Any database which is supported by a node can be used for an express application. Some of the popular databases are MySQL, Redis, MongoDB etc.

A user can interact with a database in the following ways.

* Using query language of the respective databases' - eg: SQL
* Using an ODM(Object Data Model) or (ORM) Object-Relational Model

In this module, we will be dealing with usage of Object Data Model to interact with the database.

**Need for ODM:**

Now let us assume, we want to maintain the details of the employees in below format in MongoDB database.

|  |  |  |
| --- | --- | --- |
| **empName** | **empId** | **location** |
| Alex | 123456 | Europe |
| Sam | 432567 | California |

If we try to insert the below employee record into the collection, it will be inserted. But if we carefully observe, the below record did not exactly match format as we discussed.

* { empName : "John", employeeId : 324123, location : "France"}

After inserting the above record, the collection will be looking as shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| **empName** | **empId** | **location** | **employeeId** |
| Alex | 123456 | Europe |  |
| Sam | 432567 | California |  |
| John |  | France | 324123 |

You can observe that the structure of the database has been violated.

MongoDB stores the data in the form of document in the collection. Since there is no predefined structure and constraints on values for fields, a document of any structure having any value for fields can be inserted into the collection. It becomes the responsibility of developers to strictly maintain the structure of the document and apply constraints on the values for fields, which can be easily violated. So we need a library which can maintain or model the structure of documents and impose restrictions on the values for fields

In order to ensure that the structure of the database is maintained, we need an **Object Data Modeler**, which ensures the maintenance of the collection.

**Object Data Modeling (ODM)**

An ODM is used to map the data as JavaScript objects to the underlying database format. They provide an option to perform validation and checking data to ensure data integrity.

There are many ODM based libraries available on NPM. One such library is Mongoose.

Mongoose is an Object Data Modeling (ODM) tool, which is created to work on asynchronous MongoDB environment. It imposes the structure and constraints by mapping JavaScript objects to documents in the database. It also provides the functionality for validating field values and methods for selecting, adding, modifying and deleting documents in the collection

Now if we insert the same employee record with the help of Mongoose, then the collection will be looking as shown below:

|  |  |  |
| --- | --- | --- |
| **empName** | **empId** | **location** |
| Alex | 123456 | Europe |
| Sam | 432567 | California |
| John |  | France |

Here we can observe that the field which did not match the name in the collection will not be inserted.

If we had set **a required**constraint for the empId field then the document will not be inserted at all.

**Example 2:**

Now if we try to insert the below employee record into the collection, it will be inserted. But if we carefully observe, the below record did not match the value as expected.

* { empName : "John", empId : "324123", location : "France"}

After inserting the above record, the collection will be looking as shown below:

|  |  |  |
| --- | --- | --- |
| **empName** | **empId** | **location** |
| Alex | 123456 | Europe |
| Sam | 432567 | California |
| John | "324123" | France |

If we insert the above record, with the help of Mongoose then the collection will be looking as shown below:

|  |  |  |
| --- | --- | --- |
| **empName** | **empId** | **location** |
| Alex | 123456 | Europe |
| Sam | 432567 | California |
| John | 324123 | France |

Here we can observe that, the value for empId field got type-cast to Number type.

If Mongoose is not able to convert the value to the specified type then the document will not be inserted.

While working with Mongoose we will come across some common terminologies like *Schema*and *Models.*Let us begin our understanding of Mongoose ODM by visiting these terms.

**Schema:**

The structure of the document to be inserted in the database collection is defined through '*schema*' in mongoose. Using schema we can perform validations, provide default values to the properties etc.

Example:

1. const empSchema = [{
2. "empId": Number,
3. "empName": String,
4. "address": {
5. "doorNo": String,
6. "lane": String,
7. "pincode: Number
8. }
9. }]

**Models:**

Through schema we can define  structure of the document. Finally we have interact with database collection to store or retrieve data. In order to perform database interaction mongoose provides a 'Model' method that take the predefined schema and  create an instance of a document which is similar to the records of relational database. It provides an interface to the database for creating, querying, updating, deleting records, etc. A Mongoose model is a wrapper on the Mongoose schema.

