Using MongoDB with C#

Video tutorial: IAmTimCorey - How to Connect MongoDB to C# the Easy Way (Sandbox\Mongo\MongoDbDemoApp).

We will create an ASP.Net Core 6.0 console application named **MongoDbDemoApp**.

Note: in this solution we are using the file scope not block scope. This isn't the way I usually work.

The first thing we need to do is to add the MongoDB.Driver NuGet package to our solution.

Add the **using** statement for the driver.

```
using MongoDB.Driver;
```

Create a class named **PersonModel** to your **MongoDBDemo** project. Add the namespace MongoDBDemo.

```
namespace MongoDBDemo;
```

Add the class details. and we add two attributes.

When you type in `[BsonId]` you will need to add the using statement, `using MongoDB.Bson.Serialization.Attributes;`.

When you add in the next attribute `[BsonRepresentation(BsonType.ObjectId)]` you will need to add another using statement, `using MongoDB.Bson;`.

Your class should look like this.

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace MongoDBDemo;

    public class PersonModel
    {
        [BsonId]
        [BsonRepresentation(BsonType.ObjectId)]
        public string Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
```

Note: `Id` is our unique Id and we won't need to add this value because MongoDB will automatically generate and Id for us. The decorators above the Id tell MongoDB that this property is our unique Id.

In our program.cs file we need to add the using statement, `using MongoDBDemo;`.

Connection strings

Normally we would store the connection string in `appsettings.json` but for now we will store the connection string in program.cs.

The code below is our full example.

```
using MongoDB.Driver;
using MongoDBDemo;

string connectionString = "mongodb://127.0.0.1:27017";
string databaseName = "AddressBook";
string collectionName = "people";

var client = new MongoClient(connectionString);
var db = client.GetDatabase(databaseName);
var collection = db.GetCollection<PersonModel>(collectionName);

var person = new PersonModel { FirstName = "Alan", LastName = "Robson" };

await collection.InsertOneAsync(person);

var results = await collection.FindAsync(_ => true);
foreach (var result in results.ToList())
{
    Console.WriteLine($"{result.Id}: {result.FirstName} {result.LastName}");
}
```

Note: we don't use a password with the local MongoDB.

Our database name is **AddressBook** and our collection name (table) is **people**. Both the AddressBook and people will be created by our `InsertOneAsync()` method if they don't already exist.

we create a database client with our connection string and then create a database connection with `GetDatabase`. Then we tell MongoDB what collection we will be using with `GetCollection()` and we give the collection a type with `<PersonModel>`.

```
var client = new MongoClient(connectionString);
var db = client.GetDatabase(databaseName);
var collection = db.GetCollection<PersonModel>(collectionName);
```

We create a person object and use the following to add one document to our MongoDB database.

```
var person = new PersonModel { FirstName = "Alan", LastName = "Robson" };

await collection.InsertOneAsync(person);
```

As we said previously we have now created a database and a collection in the database. We have also added one document to the collection.

Now, we want to be able to view documents from the collection we just created.

```
var results = await collection.FindAsync(_ => true);

foreach (var result in results.ToList())
{
    Console.WriteLine($"{result.Id}: {result.FirstName} {result.LastName}");
}
```

Where `collection.FindAsync(_ => true);` means just send me all of the records in the collection.

We have run the code a couple of times and this is what is returned.

> 62b3d5a29133fbd0ff62d290: Alan Robson
> 62b40b727043c25560c11929: Charley Robson
> 62b40ba81d3f5eeab9e60266: James Robson

With a little bit of code we have gained a lot of functionality. We can expand on this by adding a class library into a new Project named **MongoDataAccess**.

Create two new folders in this project, one named **DataAccess** and the other **Models**.

In the Models folder create a new class named **UserModel**.

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace MongoDataAccess.Models;

public class UserModel
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string FullName => $"{FirstName} {LastName}";
}
```

This is similar code to the PersonModel class we created previously. Once again you have to add the MongoDB.Driver package to this project.

