Hyperiondev

TASK

Database Interaction With MongoDB and Mongoose

Visit our website

Introduction

WELCOME TO THE DATABASE INTERACTION WITH MONGODB AND MONGOOSE TASK!

In this task, we will explore the essential functionalities of Mongo (MongoDB's administrative shell) for creating databases and collections, which are foundational skills for any developer working with NoSQL databases. You will learn to perform CRUD operations – creating, reading, updating, and deleting documents from collections – essential for effective data management in real-world applications. Furthermore, we will discuss how to integrate MongoDB with Node.js and Express, as well as the use of Mongoose to streamline the development process. Understanding these concepts will not only enhance your technical proficiency but also equip you with the tools necessary to build scalable and efficient web applications.

MONGO SHELL BASIC COMMANDS

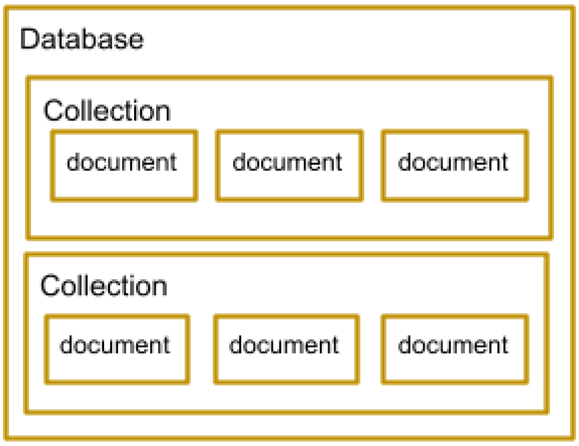
In the previous task, you created a MongoDB database. In this task, you are going to learn how to manipulate your database using CRUD operations. CRUD is simply an acronym for the four basic operations used to manipulate a database: **C**reate, **R**ead, **U**pdate, and **D**elete.

Before considering CRUD though, you first need to learn how to use some basic Mongo shell commands:

- `show dbs;`
  This command is used to list all the databases in your cluster.

- `use db_name;`
  This command selects a database, which is necessary for manipulation. If the specified database does not exist, this command will create it. Be aware that naming rules vary by operating system, and the following special characters should be avoided in database names: `/\. "$*<>:|?`

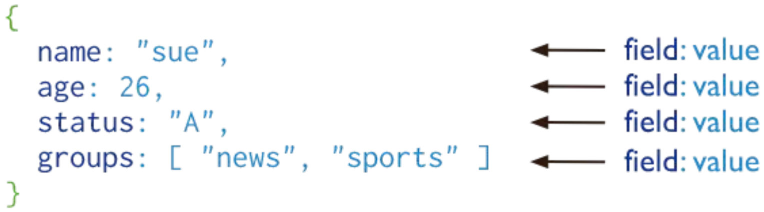
- `show collections;`
  This command shows all the collections in the previously selected database. A collection is a grouping of related documents and is equivalent to a table in a relational database. If you were creating a database for a company that sells cars, for example, you could have a car collection, customer collection, and shop collection.



- `db.dropDatabase();`
  This command is used to delete a database. Before you use this command, though, you need to make sure you have selected the database you would like to delete using the `use` command shown previously. If you are unsure which database you are currently working with, you can simply type the command `db` into your Mongo shell and it will display the name of the database you selected.

CRUD OPERATIONS

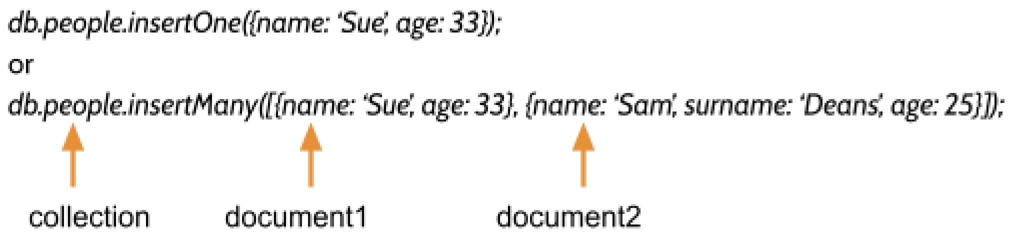
Create: After creating a database, you need to insert collections and documents. A collection groups BSON documents, which are similar to JSON but allow for richer type information. For instance, BSON recognises "date" as a type, while JSON does not. Documents store data in key-value pairs, and each document in a collection can have a slightly different structure.



```
{
   name: "sue",
   age: 26,
   status: "A",
   groups: [ "news", "sports" ]
}
```

BSON Documents. (MongoDB).