Creating a Mongoose model comprises primarily of three parts:

1. Referencing Mongoose
2. Defining the Schema
3. Creating a Model

Let us take a look at each of these parts in detail, but first let us begin by learning to install mongoose library.

For installing the Mongoose library, use the node package manager.

1. npm install --save mongoose

Executing the above command downloads the Mongoose library and add a dependency entry in your 'package.json' file.

To establish a connection to the MongoDB database, we need to create a connection object using Mongoose. This is done through the below code.

1. const mongoose = require('mongoose');
2. mongoose.connect('mongodb://localhost:27017/Database\_Name');

In Line 1, we are importing the Mongoose library into our application.

In Line 2, the connect() method takes **MongoDB**protocol followed by host and the database name as a parameter.

Let us now establish a connection to the MongoDB database for storing details of **Stone\_King\_Multiplex** using the below code:

1. const mongoose = require('mongoose');
2. mongoose.Promise = global.Promise; *//Line 1*
3. mongoose.connect('mongodb://localhost:27017/MultiplexDB', { useNewUrlParser: true }) *//Line2*

Line 1: Mongoose can be coded using callbacks or Promises. This line configures Mongoose to use the global promise library

Line 2: Here we are specifying the URI where the data will be stored/retrieved. The second parameter { useNewUrlParser: true } is to suppress the below warning thrown by MongoDB



To create a collection in mongoose, we must first start with creating a Mongoose schema.

A schema defines document properties through an object where the key name corresponds to the property name in the collection.

The Schema allows you to define the fields stored in each document along with their validation requirements and default values.

To create a schema, we need to use the following lines of code:

1. const mongoose = require('mongoose');
2. let schema = mongoose.Schema({ property\_1: Number, property\_2: String },{collection : 'collection\_name'});

In a schema, we will define the data types of the properties in the document.

The most commonly used types in **Schema**are:

* String

e.g: employee name

* Number

e.g: employee Id

* Date

Date will be stored in ISO date format. For e.g: 2018-11-15T15:22:00

* Boolean

It will take true or false, generally used for validations, which we will see in next part of course.

* ObjectId

Usually, ObjectId is assigned by MongoDB itself, every document inserted in MongoDB will have a unique **Id** which is created automatically by MongoDB is of ObjectId type.

* Array

When we need to store an array of data or even sub-document array type is used. For e.g: ["Cricket","Football"]

Let us now create a schema for **Stone\_King\_Multiplex**collection.

Stone\_King\_Multiplex wants to store movie id, movie name, language, movie details and bookings done for that particular movie.

The code to create the required schema is given below.

1. const schema = {
2. "movieId": Number,
3. "movieName": String,
4. "language": String,
5. "showDetails": [
6. {
7. "showId": Number,
8. "movieId": Number,
9. "fare": Number,
10. "availableSeats": Number
11. }
12. ],
13. "bookings": [
14. {
15. "bookingId": Number,
16. "customerName": String,
17. "bookingCost": Number,
18. "showId": Number,
19. "noOfTickets": Number,
20. "showDateTime": Date
21. }
22. ]
23. }
24. let movieSchema = mongoose.Schema(schema , { collection: 'Movie' }); *//Line 1*

Line 1: Using the Schema class provided by the Mongoose library, we can set the **schema**(provided as a 1st parameter) of a particular **collection** (provided in the 2nd parameter).

**Note:**It is a standard practice to have timestamps - (created at & updated at) field for each document inserted into the collection. This can be done by adding a timestamp option to the Schema class, as shown below:

1. let movieSchema = mongoose.Schema(schema, { collection: 'Movie', timestamps: true })

Your collection will have two extra fields whose, date values are in ISO formats, added as shown:

https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_0126228343340072962757_shared/web-hosted/assets/create.PNG

To ensure data entered into the collection is as per the requirement, we can configure validations to the schema. Let us now look at adding validations for our schema.

Mongoose provides a way to validate data before you save that data to a database. Data validation is important to make sure that "**invalid**" data does not get persisted in your application. This ensures data integrity. A benefit of using Mongoose, when inserting data into MongoDB is its built-in support for data types, and the automatic validation of data when it is persisted.