Create a new model named **ChoreModel**.

```
using MongoDB.Bson.Serialization.Attributes;
using MongoDB.Bson;

namespace MongoDataAccess.Models;

public class ChoreModel
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; }
    public string ChoreText { get; set; }
    public int FrequencyInDays { get; set; }
    public UserModel? AssignedTo { get; set; }
    public DateTime? LastCompleted { get; set; }
}
```

We are creating a chore project to keep a database of a particular users chores.

Now create a final model named **ChoreHistoryModel**.

```
using MongoDB.Bson.Serialization.Attributes;
using MongoDB.Bson;

namespace MongoDataAccess.Models;

public class ChoreHistoryModel
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; }
    public string ChoreId { get; set; }
    public string ChoreText { get; set; }
    public DateTime DateCompleted { get; set; }
    public UserModel WhoCompleted { get; set; }

    public ChoreHistoryModel()
    {

    }

    public ChoreHistoryModel(ChoreModel chore)
    {
        ChoreId = chore.Id;
        DateCompleted = chore.LastCompleted ?? DateTime.Now;
        WhoCompleted = chore.AssignedTo;
        ChoreText = chore.ChoreText;
    }
}
```

We have two constructors in this class. One is empty and the other takes on a ChoreModel and you can use this to save an object of ChoreModel.

In our DataAccess folder we are going to create a data access class named **ChoreDataAccess**.

Add this code to your class.

```
using MongoDataAccess.Models;
using MongoDB.Driver;

namespace MongoDataAccess.DataAccess;

public class ChoreDataAccess
{

}
```

Now we will add some constants to connect to our database.

```
private const string ConnectionString = "mongodb://127.0.0.1:27017";
private const string DatabaseName = "choredb";
private const string ChoreCollection = "chore_chart";
private const string UserCollection = "users";
private const string ChoreHistoryCollection = "chore_history";
```

In `program.cs` we are using this setup.

```
var client = new MongoClient(connectionString);
var db = client.GetDatabase(databaseName);
var collection = db.GetCollection<PersonModel>(collectionName);
```

We have to do this for every collection. Instead of doing this every single time for every single call we are going to create an **IMongoCollection<T>** of type **T** called `ConnectToMongo<T>(in string collection)`. This is using generics.

What we are saying is that with generics we are passing in a collection of type **T** and we want to pass back a collection of objects of that type. This allows us to pass in three different types with the same code. In our case, `ChoreHistoryModel`, `ChoreModel` and `UserModel`.

```
private IMongoCollection<T> ConnectToMongo<T>(in string collection)
{
    var client = new MongoClient(ConnectionString);
    var db = client.GetDatabase(DatabaseName);
    return db.GetCollection<T>(collection);
}
```

This allows us to get our data using one private method.

Accessing our collections

We can now access our data by creating some methods.

```
    public async Task<List<UserModel>> GetAllUsers()
    {
        var usersCollection = ConnectToMongo<UserModel>(UserCollection);
        var results = await usersCollection.FindAsync(_ => true);
        return results.ToList();
    }
```

Now, we have only one line to connect to our database rather than using the three lines we used in our original code.

FindAsync() once again is just sending us back **all** of the documents in the collection.

We return the results as a list.

We can also create another method to **GetAllChores()** with basically the same code as we used in GetAllUsers().

Filtering a collection

Now we want to create a method that filters documents so that we can get all chores for a particular user. Once again we are using the same code with minor modifications.

```
    public async Task<List<ChoreModel>> GetAllChoresForAUser(UserModel user)
    {
        var choresCollection = ConnectToMongo<ChoreModel>(ChoreCollection);
        var results = await choresCollection.FindAsync(c => c.AssignedTo.Id ==
user.Id);
        return results.ToList();
    }
```

This is the way we do a one to many filter of documents. `FindAsync(c => c.AssignedTo.Id == user.Id)` is saying get the sub Id's of the chore collection that are matched to the user Id.

Inserting a document

We are going to build a method to create a user. Once again we will use the same connection line from previous methods to save us some time.

```
    public Task CreateUser(UserModel user)
    {
        var usersCollection = ConnectToMongo<UserModel>(UserCollection);
```

```
        return usersCollection.InsertOneAsync(user);
    }
```

We can do the same to insert a chore.