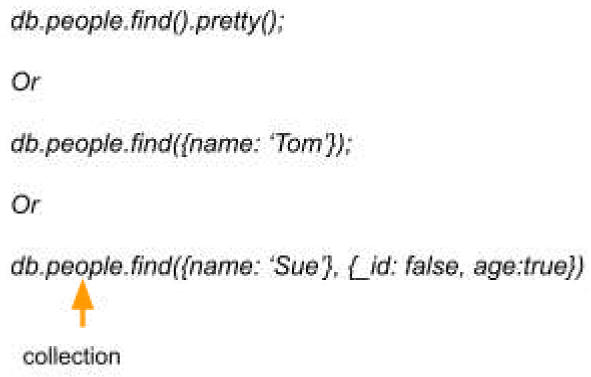
To insert a new document into a collection, use the `insertOne` or `insertMany` commands as shown below:



```
db.people.insertOne({name: ‘Sue’, age: 33});
or
db.people.insertMany([{name: ‘Sue’, age: 33}, {name: ‘Sam’, surname: ‘Deans’, age: 25}]);
```

Note that not all the documents in your collection need exactly the same format. See how the BSON document for Sue only stores `name` and `age`, whereas the BSON document for Sam stores `name`, `surname`, and `age` information. If the collection you specify in the `insertOne` or `insertMany` command does not exist, MongoDB will create the collection for you, because every document needs a unique identifier. In MongoDB, the field `_id` must be specified for each document. If you don't explicitly specify an `_id` field when you insert a document, MongoDB will automatically generate one for you.

Read: Once your collection contains documents, you will want to read those documents. We do this using the `find()` command as illustrated below.



```
db.people.find().pretty();

Or

db.people.find({name: ‘Tom’});

Or

db.people.find({name: ‘Sue’}, {_id: false, age:true})
```

The `pretty()` method is used to make the output more readable. The `find()` command can be used without it. When you specify `find()` without any arguments, all documents in the collection will be returned.

You can also pass one or more key-value pairs as shown in the second code example, which will then find all documents that match the criteria specified. For instance, in the aforementioned second code example, the find command will find all documents where the person has the `name` “Tom”, as well as all the information contained in all those documents, including the `_id` for each document.

You often do not want to read or display all the information in a document. For instance, you may not want a user to see the `_id` value for each document. You can, therefore, also pass a second argument set to the **`find()`** method, as shown in the third example above. The arguments in the second set of curly brackets (**`{_id: false, age: true }`**) specify which fields will be retrieved and which won't be. In this example, the **`_id`** field won't be output, but the **`age`** field will.

You can also use the **`findOne()`** method instead of the **`find()`** method. **`findOne()`** will return only the first document that matches the specified criteria.

Update: The **`updateOne()`** and **`updateMany()`** methods are used to modify documents within a collection in MongoDB. The **`updateOne()`** method modifies the first document that matches the query. The **`updateMany()`** method modifies all documents that match the query. The **`$set`** operator is used to specify the fields that should be updated.

```
// Update a single document with specific fields
db.people.updateOne({ name: 'Sue' }, { $set: { age: 34, name: 'Sue' } });

// Update a single document with a specific field
db.people.updateOne({ name: 'Sue' }, { $set: { age: 34 } });

// Update multiple documents with a specific field
db.people.updateMany({ name: 'Sue' }, { $set: { name: 'Susan' } });
```

These methods offer more control and are recommended for updating documents in modern MongoDB practices compared to the deprecated **`update()`** method.

If you specify **`updateOne()`** or **`updateMany()`** without using the **`$set`** keyword, the entire document will be updated. However, if you use the **`$set`** keyword, only the field specified will be updated and the other fields will remain the same.