Mongoose’s validators are easy to configure. When defining the schema, specific validations need to be configured.

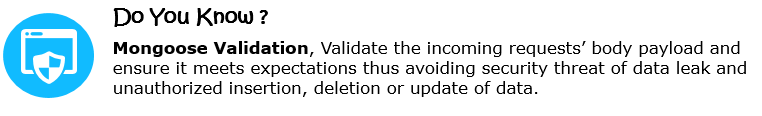
The following rules should be kept in mind while adding validation:

* Validations are defined in the Schema
* Validation occurs when a document attempts to be saved
* Validation will not be applied for default values
* Validators will not be triggered on undefined values. The only exception to this is the **required** validator
* Customized validators can be configured

In built validators

Mongoose has several built in validators.

* required validator can be added to all SchemaTypes
* Number schema type has min and max validators
* Strings have enum and match validators



Let us see how to add validator to a Schema.

# Required

If you want any field to be mandatory, use the required property.

1. let schema = mongoose.Schema({
2. name:{
3. required : true
4. }
5. })

# Types

You can declare a schema type using the type directly, or an object with a type property.

Below code snippet shows how to declare schema type using an object with a type property.

1. let schema = mongoose.Schema({
2. property\_1: {
3. type: String
4. },
5. property\_2:{
6. type : Number
7. }
8. });

# Default

Your schema can define default values for certain property. If you create a new document without that property set, the default value provided to that property will be assigned.

Below code snippet shows how to add default to schema type.

1. let schema = mongoose.Schema({
2. property\_1: {
3. default: "Client"
4. },
5. property\_2:{
6. default: 1
7. }
8. });

# Custom validations

In Mongoose we can specify custom validators that are tailored specific to fields if the built-in validators are not enough. They are provided as values of validate property

Below code snippet shows how to add custom validator to schema type.

1. let schema = mongoose.Schema({
2. property\_1: {
3. validate: (value) => { */\*Validation code\*/* }
4. }
5. });

For more information on validators, please visit [Mongoose docs](https://mongoosejs.com/docs/validation.html).

Let us now add validation and specify types for Movie Schema.

1. mongoose.set('useCreateIndex',true); *//added to suppress warning in mongoose v5.2.9*
2. const schema = {
3. "movieId": {
4. required: [true, 'Required field'],
5. type: Number,
6. unique:true
7. },
8. "movieName": {
9. required: [true, 'Required field'],
10. type: String
11. },
12. "language": {
13. required: [true, 'Required field'],
14. match: [/^[A-Za-z]{3,8}$/,'Please enter only alphabets'],
15. type: String
16. },
17. "showDetails": {
18. type: [
19. {
20. "showId": {
21. required: [true, 'Required field'],
22. type: Number
23. },
24. "movieId": {
25. required: [true, 'Required field'],
26. type: Number
27. },
28. "fare": {
29. required: [true, 'Required field'],
30. type: Number
31. },
32. "availableSeats": {
33. required: [true, 'Required field'],
34. type: Number
35. },
36. "showDate": {
37. required: [true, 'Required field'],
38. type: Date,
39. validate: [(showDate) => new Date(showDate) >= new Date(), 'Date should be greater than today']
40. }
41. }
42. ],
43. default: []
44. },
45. "bookings": {
46. type: [
47. {
48. "bookingId": {
49. required: [true, 'Required field'],
50. type: Number
51. },
52. "customerName": {
53. type: String,
54. default: "New User"
55. },
56. "bookingCost": {
57. required: [true, 'Required field'],
58. type: Number
59. },
60. "showId": {
61. required: [true, 'Required field'],
62. type: Number
63. },
64. "noOfTickets": {
65. type: Number,
66. min: [1, 'Minimum 1 ticket'],
67. },
68. "bookedAt": {
69. type: Date,
70. default: new Date().toLocaleDateString()
71. }
72. }
73. ],
74. default: []
75. }
76. }

Other than Schema Types, the error message for validators can be customized by enclosing the validator value and the message in [] brackets.

E.g.: min: [1, 'Minimum 1 ticket']

**Note:**Ensure that there is no pre-existing data in the database as the unique validation will not be applied if data is already present.

