Create & Read Operations

This lesson teaches how to create the class used to access MongoDB and the methods to perform the basic CRUD operations is C#.

WE'LL COVER THE FOLLOWING

Users Repository

The goal of this lesson is to create a class which will give us the ability to do simple CRUD operations on the `users` collection.

The first thing we need to do is connect to the database from our application. The easiest way to do this is by using the MongoClient class, available via the MongoDB driver that has already been installed for you. (its constructor requires a connection string, which we will provide).

Alternatively, *MongoClientSettings* class, available via the MongoDB driver, can be used as it provides various possibilities. One such possibility is the *ClusterConfiguration* property, which is of the *ClusterBuilder* type and used for configuring clusters.

Other classes that we need to use are MongoDatabase, to access defined databases (`blog` in this case), and MongoCollection, to access defined collections (`users` in this case).

Here is how it looks in code:

```
/// <summary>
/// Class used to access Mongo DB.
/// </summary>
public class UsersRepository
{
    private IMongoClient _client;
    private IMongoDatabase _database;
    private IMongoCollection<User> _usersCollection;

    public UsersRepository(string connectionString)
    {
        _client = new MongoClient(connectionString);
        _database = _client.GetDatabase("blog");
        _usersCollection = _database.GetCollection<User>("users");
    }
}
```

Ok, this is pretty straightforward; but still, there is one thing that should be noted. In line **8**, *MongoCollection* is defined using *Users* as a template type. This is only possible because we added those attributes to the *Users* class. The other way to achieve this is by using the *BsonDocument*, but then we would manually have to map the fields to the properties of the *Users* class.

Create Users

Inserting a document in the database is easily done easily once the previous steps have been followed:

```
public async Task InsertUser(User user)
{
    await _usersCollection.InsertOneAsync(user);
}
```

Obviously, I have used the asynchronous operation `InsertOneAsync`, but you can use a synchronous one too.

Again, because we mapped JSON fields on `User` properties, it is easy to work with these async operations.

Implementation

Now, let's look at the executable code for the *Insert User* function.

> **Note**: The `InsertUser()` function shows no output, as the entry is simply inserted into the database. However, the code below has been modified

> to display *User Inserted* when an insertion is done.

The output of this executable will be displayed in the terminal tab.

```
using MongoDB.Bson;
using System.Linq;
using System.Threading.Tasks;
using Xunit;

namespace mongonetcore
{
    /// <summary>
    /// Testing MongoDbRepository class.
    /// </summary>
    /// <notes>
    /// In order for these tests to pass, you must have mongo server runing on localhost:2701
    /// If you need more info on how to do so check this blog post:
    /// https://rubikscode.net/2017/07/24/mongo-db-basics-part-1/
    /// </notes>
    public class MongoDbRepositoryTests
    {

    }
}
```

Read Users

Reading users is done like this:

```
public async Task<List<User>> GetAllUsers()
{
    return await _usersCollection.Find(new BsonDocument()).ToListAsync();
}

public async Task<List<User>> GetUsersByField(string fieldName, string fieldValue)
{
    var filter = Builders<User>.Filter.Eq(fieldName, fieldValue);
    var result = await _usersCollection.Find(filter).ToListAsync();

    return result;
}

public async Task<List<User>> GetUsers(int startingFrom, int count)
{
    var result = await _usersCollection.Find(new BsonDocument())
                                       .Skip(startingFrom)
                                       .Limit(count)
                                       .ToListAsync();

    return result;
}
```

There are three different implementations of this functionality. Let's go

through each of them.

- In **line 1**, function `GetAllUsers` returns all users from the database.
    - We use the `Find` method (**line 3**) of the `MongoCollection` class to do so and pass the empty *BsonDocument* into it.
- In the next method, `GetUsersByField` method (**line 6**), we can see that the `Find` method (**line 9**) actually receives the filter object (**line 8**), which it will use as a criterion for getting the data.
- In the first function (**line 1**), we use an empty filter and thus receive all users from the collection.
- In the second function (**line 8**), we use `Builder` to create the filter which will be used against the database.
- Finally, the last function – `GetUsers` (**line 14**) uses `Skip` and `Limit` methods (**lines 17 and 18**) of *MongoCollection* to get a necessary chunk of data. This last function can be used for *paging*.

Implementation

Now, let's look at the executable code for the three *Read Users* functions discussed above.

The output of this executable will be displayed in the terminal tab.

```
using MongoDB.Bson;
using System.Linq;
using System.Threading.Tasks;
using Xunit;

namespace mongonetcore
{
    /// <summary>
    /// Testing MongoDbRepository class.
    /// </summary>
    /// <notes>
    /// In order for these tests to pass, you must have mongo server runing on localhost:2701
    /// If you need more info on how to do so check this blog post:
    /// https://rubikscode.net/2017/07/24/mongo-db-basics-part-1/
    /// </notes>
    public class MongoDbRepositoryTests
    {

    }
}
```

We discussed some of the CRUD operations in this lesson; let's take a look at some more in the next lesson.