Be careful when you use the update command! Fields not defined in the update section will be removed from the document. Similarly, if you specify fields that were not previously part of your document with the update command, new fields will be added to your document. Let's look at this scenario play out below:

Let's assume we have a MongoDB collection named "**`users`**" with the following document:

```
{
    "_id": 1,
    "name": "Alice",
    "age": 25,
    "email": "alice@example.com"
}
```

If we use an update command without properly specifying all existing fields, we might inadvertently remove some fields. For example:

```
db.users.updateOne({_id: 1}, {$set: {age: 26}});
```

The document will then have missing fields as below:

```
{
    "_id": 1,
    "age": 26
}
```

Delete: To remove documents from a collection, you can utilise the `deleteMany()` and `deleteOne()` methods, as demonstrated in the code examples below.

```
// Delete all documents in the collection
db.people.deleteMany({});

// Delete documents with a specific name
db.people.deleteMany({ name: 'Sue' });

// Delete a single document with a specific name
db.people.deleteOne({ name: 'Sue' });
```

When employing the `deleteMany()` method, you can clear out all documents within the collection by omitting any filter criteria (as depicted in the first example). If you pass one argument (as in the second example), all documents with the specified criteria will be removed. If you intend to remove just one document that matches the specified criteria, you can use the `deleteOne()` method (as shown in the third example).

Extra resource

For more information about performing CRUD operations with MongoDB, see the **official online documentation**.

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 such as Mongo. We will now 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:

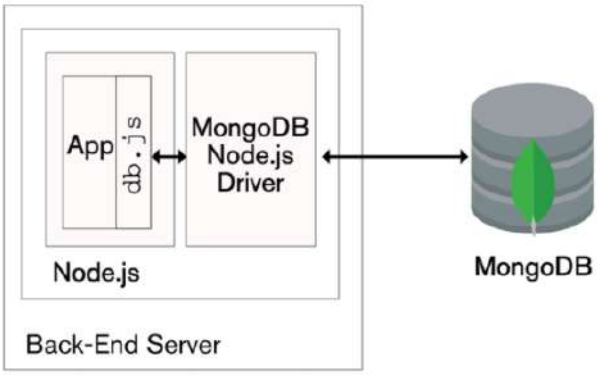


Image source (MongoDB)

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 necessary functionalities for data-driven apps is to be able 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.

Extra resource

Follow this link for quick-start instructions on how to use **MongoDB's Node.js driver**. This is an additional reading that can help you understand how Mongoose simplifies things, but it is not necessary to progress with using Mongoose.

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.

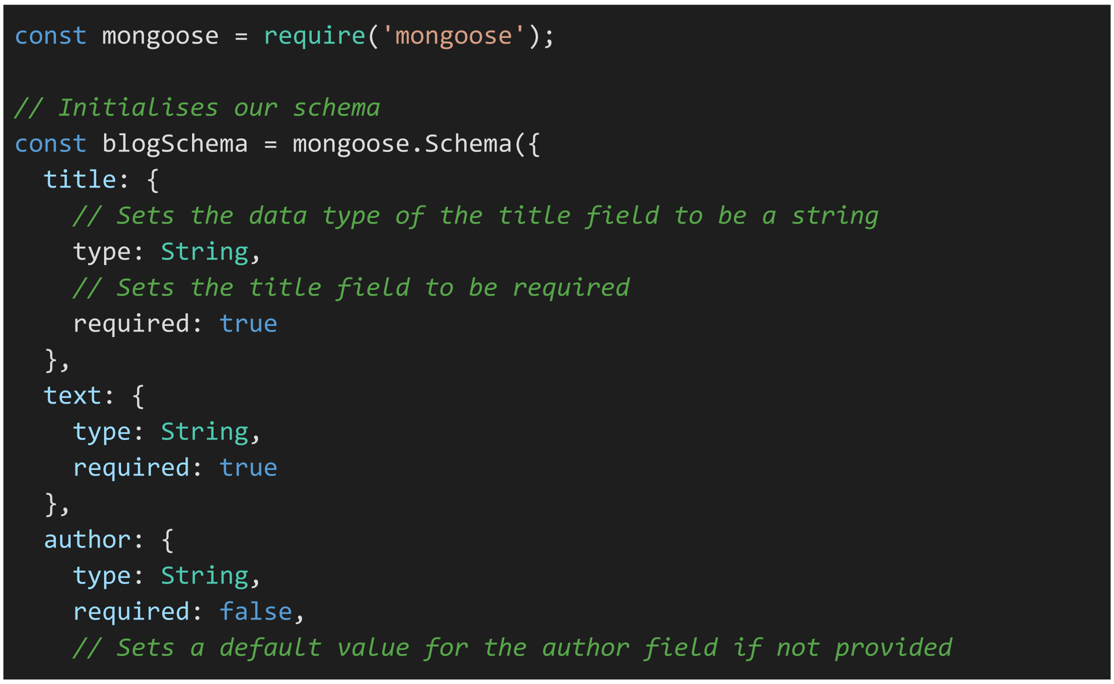
To use Mongoose, do the following:

