TASK

Database Interaction II

Visit our website

Introduction

WELCOME TO THE SECOND DATABASE INTERACTION TASK!

In the previous task, you successfully used Mongo to perform CRUD operations on your database. However, you can also create *programs and functions* to perform CRUD operations on your database. In this task, you will learn to write code using Node to interface with your MongoDB.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

Our team is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

MONGODB AND NODE.JS

As web developers, we want to be able to create and modify databases using code, not just using an administrative shell like mongo. In this task, we will be creating code that will allow Node.js to interact with MongoDB.

To be able to do this, it is important to understand the architecture of what we are going to be creating. This is illustrated in the image below:

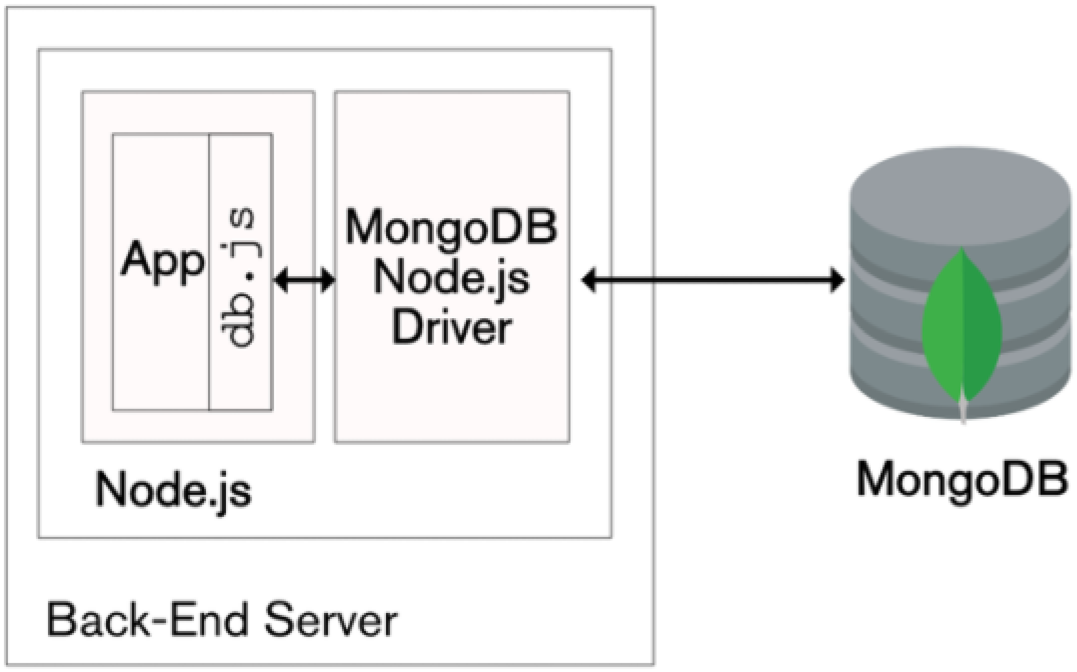


Image source:
[***https://www.mongodb.com/blog/post/the-modern-application-stack-part-2-using-mongodb-with-nodejs***](https://www.mongodb.com/blog/post/the-modern-application-stack-part-2-using-mongodb-with-nodejs)

As you can see in the image, we have a back-end server running Node.js. Using Express, we create an **app.js** module that handles all the routing and logic required for our web server. One of the things that the app must be able to do for data-driven apps is to communicate with the database. We can write the code (e.g. **db.js** in the image above) for communicating with the database in two possible ways:

- by writing code that uses MongoDB's Node.js driver to interact with MongoDB directly or
- by writing code that uses Mongoose. Mongoose is a library that sits on top of MongoDB's Node.js driver and abstracts some of the boilerplate code for you.

In this task, you will learn how to use Mongoose. See the additional reading that accompanies this task for instructions to use MongoDB's Node.js driver directly.

CREATE CODE USING MONGOOSE

When working with the MongoDB driver, you are required to write a lot of boilerplate code for database manipulation. Mongoose is a library that sits on top of the MongoDB driver and abstracts some of the boilerplate code for you. It includes built-in typecasting, validation, query building and business logic hooks. Therefore, Mongoose can make it easier to write code for manipulating data in databases.

Mongoose is an Object Data Model (ODM). An ODM is a tool that allows the programmer to treat documents stored in databases as JavaScript objects.

To use Mongoose, do the following:

- Step 1: Install Mongoose.
  From the command line interface, change to the project directory of the Express project that you want to use to manipulate your database and type the following instruction:
  `npm install mongoose`

  This will install Mongoose and all its dependencies, including the MongoDB driver.

- Step 2: Create a schema
  Although MongoDB is schemaless, Mongoose works with schemas. Remember, a schema describes what data is in a database and how it is organised and structured.