Now that our Movie schema is ready, let us create a model for the schema.

To use our schema definition, we need to wrap the **Schema**into a **Model** object we can work with.

Model provides an object which provides access to query documents in a named collection. Schemas are compiled into models using the **model()** method.

1. let Model = mongoose.model(name , schema)

The first argument is the singular name of the collection for which you are creating a **Model**.

The model() function makes a copy of schema. Make sure that you have added everything you want to schema before calling model().

Let us now create a model for our **Movie**collection using the below code.

1. let movieModel = mongoose.model("Movie",movieSchema)

Here **movieModel** is a collection object that will be used to perform CRUD operations.

Now we have created the model for our database collection, let us insert some data into the collection.

Mongoose library offers several functions to perform various CRUD (Create-Read-Update-Delete) operations. Each of these functions returns a mongoose Query object. The query syntax is similar to the one written in the MongoDB shell.

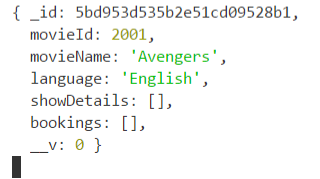
# Inserting a document into the collection

To insert a single document to MongoDB, use create() method, it will take the document instance as a parameter. Insertion happens asynchronously and any operations dependent on the inserted document have to happen by unwrapping the promise response. Here we will be making use of Async-Await to do the insert operation.

We can insert a new document into our Movie Collection using the below code:

1. *//Creating movie object*
2. var movie = {}
3. *//Movie object have a method addMovie*
4. *//This method interacts with DB*
5. *//This method make use of create(mongoose method) to insert movie object*
6. movie.addMovie = async(movieObj)=> {
7. let insertedData = await movieModel.create(movieObj)
8. if (insertedData) {
9. return insertedData
10. } else {
11. let err = new Error("Data not inserted")
12. err.status = 500
13. throw err;
14. }
15. }
16. *//Movie object to be inserted*
17. let obj = {
18. movieId: 2001,
19. movieName: "Avengers",
20. language: "English"
21. }
22. *//Wrapping movie.addMovie into a new function*
23. addMovie = async(objectToBeInserted) => {
24. try {
25. let data = await movie.addMovie(objectToBeInserted)
26. console.log(data);
27. } catch (err) { console.log(err.message) }
28. }
29. *//Calling addMovie function*
30. addMovie(obj)

After successful insertion, you will get the following output in the console



Here, showDetails and bookings get created automatically since we assigned a default value.

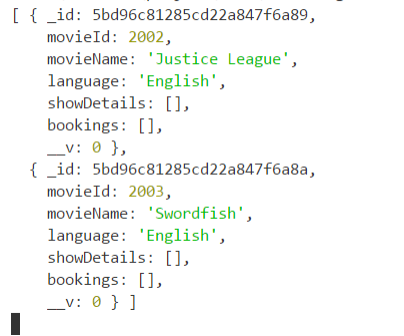
In order to insert multiple documents into a collection, we have a method called insert many(). The syntax for inserting multiple documents into a collection is:

1. Model.insertMany([document1, document2]);

Let us now insert more document into our Movie collection using the below code:

1. *//Creating movie object*
2. var movie = {}
3. *//Movie object have a method multipleInsert*
4. *//This method interacts with DB*
5. *//This method make use of insertMany (mongoose method) to inserts array of movie object*
6. movie.multipleInsert = async(movieObj)=> {
7. let insertedData = await movieModel.insertMany(movieObj)
8. if (insertedData) {
9. return insertedData
10. } else {
11. let err = new Error("Data not inserted")
12. err.status = 500
13. throw err;
14. }
15. }
16. *//Array of movie object to be inserted*
17. let obj = [{
18. movieId: 2002,
19. movieName: "Justice League",
20. language: "English"
21. }, {
22. movieId: 2003,
23. movieName: "Swordfish",
24. language: "English"
25. }]
26. *//Wrapping movie.multipleInsert into a new function*
27. multipleInsert = async(objectToBeInserted) => {
28. try {
29. let data = await movie.multipleInsert(objectToBeInserted)
30. console.log(data);
31. } catch (err) { console.log(err.message) }
32. }
33. *//Calling multipleInsert method*
34. multipleInsert(obj)