- **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');

// Initialises our schema
const blogSchema = mongoose.Schema({
  title: {
    // Sets the data type of the title field to be a string
    type: String,
    // Sets the title field to be required
    required: true
  },
  text: {
    type: String,
    required: true
  },
  author: {
    type: String,
    required: false,
    // Sets a default value for the author field if not provided
```

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

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

/* The first argument for the mongoose.model should be the name of the
document in your MongoDB collection (remember that spelling is
Important, and that 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

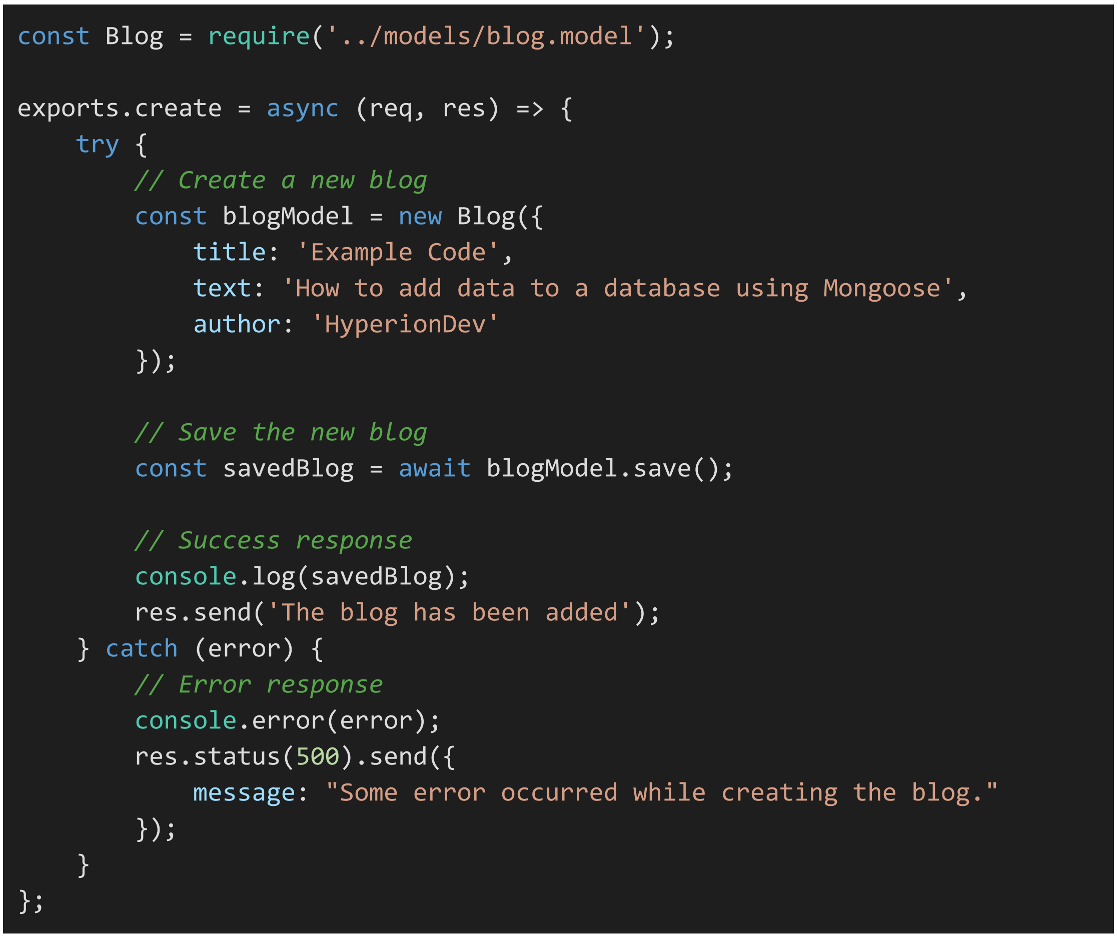
```
let Blog = mongoose.model('Blog', blogSchema);
```

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

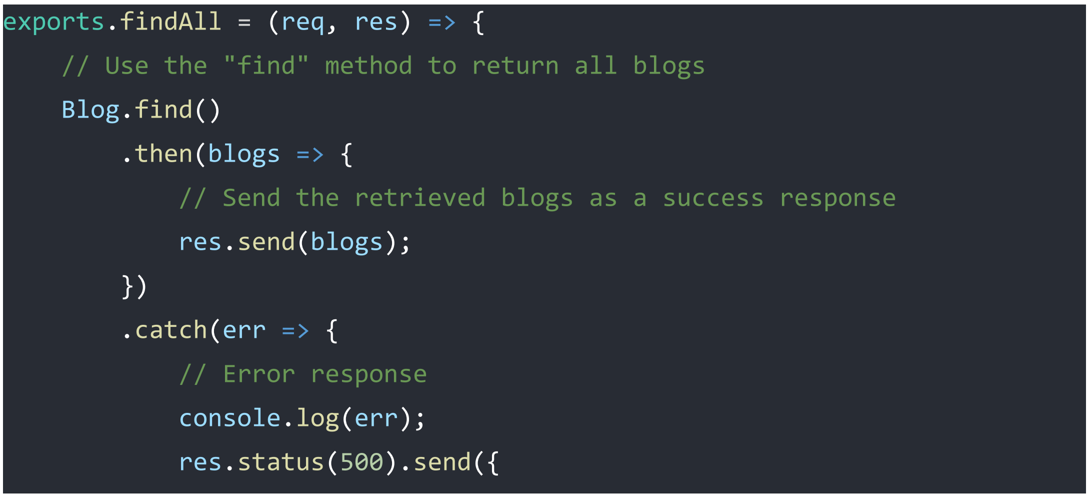
In your project directory, create another directory called “controllers”. In this directory, create a file called **blog.controller.js**. In this file, you will create all the code needed to perform CRUD operations using Mongoose.

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

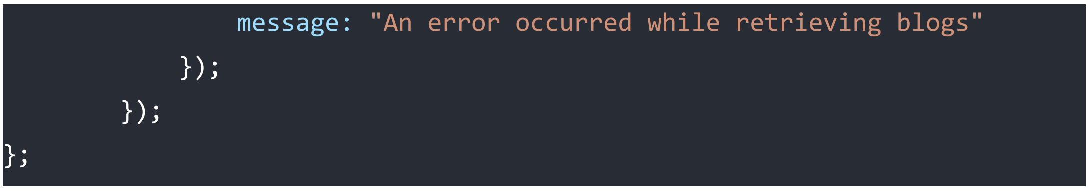