Updating a document

We can update a chore with the following code.

```
    public Task UpdateChore(ChoreModel chore)
    {
        var choresCollection = ConnectToMongo<ChoreModel>(ChoreCollection);
        var filter = Builders<ChoreModel>.Filter.Eq("Id", chore.Id);
        return choresCollection.ReplaceOneAsync(filter, chore, new ReplaceOptions
{ IsUpsert = true });
    }
```

This searches for a document with a chore.Id that matches the Id in the chore parameter. `IsUpsert = true` states that if the chore with the Id is found then replace that chore with chore that has been sent to the method. If it isn't found then just insert the chore document into the database.

In this case it can either be an update or an insert.

This operation isn't like SQL Server that will just update the new fields. MongoDB is more efficient and doesn't bother to replace the fields but pulls a document out of the collection and replaces it with a new document.

Deleting a document

```
    public Task DeleteChore(ChoreModel chore)
    {
        var choresCollection = ConnectToMongo<ChoreModel>(ChoreCollection);
        return choresCollection.DeleteOneAsync(c => c.Id == chore.Id);
    }
```

What this is doing is saying delete one document where the Id's match.

We now have a full CRUD access to MongoDB.

ChoreDataAccess.cs

```
    using MongoDataAccess.Models;
    using MongoDB.Driver;

    namespace MongoDataAccess.DataAccess;

    public class ChoreDataAccess
```

```
    {
        private const string ConnectionString = "mongodb://127.0.0.1:27017";
        private const string DatabaseName = "choredb";
        private const string ChoreCollection = "chore_chart";
        private const string UserCollection = "users";
        private const string ChoreHistoryCollection = "chore_history";

        private IMongoCollection<T> ConnectToMongo<T>(in string collection)
        {
            var client = new MongoClient(ConnectionString);
            var db = client.GetDatabase(DatabaseName);
            return db.GetCollection<T>(collection);
        }

        public async Task<List<UserModel>> GetAllUsers()
        {
            var usersCollection = ConnectToMongo<UserModel>(UserCollection);
            var results = await usersCollection.FindAsync(_ => true);
            return results.ToList();
        }

        public async Task<List<ChoreModel>> GetAllChores()
        {
            var choresCollection = ConnectToMongo<ChoreModel>(ChoreCollection);
            var results = await choresCollection.FindAsync(_ => true);
            return results.ToList();
        }

        public async Task<List<ChoreModel>> GetAllChoresForAUser(UserModel user)
        {
            var choresCollection = ConnectToMongo<ChoreModel>(ChoreCollection);
            var results = await choresCollection.FindAsync(c => c.AssignedTo.Id ==
user.Id);
            return results.ToList();
        }

        public Task CreateUser(UserModel user)
        {
            var usersCollection = ConnectToMongo<UserModel>(UserCollection);
            return usersCollection.InsertOneAsync(user);
        }

        public Task CreateChore(ChoreModel chore)
        {
            var choresCollection = ConnectToMongo<ChoreModel>(ChoreCollection);
            return choresCollection.InsertOneAsync(chore);
        }

        public Task UpdateChore(ChoreModel chore)
        {
            var choresCollection = ConnectToMongo<ChoreModel>(ChoreCollection);
            var filter = Builders<ChoreModel>.Filter.Eq("Id", chore.Id);
            return choresCollection.ReplaceOneAsync(filter, chore, new
ReplaceOptions { IsUpsert = true });
```

```
        }

        public Task DeleteChore(ChoreModel chore)
        {
            var choresCollection = ConnectToMongo<ChoreModel>(ChoreCollection);
            return choresCollection.DeleteOneAsync(c => c.Id == chore.Id);
        }
    }
```

Using our data access layer

We will go back to our **MongoDBDemo** project and comment out our database connection code and the `person` code in the Program.cs file.

We also need to change our dependencies by adding our **MongoDataAccess** library. We do this by adding a project reference to the library.

We have to add the following **using** statements.

```
    using MongoDataAccess.DataAccess;
    using MongoDataAccess.Models;
```

Inserting a User

```
    var db = new ChoreDataAccess();

    await db.CreateUser(new UserModel() { FirstName = "Alan", LastName = "Robson"
});
```

Here we open our database connection and then use it to create a new user.

Note: all of our database methods are **async** methods so we need to use the `await` keyword in front of our database interactions.