After successful insertion, you will get the following output in the console



# ****Retrieve all documents from a collection****

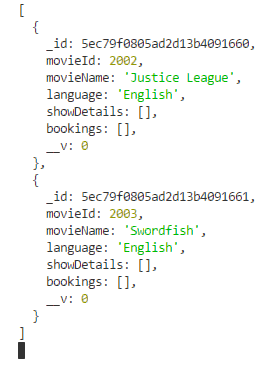
Documents can be retrieved through, find, findOne and, findById These methods are executed on your Models.

 Lets us now retrieve all the documents that we have inserted into our Movie Collection.

Here we will be making use of Async-Await to do the read operation.

1. *//This method make use of find (mongoose method) to read the array of movie objects*
2. movie.find = async () => {
3. let movieData = await movieModel.find();
4. if (movieData) {
5. return movieData;
6. }
7. else {
8. let err = new Error("No record found");
9. err.status = 404;
10. throw err;
11. }
12. }
13. *//Wrapping movie.find into a new function*
14. find = async () => {
15. try {
16. let data = await movie.find();
17. console.log(data);
18. } catch (err) {
19. console.log(err.message);
20. }
21. }
22. *//Calling find function*
23. find();

On successful retrieving of the document, you will get the following output in the console

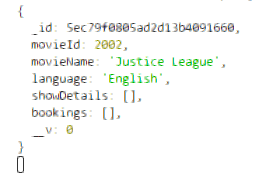


# ****Retrieving data based on the condition****

Let us now retrieve movie details based on movieId from the Movie Collection using the below code:

1. *// Find movies based on id*
2. *//This method make use of findOne (mongoose method) to read the particular movie object*
3. movie.getMovie = async (movieId) => {
4. let movieDetails = await movieModel.findOne({ movieId: movieId });
5. if (movieDetails) { return movieDetails }
6. else {
7. let err = new Error("No record found");
8. err.status = 404;
9. throw err;
10. }
11. }
12. *//Wrapping movie.getMovie into a new function*
13. findById = async () => {
14. try {
15. let data = await movie.getMovie(2002);
16. console.log(data);
17. }
18. catch (err) {
19. console.log(err.message);
20. }
21. }
22. *//Calling findById function*
23. findById();

Now you will get the movie details for the entered movieId in the console as shown below:



We can do a projection in order to retrieve only the required properties. For example, we will retrieve only movieName and language for the given movieId using the below code:

1. *//Find movies based on id and return specific values*
2. *//This method make use of findOne (mongoose method) to read the details of movieName and language of the movie object*
3. movie.getMovie = async (movieId) => {
4. let movieDetails = await movieModel.findOne({ movieId: movieId }, { \_id: 0, movieName: 1, language: 1 })
5. if (movieDetails) { return movieDetails }
6. else {
7. let err = new Error("No record found");
8. err.status = 404;
9. throw err;
10. }
11. }
12. *//Wrapping movie.getMovie into a new function*
13. findById = async () =>{
14. try{
15. let data = await movie.getMovie(2002);
16. console.log(data);
17. }
18. catch(err){
19. console.log(err.message);
20. }
21. }
22. *//Calling findById function*
23. findById();

The following will be the output in the console:

https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_01302632538167705616623_shared/web-hosted/assets/findById2.PNG

# 

# Updating collection

Now we have the details of the movies that will be screen in the multiplex, let us add the show details for the movies by updating the collection.

