. });

Retrieving only certain fields

Example: get the age of all documents with first name = ‘Tim’

1. Author.**find**({ 'name.firstName': 'Tim' }, 'age', **function** (err, docs) {
2. //docs is an array
3. console.**log**(docs);
4. });

Find only one

1. Model.findOne({name.firstName:'Tim'}, 'age', function (err, doc) {
2. //doc is a document
3. });

Find by ID

1. Author.findById(author1.\_id, 'age', function (err, docs) {
2. console.log(docs);
3. });

**Using the ‘where’ clause**

Using ‘where’, we can create complex expressions.

Example: Find all documents that have firstName starts with the letter ‘T’ and the age >= 25.

1. Author.where({ 'name.firstName': /^T/ }).where('age').gte(25).exec(function (err, docs) {
2. console.log(docs);
3. });

Note **exec** indicates the end of the chain and invokes the callback function.

and age <=35

1. Author.where({ 'name.firstName': /^T/ }).where('age').gte(25).lte(35).exec(function (err, docs) {
2. console.log(docs);
3. });

limit the result to 10 only:

1. Author.where({ 'name.firstName': /^T/ }).where('age').gte(25).lte(35).limit(10).exec(function (err, docs) {
2. console.log(docs);
3. });

and sort the results in ascending order by the **age**

1. Author.where({ 'name.firstName': /^T/ }).where('age').gte(25).lte(35).limit(10).sort('age').exec(function (err, docs) {
2. console.log(docs);
3. });

**Update Documents**

**.updateOne() function**

This document updates only the first document that matches criteria.

1. Author.**updateOne**({ 'name.firstName': 'Alex' }, { $set: { 'name.firstName': 'John' } }, **function** (err, doc) {
2. console.**log**(doc);
3. });

**.updateMany() function**

Similar to updateOne() except MongoDB will update all the document that matche criteria.

**Delete Documents**

**.deleteOne() function**

1. Author.deleteOne({ 'name.firstName': 'Tim' }, function (err, doc) {
2. console.log(doc);
3. });

deleteOne() function deletes a document based on condition (criteria).

**.deleteMany()**

This function deletes all the documents that match the criteria.

1. Author.deleteMany({ 'name.firstName': 'Tim' }, function (err, doc) {
2. console.log(doc);
3. });

The full list of Model’s functions can be found here: <https://mongoosejs.com/docs/api.html#Model>

**Lab Week 7**

Update the To-Do web app that you have developed in week 6 by adding new features:

1. It uses Mongoose to store all the data instead of plain MongoDB
2. It has two collections: Tasks and Developers
3. Each task has the following fields (Tasks Schema):
   1. Task name
   2. Assign to (the ID of the developer’s document)
   3. Due date (should be in date datatype, not a string)
   4. Task status (either InProgress or Complete)
   5. Task Description
4. Each developer has the following fields (Developers Schema):
   1. name: an object has
      1. first name  (required)
      2. last name
   2. Level: String and can be either ‘Beginner or Expert’. (required and should be saved in all caps)
   3. Address: Object has
      1. State
      2. Suburb
      3. Street
      4. Unit
5. Pages
   1. Insert a new Developer: adds a new document to ‘Developers’ collection
   2. Get all developers page: shows all the developers in a table format (including the \_id field)
   3. Insert new task page: adds a new document to Tasks collection. (Hint: Get the Developer’s ID from the  ‘Get all developers page’ manually)
   4. Get all tasks page: shows all the tasks in a table format
   5. Delete task by taskID: the page takes a taskID as input and deletes its tasks from the DB
   6. Delete all the completed tasks
   7. Update task status by taskID: the page takes two inputs: a taskID and a new status (either InProgress or Complete). It sets the new status to the task with taskID.
6. Add a logo for the home page
7. change the background of all pages using a CSS file
8. redirect the client to the get all tasks page after the insert, update and delete operations.
9. Deploy your application to a VM in your GCP account.

References:

1. https://mongoosejs.com/
2. Practical Node.js, Building Real-World Scalable Web Apps [Book]
3. https://mongoosejs.com/docs/guide.html
4. https://medium.freecodecamp.org/introduction-to-mongoose-for-mongodb-d2a7aa593c57

Copyright © Monash University, unless otherwise stated. All Rights Reserved, except for individual components (or items) marked with their own licence restrictions

I have completed reading this Module

This registers completion and allows you to provide feedback

[Privacy](https://www.alexandriarepository.org/module/week-6-advanced-mongodb/)

Copyright © 2019 Monash University, unless otherwise stated|[Disclaimer and Copyright](https://www.monash.edu/disclaimer-copyright)|[Privacy](https://www.monash.edu/privacy)|[Service Status](http://status.alexandriarepository.org/)