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

let BlogSchema = mongoose.Schema({
  title:{
      type:String,
      required:true
  },
  text:{
      type:String,
      required:true
```

```
   },
   author:{
       type:String,
       required:false,
       default:"anonymous"
   },
   createDate:{
       type:Date,
       required:false,
       default: Date.now
   }
});

// module.exports makes the model available outside of your module

// The first argument for mongoose.model should be the name of the
// document in your MongoDB collection (remember that spelling
// is important, this includes casing)
module.exports = mongoose.model('Blog', blogSchema);
```

As you can see in the example above, the schema describes the data and the type of data that will be stored for each document in a MongoDB collection. You must require Mongoose and create a variable to hold the schema object, before you can create the schema.

Models are special constructors that are compiled based on the schema you have defined. According to **Mongoose's official documentation**:

"Instances of these models represent documents which can be saved and retrieved from our database. All document creation and retrieval from the database is handled by these models."

Below is an example of how you create a model using the model() method. The two arguments you pass to this method are:

- The name of the model
- the schema object you created in the previous step

```
let Blog = mongoose.model('Blog', blogSchema);
```

It is good practice to create a directory called "models" in the root directory of your express app in which you define your schemas and create your models.

- Step 3: Create a controller file to perform CRUD operations

  In your project directory, create another directory called “controllers”. In this directory, create a file called **blog.controller.js**. In this file, you will create all the code needed to perform CRUD operations using Mongoose.

  To create a document with Mongoose, use the `save()` function as shown below:

```
const Blog = require('../models/models.js');

//function takes HTTP request and response
exports.create = function(req, res) {    // Create and save a new blog
    let blogModel = new Blog({title: 'Example Code',
    text: 'This is to demonstrate how to add data to a database using
Mongoose',
    author: 'HyperionDev'});

        blogModel.save(function(err,data) {
            if(err) {
                console.log(err);
                res.status(500).send({message: "Some error occurred while
creating the blog."});
            }
            else {
                console.log(data);
                res.send('The blog has been added');
            }
    });
};
```

  To read or query documents, use the `find()` method as shown below:

```
exports.findAll = function(req, res) {
    Blog.find(function(err, blogs){
        if(err) {
            console.log(err);
                res.status(500).send({message: "Some error occurred while
retrieving blogs"});
        }
        else {
            res.send(blogs);
        }
    });
}
```

It is very important that you are able to build queries to meet your needs. Be sure to consult **this guide** for extra guidance.

To update a document use the `update()`, `updateOne()`, `updateMany()` or `findOneAndUpdate()` methods. See example below. For more information, see **here**.

```
exports.updateByAuthor = function(req, res) {
    let query = {author: 'HyperionDev'};
    Blog.findOneAndUpdate(query, {author: 'HyperionDev'},
    {new: true}, function(err, doc){
        if(err) {
            console.log("Something went wrong when updating data.");
        }
            res.send("Updated");
    });
}
```

To delete a document, use the `remove()` function as shown below. All documents that meet the specified condition will be removed from the collection. In the example below, all documents with empty strings values for blog titles will be removed.

```
exports.deleteBlogsByAuthor = function(req, res) {
    Blog.remove({author:'HyperionDev'}, function(err){
        if(err) return handleError(err);
        console.log("Blogs removed");
        res.send("Blogs removed");
    });
}
```

- Step 4 : Connect to the database and execute appropriate CRUD operations
  The code below can be used to connect to the database.

```
let mongoose = require('mongoose');
const uri =
'mongodb://hyperionDB:password@hyperion-shard-00-00-f78fc.m...';
mongoose.Promise = global.Promise;

mongoose.connect(uri, {
  useMongoClient: true,
  dbName: 'Blogs' // Connect to the Blogs database.
});

mongoose.connection.on('error', function() {
    console.log('Could not connect to the database. Exiting now...');
    process.exit();
});

mongoose.connection.once('open', function() {
    console.log("Successfully connected to the database");
})
```

To be able to connect to the database using Mongoose, we first require Mongoose. The instruction **`mongoose.connect()`** is then used to connect to the database. The argument passed into the **`connect()`** method is the connection string for your database. Remember that you can get this connection string from Atlas (or Compas), as you have done previously.

Take note:

Certain passwords may lead to problems with your connection string. Passwords with special characters like @ can lead to errors. This is because the connection string is a Uniform Resource Identifier (URI). A URI is composed of a limited set of characters consisting of digits, letters, and a few graphic symbols. Characters that aren't recognised can be included by using percentage-encoding. A percent-encoded character is encoded using 3 other characters: "%" followed by the ASCII code that represents that character. For example, %20 is used to represent a single space character (" ") and %40 is used to represent an "at" symbol (@). For more information about how URIs are used see here. For a list of ASCII codes, see **here**.

Based on the above information, we recommend you use an alphanumeric password for accessing MongoDB. If you choose to automatically generate a password on Atlas, it will generate a password that will work with your connection string without any changes. If you choose to use a password with special characters, however, you will have to use percentage-encoding in your connection string.

E.g. `const url = 'mongodb+srv://hyperiondevDB:myp@ssword@hyperiondev-78c.mongodb.net/test';` could be replaced with