We will update the show details of the movie collection using the below code:

1. *//Update details*
2. *//This method make use of updateOne (mongoose method) to update the showDetails array of movie objects*
3. movie.addShows = async (showObj) => {
4. let updatedData = await movieModel.updateOne({ movieId: 2002 }, { $push: { showDetails: showObj } }, { runValidators: true })
5. if (updatedData) { return updatedData }
6. else {
7. let err = new Error("Data not updated");
8. err.status = 500;
9. throw err;
10. }
11. }
12. *//Object to be added to the array*
13. let showObj = {
14. showId: 102,
15. fare: 180,
16. availableSeats: 20,
17. movieId:2002
18. }
19. *//Wrapping movie.addShows into a new function*
20. update = async () =>{
21. try{
22. let data = await movie.addShows(showObj);
23. console.log(data);
24. }
25. catch(err) {
26. console.log(err.message);
27. }
28. }
29. *//Calling update function*
30. update();

If the document was successfully updated the following will be the output in the console:

https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_01302632538167705616623_shared/web-hosted/assets/update.PNG

**Note**: The validators by default are not triggered on update() or findOneAndUpdate() operations. They need to be enabled by setting the runValidators option to true

# Deleting a collection

The only remaining CRUD operation is deleting a collection. Let us see how to delete the details of a movie through the below code:

1. *//Delete details based on movie name*
2. *//This method make use of deleteOne (mongoose method) to delete the particular movie object*
3. movie.removeShow = async (movie) => {
4. let deletedData = await movieModel.deleteOne({ movieName: movie })
5. if (deletedData) { return deletedData }
6. else {
7. let err = new Error("Could not delete data");
8. err.status = 500;
9. throw err;
10. }
11. }
12. *//Wrapping the movie.removeShow into a new function*
13. deleteShow = async () => {
14. try {
15. let data = await movie.removeShow('Swordfish');
16. console.log(data);
17. }
18. catch (err) {
19. console.log(err.message);
20. }
21. }
22. *//Calling deleteShow function*
23. deleteShow();

If the deletion is successful, we will get the following output:

https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_01302632538167705616623_shared/web-hosted/assets/delete.PNG

To know more on querying using Mongoose, refer [Mongoose docs](https://mongoosejs.com/docs/queries.html)

Till now we have been placing the code that interacts with the database and the functions that trigger these interactions in the same file.

For ease of coding and debugging, let us modularize the entire code.

We will place the below code to establish a connection to a database and creating a schema in 'src/models/connection.js' file.

1. const mongoose = require('mongoose');
2. mongoose.Promise = global.Promise;
3. mongoose.set('useCreateIndex', true)
4. mongoose.connect('mongodb://localhost:27017/MultiplexDB', { useNewUrlParser: true , useUnifiedTopology: true })
5. .catch((error) => {
6. let err = new Error("Could not connect to Database")
7. err.status = 500;
8. throw err;
9. })
10. const schema = {
11. "movieId": {
12. required: [true, 'Required field'],
13. type: Number,
14. unique: true
15. },
16. "movieName": {
17. required: [true, 'Required field'],
18. type: String
19. },
20. "language": {
21. required: [true, 'Required field'],
22. match: [/^[A-Za-z]{3,8}$/, 'Please enter only alphabets'],
23. type: String
24. },
25. "showDetails": {
26. type: [
27. {
28. "showId": {
29. required: [true, 'Required field'],
30. type: Number
31. },
32. "movieId": {
33. required: [true, 'Required field'],
34. type: Number
35. },
36. "fare": {
37. required: [true, 'Required field'],
38. type: Number
39. },
40. "availableSeats": {
41. required: [true, 'Required field'],
42. type: Number
43. },
44. "showDate": {
45. required: [true, 'Required field'],
46. type: Date,
47. validate: [(showDate) => new Date(showDate) >= new Date(), 'Date should be greater than today']
48. }
49. }
50. ],
51. default: []
52. },
53. "bookings": {
54. type: [
55. {
56. "bookingId": {
57. required: [true, 'Required field'],
58. type: Number
59. },
60. "customerName": {
61. type: String,
62. default: "New User"
63. },
64. "bookingCost": {
65. required: [true, 'Required field'],
66. type: Number
67. },
68. "showId": {
69. required: [true, 'Required field'],
70. type: Number
71. },
72. "noOfTickets": {
73. type: Number,
74. min: [1, 'Minimum 1 ticket'],
75. },
76. "bookedAt": {
77. type: Date,
78. default: new Date().toLocaleDateString()
79. }
80. }
81. ],
82. default: []
83. }
84. }
85. let movieSchema = mongoose.Schema(schema, { collection: 'Movie', timestamps: true });
86. exports.movieModel= mongoose.model("Movie",movieSchema );

We will place the below code to interact with the database in 'src/models/db.js' file.