Inserting a Chore

With our database connection still open we can add a chore to the current user. First, we must get our users collection and in our case we are going to select the first user to attach a chore to.

```
    var users = await db.GetAllUsers();

    var chore = new ChoreModel()
    {
        AssignedTo = users.First(),
        ChoreText = "Mow the Lawn",
        FrequencyInDays = 7
```

```
    };

    await db.CreateChore(chore);
```

Completing Chore history

We also need to update the chore if that chore has been completed. We can do this with.

```
    public async Task CompleteChore(ChoreModel chore)
    {
        var choresCollection = ConnectToMongo<ChoreModel>(ChoreCollection);
        var filter = Builders<ChoreModel>.Filter.Eq("Id", chore.Id);
        await choresCollection.ReplaceOneAsync(filter, chore);

        var choreHistoryCollection = ConnectToMongo<ChoreHistoryModel>
(ChoreHistoryCollection);
        await choreHistoryCollection.InsertOneAsync(new ChoreHistoryModel(chore));
    }
```

This code is adequate but seeing as we are using two collections we should do some exception checking. A better way would be to add the following code using transactions so that we can roll back if something goes wrong.

```
    public async Task CompleteChore(ChoreModel chore)
    {
        var client = new MongoClient(ConnectionString);
        using var session = await client.StartSessionAsync();

        session.StartTransaction();

        try
        {
            var db = client.GetDatabase(DatabaseName);
            var choresCollection = db.GetCollection<ChoreModel>(ChoreCollection);
            var filter = Builders<ChoreModel>.Filter.Eq("Id", chore.Id);
            await choresCollection.ReplaceOneAsync(filter, chore);

            var choreHistoryCollection = db.GetCollection<ChoreHistoryModel>
(ChoreHistoryCollection);
            await choreHistoryCollection.InsertOneAsync(new
ChoreHistoryModel(chore));

            await session.CommitTransactionAsync();
        }
        catch (Exception ex)
        {
            await session.AbortTransactionAsync();
            Console.WriteLine(ex.Message);
```

```
        }
    }
```

Having said that we can't actually run a transaction on a local machine because there is only one cluster.

To get around that we can use MongoDB Atlas in the cloud where we are using three clusters and now the transaction will work.

Note: in the transaction version we aren't using the `ConnectToMongo<>()` method. This is because we are using a `using` statement to create a session.

We can test the CompleteChore method with this code.

```
    //// Update a chore
    var chores = await db.GetAllChores();

    var newChore = chores.First();
    newChore.LastCompleted = DateTime.UtcNow;

    await db.CompleteChore(newChore);
```

Result:

> _id: 62b54465235cdea1dd14db49
> ChoreId: "62b53574de42f5dbe02d1abf"
> ChoreText: "Mow the Lawn"
> DateCompleted: 2022-06-24T04:58:13.887+00:00
>
> WhoCompleted
> Object
> _id: 62b533031e49b6de65344b2a
> FirstName: "Alan"
> LastName: "Robson"

Getting a user and their chores

```
    // View a user and their chores
    var users = await db.GetAllUsers();

    foreach (var user in users.ToList())
    {
        Console.WriteLine($"{user.Id}: {user.FirstName} {user.LastName}");

        var chores = await db.GetAllChoresForAUser(user);

        foreach (var chore in chores)
        {
            Console.Write($"{chore.Id}: {chore.ChoreText}");
```

```
            if (chore.FrequencyInDays > 0)
            {
                Console.Write($"  - Frequency {chore.FrequencyInDays} days.");
            }

            if (chore.LastCompleted != null)
            {
                Console.Write($" Last completed: {chore.LastCompleted}\n");
            }
        }
    }
```

Conclusion

This video course has just scratched the surface of using MongoDB with C#. We are using `users.First()` to grab the first user in our collection and then attaching a chore to that user. We need to harden these routines so that we can select a particular user.

We also need to use LINQ to query more than one collection at a time and return one to many results.

This tutorial is a continuation of the *Intro to MongoDB with C#* (Sandbox\Mongo\MongoDbDemo) tutorial I did. Both tutorials created a data access layer but in the examples above we are using async functions to retrieve our data which is more efficient.