```
const url =
'mongodb+srv://hyperiondevDB:myp%40ssword%40hyperiondev-78c.mongodb.net/test';
```

To use the CRUD operations, call the appropriate functions from the model you have created.

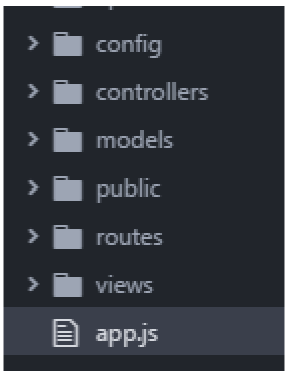
MONGODB AND EXPRESS

The MERN stack is a stack of tools, not a web framework. It provides a highly flexible, un-opinionated approach to web development. As we have said before, this has its pros and cons. When using the MERN stack, there is no predefined way of structuring your project. This enables you to have complete control over your project structure. However, this can also lead to challenges, since developers new to web development may have no idea of how to structure their project, potentially leading to projects that are difficult to troubleshoot, maintain and develop as a team.

Since the MERN stack's un-opinionated approach to web development is one of its key strengths, it would be inappropriate to give you rigid rules regarding what your project structure should look like. However, some guidelines are in order at this stage.

Although which directories you have and what you choose to name them is up to you, it is recommended that your project is based on a recognised software architecture pattern. Remember that in a previous task, we learned that architecture patterns are well-researched and provide ways of thinking about and approaching web development that is known to work. We considered layered architecture patterns, event-driven architecture patterns, and microservice architecture patterns. The most commonly used software architecture pattern is a layered architecture pattern known as the MVC (model-view-controller) pattern.

A project structure that is based on the MVC architecture pattern could look something like the one shown in the image below:



Notice that we have routes, models, controllers, and views directories.

You have already learnt how we use *models* in Mongoose. *Controllers* are the JavaScript files that contain all the methods and functions which will handle your data. This includes not only the methods for creating, reading, updating and deleting items but also any additional business logic. There should be at least one model file and one controller file for each type of data in your database.

If you don't want to create all your own boilerplate code and directory structure for each project, there are several generators that you can use. For example, **the Node Express Mongo Stack generator**. However, it takes a while to understand the boilerplate code and project structure. Ultimately you are the one who will have to decide when to use a generator and which generator to use.

Extra resource

For more information about working with Mongoose, see Chapter 24 and Chapter 87 of the additional reading that accompanies this task. This free e-book is written by the 'beautiful people of Stack Overflow'

SPOT CHECK 1

Let's see what you can remember from this section.

1. What does Mongoose do to make writing code to manipulate a database easier for you?
2. What is an ODM?

Instructions

- See the example code that accompanies this task before you attempt this compulsory task. This example uses Mongoose for database manipulation.

- The Express-related tasks involve creating apps that need some modules to run. These modules are located in a folder called 'node_modules', which is created when you run the following command from your command line: 'npm install' or similar. Please note that this folder typically contains thousands of files which, if you're working directly from Dropbox, has the potential to ***slow down Dropbox sync and possibly your computer***. As a result, please follow this process when creating/running such apps:
    - Create the app on your local machine (outside of Dropbox) by following the instructions in the compulsory task.
    - When you're ready to have a reviewer review the app, please **delete the node_modules** folder.
    - Compress the folder and upload it to Dropbox.
    - Your reviewer will, in turn, decompress the folder, install the necessary modules, and run the app from their local machine.

Compulsory Task 1

Follow these steps:

- Create a full-stack web application in a project directory called “task 7”. Create the back-end of the application using Express and the front-end using React. You should create a MongoDB that stores information about cars in a collection called cars.

- Your application should allow one to:
    - Add a car to the cars collection.
    - Update information about a single car.
    - Update information about more than one car.
    - Delete a specific document.
    - List all the information for all cars in your database.
    - List model, make, registration number, and current owner for all cars older than 5 years.

- Ensure that for the back-end of your application you:
    - Install Mongoose.
    - Create 2 directories in your project directory called "models" and "controllers".
    - Write all the code needed to perform the necessary CRUD operations for your application.

If you are having any difficulties, please feel free to contact our specialist team **on Discord** for support.

Things to look out for:

1. Make sure that you delete 'Node_modules' before submitting the code.
2. Review the tasks that you have already completed that show you how to "Create a custom API with Express" and how to get React and Express to work together ("Full stack with React and Express").

Completed the task(s)?

Ask an expert to review your work!

Review work

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.

SPOT CHECK 1 ANSWERS

1. It includes built-in typecasting, validation, query building and business logic hooks.
2. An Object Data Model (ODM) is a tool that allows the programmer to treat documents stored in databases as JavaScript objects.