1. const movieModel = require('./connection').movieModel
2. var movie = {}
3. *//Insert a single data*
4. movie.addMovie = async (movieObj) => {
5. let insertedData = await movieModel.create(movieObj)
6. if (insertedData) {
7. return insertedData
8. }
9. else {
10. let err = new Error("Data not inserted")
11. err.status = 500
12. throw err;
13. }
14. }
15. *//Retrieve all data*
16. movie.find = async () => {
17. let movieData = await movieModel.find()
18. if (movieData) { return movieData }
19. else {
20. let err = new Error("No record found")
21. err.status = 404
22. throw err;
23. }
24. }
25. *//Retrieve data based on MovieId*
26. movie.getMovie = async (movieId) => {
27. let movieDetails = await movieModel.findOne({ movieId: movieId }, { \_id: 0, movieName: 1, language: 1 })
28. if (movieDetails) { return movieDetails }
29. else {
30. let err = new Error("No record found")
31. err.status = 404
32. throw err;
33. }
34. }
35. *//Insert multiple data*
36. movie.multipleInsert = async (movieObj) => {
37. let insertedData = await movieModel.insertMany(movieObj)
38. if (insertedData.length > 0) {
39. return insertedData
40. }
41. else {
42. let err = new Error("Data not inserted")
43. err.status = 500
44. throw err;
45. }
46. }
47. *//Update details*
48. movie.addShows = async (showObj) => {
49. let updatedData = await movieModel.updateOne({ movieId: 2002 }, { $push: { showDetails: showObj } })
50. if (updatedData.nModified > 0) {
51. return updatedData
52. }
53. else {
54. let err = new Error("Data not updated")
55. err.status = 500
56. throw err;
57. }
58. }
59. *//Delete details of a movie*
60. movie.removeShow = async (movie) => {
61. let deletedData = await movieModel.deleteOne({ movieName: movie })
62. if (deletedData.deletedCount > 0) {
63. return deletedData;
64. }
65. else {
66. let err = new Error("Could not delete data")
67. err.status = 500;
68. throw err;
69. }
70. }
71. module.exports = movie;

Since you have learned how to make an express application interact with a database, let us replace the static file data.json in the Express wallet application with a MongoDB database.

Step 1: Download the [ExpressWallet](https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_01302890710630400029586_shared/web-hosted/assets/ExpressWalletUpdated.zip)

Step 2: Open the application in VS Code IDE

Step 3: Open the file src/utilities/connection.js

1. const mongoose = require("mongoose")
2. mongoose.Promise = global.Promise;
3. const Schema = mongoose.Schema
4. mongoose.set("useCreateIndex", true)
5. const url = 'mongodb://localhost:27017/ExpressWalletDB'
6. let schema = {
7. "username": {
8. type: String,
9. required: true,
10. unique: true
11. },
12. "password": {
13. type: String,
14. required: true
15. },
16. "PAN": {
17. type: String,
18. required: true
19. },
20. "transactions": {
21. type: [{
22. "tid": {
23. type: Number,
24. required: true
25. },
26. "amount": {
27. type: Number,
28. min: 100,
29. required: true
30. },
31. "transactionType": {
32. type: String,
33. required: true
34. },
35. "tDate": {
36. type: Date,
37. required: true
38. }
39. }],
40. default: []
41. }
42. }
43. let accountSchema = new Schema(schema, { collection: "Account", timestamps: true })
44. let connection = {}
45. connection.getCollection = async() => {
46. try {
47. return (await mongoose.connect(url, { useNewUrlParser: true, useUnifiedTopology: true })).model("Account", accountSchema)
48. } catch (err) {
49. let error = new Error("Could not connect to database")
50. error.status = 500
51. throw error
52. }
53. }
54. module.exports = connection

Note:

* **Line 2:** Configures mongoose to make use of global promise library
* **Line 6 - 41:**Defines the structure of the data which has to be inserted into the collection
* **Line 45:**Creating a Mongoose schema
* **Line 49:** Establishing a connection between Express wallet and MongoDB database
* **Line 50:** Creating a model object for the 'Account' schema

Now that we have configured our database, let us add some data to our database.