```
const Blog = require('../models/blog.model');

exports.create = async (req, res) => {
    try {
        // Create a new blog
        const blogModel = new Blog({
            title: 'Example Code',
            text: 'How to add data to a database using Mongoose',
            author: 'HyperionDev'
        });

        // Save the new blog
        const savedBlog = await blogModel.save();

        // Success response
        console.log(savedBlog);
        res.send('The blog has been added');
    } catch (error) {
        // Error response
        console.error(error);
        res.status(500).send({
            message: "Some error occurred while creating the blog."
        });
    }
};
```

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



```
exports.findAll = (req, res) => {
    // Use the "find" method to return all blogs
    Blog.find()
        .then(blogs => {
            // Send the retrieved blogs as a success response
            res.send(blogs);
        })
        .catch(err => {
            // Error response
            console.log(err);
            res.status(500).send({
```

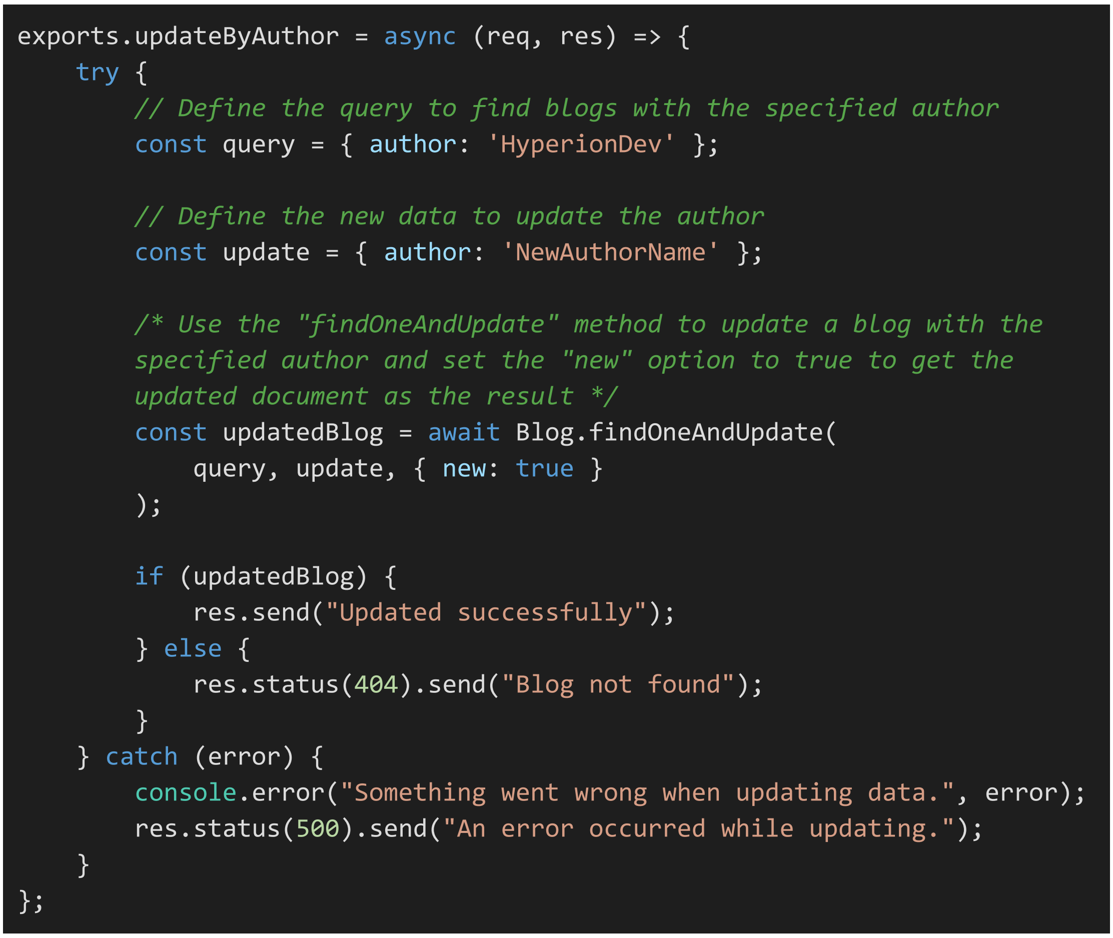


```
                message: "An error occurred while retrieving blogs"
            });
        });
};
```

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

To update a document, use the `updateOne()`, `updateMany()`, or `findOneAndUpdate()` methods. For more information on the various methods available in Mongoose and how they are used, see the documentation **here**.

See the example below:



```
exports.updateByAuthor = async (req, res) => {
    try {
        // Define the query to find blogs with the specified author
        const query = { author: 'HyperionDev' };

        // Define the new data to update the author
        const update = { author: 'NewAuthorName' };

        /* Use the "findOneAndUpdate" method to update a blog with the
        specified author and set the "new" option to true to get the
        updated document as the result */
        const updatedBlog = await Blog.findOneAndUpdate(
            query, update, { new: true }
        );

        if (updatedBlog) {
            res.send("Updated successfully");
        } else {
            res.status(404).send("Blog not found");
        }
    } catch (error) {
        console.error("Something went wrong when updating data.", error);
        res.status(500).send("An error occurred while updating.");
    }
};
```

To delete documents, use the `deleteMany()` function as shown below. All documents that meet the specified condition will be removed from the collection. In the example below, all documents with the author name: “NewAuthorName” will be removed:

