Update, Delete & Index Operations

This lesson will teache you how to perform the update and delete user operations in C#. It will also teache you how to create indexes.

WE'LL COVER THE FOLLOWING

In the previous lesson, we looked at how the basic CRUD operations, create and read, could be performed using C#. Let's learn about some more of them below.

Update Users

Documents can be updated in a similar manner:

```
public async Task<bool> UpdateUser(ObjectId id, string udateFieldName, string updateFieldValu
{
    var filter = Builders<User>.Filter.Eq("_id", id);
    var update = Builders<User>.Update.Set(udateFieldName, updateFieldValue);

    var result = await _usersCollection.UpdateOneAsync(filter, update);

    return result.ModifiedCount != 0;
}
```

In this example, function `UpdateOneAsync` is used, but if you want to change values of multiple documents, you can use `UpdateMultipleAsync`.

Synchronous siblings of these operations also exist.

Implementation

Now, let's look at the executable code for the *Update Users* function 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
    {

    }
}
```

Another interesting fact is that, using this function, you can add fields that are not in the "schema". For example, if you want to add a new field in the `user` document, you can do this:

```
var users = await _mongoDbRepo.GetUsersByField("name", "Nikola");
var user = users.FirstOrDefault();
var result = await _mongoDbRepo.UpdateUser(user.Id, "address", "test address");
```

This way the big MongoDB's (and the rest of the document databases) feature of a modular schema is harnessed; you are not blocked by defined objects, and you can store information dynamically if you need to do so.

Delete Users

Users can be deleted this way:

```
public async Task<bool> DeleteUserById(ObjectId id)
{
    var filter = Builders<User>.Filter.Eq("_id", id);
```

```
    var result = await _usersCollection.DeleteOneAsync(filter);
    return result.DeletedCount != 0;
}

public async Task<long> DeleteAllUsers()
{
    var filter = new BsonDocument();
    var result = await _usersCollection.DeleteManyAsync(filter);
    return result.DeletedCount;
}
```

Pretty straightforward, don't you think? Same as in the Reading examples, `Builder` was used for creating a filter.

Implementation

Now, let's look at the executable code for the *Delete 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
    {

    }
}
```

Creating Indexes

Adding indexes is a bit different. If we want to do it dynamically, the easiest way is to use *BsonDocument*. Here is how it is done:

```
public async Task CreateIndexOnCollection(IMongoCollection<BsonDocument> collection, string f
{
    var keys = Builders<BsonDocument>.IndexKeys.Ascending(field);
    await collection.Indexes.CreateOneAsync(keys);
```

```
}
```

On the other hand, if we know what our indexes will be beforehand, we can use strongly typed implementation like this:

```
public async Task CreateIndexOnNameField()
{
    var keys = Builders<User>.IndexKeys.Ascending(x => x.Name);
    await _usersCollection.Indexes.CreateOneAsync(keys);
}
```

Conclusion

Here, in one example, we went through many of the MongoDB Driver features; and we learned how to map JSON documents to .NET objects, which gave us an easy way to use the rest of the features. The uses of CRUD operations, which are the operations that are most commonly used, have been demonstrated and explained.

In short, we have shown how to use MongoDB features with C# in a .NET environment

---

In the next lesson, we will compile the whole project in order to test the functions that have discussed so far.