We will persist the same data present in our data.json file to our ExpressWalletDB

Step 1: Open the file src/model/account.js and observe the below code

1. model.insertScript = async() => {
2. await collection.deleteMany();
3. let response = await collection.insertMany(initialData);
4. if (response && response.length > 0) {
5. return response.length
6. } else {
7. let err = new Error("Script insertion failed")
8. err.status = 500
9. throw new Error
10. }
11. }

Note:

* Line 2 - Deletes all the documents present in the 'Account' schema
* Line 3- Inserts data into the 'Account' schema
* Line 5 - Returns the number of documents inserted wrapped in a promise

Step 2: Open the file src/service/account.js and observe the below code

1. service.insertScript = async() => {
2. let data = await dbLayer.insertScript();
3. return data;
4. }

Note:

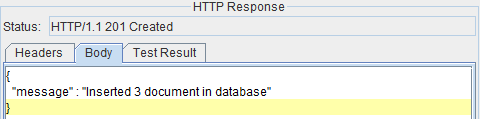
* Line 2 - Invokes the insertScript() method present in model/account.js file
* Line 3 - Return the number of documents inserted

Step 3: Open the file src/routes/routing.js and observe the below code

1. routing.get("/setupDB", async(req, res, next) => {
2. try {
3. let data = await service.insertScript();
4. if (data) {
5. res.status(201)
6. res.json({ message: "Inserted " + data + " document in database" })
7. }
8. } catch (err) { next(err) }
9. })

Note: Line 1 - When the user sends a GET request to /setupDB, the insertScript() method of service gets invoked

Step 4: Open RESTClient jar file & send a GET request to http://localhost:3000/setupDB. You will get the following response



Now we have a brief idea of how the Express wallet is structured, let us dive deep into understanding the data/control flow in the Express Wallet application

Assume the user wants to create a new account and save the details in the ExpressWalletDB collection

Step 1: Open the file src/routes/routing.js file and observe the below code

1. routing.post("/accounts", async(req, res, next) => {
2. let accountObj = req.body
3. try {
4. let accountData = await service.createAccount(accountObj);
5. res.json({ message: "Account Created Successfully" })
6. } catch (err) { next(err) }
7. })

Note:

* When the user sends a POST request to /account, then the data sent in the request body will be passed as a parameter to the createAccount() method of service.
* If createAccount() method returns a success response, the server will send a JSON response as "Account Created Successfully"
* Else if createAccount() method returns a failure response, it is handled by the catch block where the error is passed to the error handler function

Step 2: Open the file src/service/account.js and observe the below code

1. service.createAccount = async(accountObj) => {
2. validator.validatePAN(accountObj.PAN);
3. let data = await dbLayer.getUser(accountObj.username);
4. if (data) {
5. let err = new Error("User already exists")
6. err.status = 406
7. throw err
8. } else {
9. let accountData = await dbLayer.createAccount(accountObj);
10. if (accountData) {
11. return accountData
12. } else {
13. let err = new Error("Account not created")
14. err.status = 500
15. throw err;
16. }
17. }
18. }

Step3: Open the file src/model/account.js and observe the below code

1. let createConnection = async() => {
2. collection = await connection.getCollection();
3. }
4. model.getUser = async(username) => {
5. let userData = await collection.findOne({ username: username }, { \_id: 0, username: 1, password: 1 })
6. return userData;
7. }
8. model.createAccount = async(accountObj) => {
9. let accountData = await collection.create(accountObj)
10. return accountData ? true : false;
11. }
12. createConnection();

Note:

* **Line 2**-   getCollection of src/utilities/connection.js is invoked which will get you the model object, through which you can perform CRUD operation
* **Line 5** -  Same find operation which we will be doing through MongoDB driver can be performed through the model instance of mongoose
* **Line 6** - If a matching document is found, the same document is sent back to service
* **Line 10** - A new document will be inserted through create a method of mongoose, which is similar to insertOne of MongoDB

Now you have completely seen one flow of control between the Express application and database, you can now trace the flow of control/data for other routes in the Express Wallet application.