```
exports.deleteBlogsByAuthor = async (req, res) => {
    try {
        // Remove all blogs with the specified author name
        const deleteResult =
            await Blog.deleteMany({ author: 'NewAuthorName' });

        if (deleteResult.deletedCount > 0) {
            res.send("Successfully deleted all blogs from author.");
        } else {
            res.send("Author not found...");
        }
    } catch (error) {
        console.error("An error occurred while removing blogs.", error);
        res.status(500).send("An error occurred while removing blogs.");
    }
};
```

- **Step 4**: Connect to the database and execute appropriate CRUD operations

The code below can be used to connect to the database.

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

// Replace the uri string with your MongoDB deployment's connection
string. You can get it from your Atlas cluster.
const uri =
'mongodb://hyperionDB:password@hyperion-shard-00-00-f78fc.m...';


// Connect to the database
mongoose.Promise = global.Promise;
mongoose.connect(uri, { useNewUrlParser: true }).then(
  () => { console.log('Successfully connected to the database!') },
  err => { console.log('Could not connect to the database...' + err) }
);
```

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 Compass), 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 consists of digits, letters, and a few symbols. Unrecognised characters can be included using percentage encoding, which replaces them with three encoded 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**.

We recommend using an alphanumeric password for MongoDB access. If you generate a password on Atlas, it will work with your connection string. For passwords with special characters, you will need to use percentage encoding in the connection string.

e.g. `const url =`

```
'mongodb+srv://hyperiondevDB:myp@ssword@hyperiondev-78c.mongodb.net/test';
```

could be replaced with ...

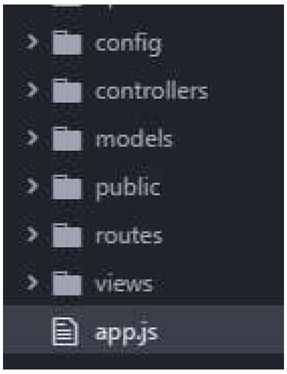
```
const url =
'mongodb+srv://hyperiondevDB:myp%40ssword%40hyperiondev-78c.mongodb.net/test';
```

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

MONGODB AND EXPRESS

The MERN stack is a flexible, un-opinionated toolset, not a web framework. While it gives you control over project structure, this can be challenging for new developers, leading to hard-to-maintain projects. Since the MERN stack's flexibility is its strength, there are no rigid rules for structuring projects. However, it's recommended to follow a recognised software architecture pattern, such as the **model-view-controller** (MVC) pattern, which is widely used and reliable.

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



```
> config
> controllers
> models
> public
> routes
> views
  app.js
```

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

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 do not want to create your own boilerplate code and directory structure for each project, there are several generators that you can use, such as **the Node Express Mongo Stack generator**. However, it takes a while to understand the boilerplate code and project structure created by such generators. Ultimately, you are the one who will have to decide when to use a generator and which generator to use.

Instructions

Read and run the accompanying example files provided with this task to become more familiar with the concepts covered in the task before doing the labs. The examples use Mongoose for database manipulation.

For comprehensive guidance on setting up, running, and testing the application's endpoints, please consult the **README.md** file.

When creating Express and React apps, essential modules are stored in the `node_modules` directory, generated by running `npm install` (or `yarn install`). This step is crucial for app functionality. Remember to delete the `node_modules` folder before submitting, as it can slow down your computer.

Practical Task 1

Follow these steps:

- Create a word document called **taskCRUD** (you can use either Microsoft Word or Google Docs). Add screenshots of your command line interface (CLI) to provide evidence that you have successfully carried out every instruction specified in this task.
- After adding all the required information and screenshots, save your Word or Google document as a PDF file in your task folder.

- In the database you created in your previous task, do the following using the Mongo shell:
    - Add a collection called “cars” and add at least five documents to your collection. Assume that you will have some of the following information for each car you add to your collection: The model, make, colour, registration number, owner, and address.
    - Include another document that contains the following information:
        - Model: 2005
        - Make: Ford Fiesta
        - Owner: Sue Bailey
        - Registration: ABC 123 GP
        - Address: 13 Main Road, Johannesburg, South Africa.
- Once you have added all the cars to your database, display all the documents you have added.
- Assume that Sue Bailey moves. Update her address to 21 Maureen Street, Bluewater Bay, Port Elizabeth, South Africa.
- Now assume that Sue marries and her surname changes to Smith. Update your database accordingly.
- Display all cars older than five years.
- Create another document where there is another Ford Fiesta owned by someone called Sue Smith.
- Our original Sue Smith (Sue Bailey) has her car scrapped. Remove the document that described her car from your database. Be sure to remove **only** her document!
- Imagine that you would like to store all the previous owners for a specific car. Add a document to a database that does this. The selected car should have the names of at least three previous owners.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

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.