

**Generic Mongo Repository pattern implemented in .NET Core**

https://medium.com/@marekzyla95/mongo-repository-pattern-700986454a0e

**Jump into code**

Hello, today I want to present you modern Mongo Generic Repository implemented in loved by back-end developers .NET Core, let’s jump right into the shark’s mouth! 🦈

**Quick Start**

**Packages installation**

Okay, to get started we have to prepare our environment, let’s install necessary .nuget packages, you can do it in several ways:

* using .NET CLI  
  dotnet add package MongoDB.Driver
* using Package Manager  
  Install-Package MongoDB.Driver

**Mongo Db server configuration**

In this article I hope you already have MongoDb server instance configured. I’m using **Docker** to run my MongoDb server. The most important thing is to get proper connection string while connecting to our database. My server is running on localhost:32772.

**Configure Startup**

Once, we have our packages installed, what we need to do, is to connect our application to Mongo database. Let’s do it!

Modify appsettings.json to provide proper connection string and create database name. I created additional JSON object containing important settings.

"MongoDbSettings": {

"ConnectionString": "mongodb://localhost:32772",

"DatabaseName": "ExampleDatabase"

},

Now, let’s create **settings**class to keep clean code inStartup.cs when it comes to types registration.

public interface IMongoDbSettings

{

string DatabaseName { get; set; }

string ConnectionString { get; set; }

}

public class MongoDbSettings : IMongoDbSettings

{

public string DatabaseName { get; set; }

public string ConnectionString { get; set; }

}

Next, we have to register our MongoDb client instance in Startup.cs using previously prepared settings class. Do it **before**AddControllers() in ConfigureServices method.

public void ConfigureServices(IServiceCollection services)

{

services.Configure<MongoDbSettings>(Configuration.GetSection("MongoDbSettings"));

services.AddSingleton<IMongoDbSettings>(serviceProvider =>

serviceProvider.GetRequiredService<IOptions<MongoDbSettings>>().Value);

services.AddControllers();

}

**Repository implementation**

**Document preparation**

We’ll start our implementation by creating proper abstraction layer for our documents (*entities*in Mongo). I created a class and interface that contains base information about each document

public interface IDocument

{

[BsonId]

[BsonRepresentation(BsonType.String)]

ObjectId Id { get; set; }

DateTime CreatedAt { get; }

}

public abstract class Document : IDocument

{

public ObjectId Id { get; set; }

public DateTime CreatedAt => Id.CreationTime;

}

As you can see, Id property has Bson attributes for configuration. It tells our Mongo Driver to treat this property as native mongoobjectId.

**Collection naming**

Mongo Driver lacks attribute that allows us to set the collection name for documents, so I created one.

[AttributeUsage(AttributeTargets.Class, Inherited = false)]

public class BsonCollectionAttribute : Attribute

{

public string CollectionName { get; }

public BsonCollectionAttribute(string collectionName)

{

CollectionName = collectionName;

}

}

And here is this attribute in action for our sample document

[BsonCollection("people")]

public class Person : Document

{

public string FirstName { get; set; }

public string LastName { get; set; }

public DateTime BirthDate { get; set; }

}

As you can see, we have a class called Person, but actual collection in database will have name of people.

**Repository**

Oh and that’s what you’ve been waiting for! Finally, repository pattern implemented in dotnet core. We have here interface and its implementation, I definitely suggest you putting it in separate files, because we have here a lot of methods.

public interface IMongoRepository<TDocument> where TDocument : IDocument

{

IQueryable<TDocument> AsQueryable();

IEnumerable<TDocument> FilterBy(

Expression<Func<TDocument, bool>> filterExpression);

IEnumerable<TProjected> FilterBy<TProjected>(

Expression<Func<TDocument, bool>> filterExpression,

Expression<Func<TDocument, TProjected>> projectionExpression);

TDocument FindOne(Expression<Func<TDocument, bool>> filterExpression);

Task<TDocument> FindOneAsync(Expression<Func<TDocument, bool>> filterExpression);

TDocument FindById(string id);

Task<TDocument> FindByIdAsync(string id);

void InsertOne(TDocument document);

Task InsertOneAsync(TDocument document);

void InsertMany(ICollection<TDocument> documents);

Task InsertManyAsync(ICollection<TDocument> documents);

void ReplaceOne(TDocument document);

Task ReplaceOneAsync(TDocument document);

void DeleteOne(Expression<Func<TDocument, bool>> filterExpression);

Task DeleteOneAsync(Expression<Func<TDocument, bool>> filterExpression);

void DeleteById(string id);

Task DeleteByIdAsync(string id);

void DeleteMany(Expression<Func<TDocument, bool>> filterExpression);

Task DeleteManyAsync(Expression<Func<TDocument, bool>> filterExpression);

}

This interface defines all public methods that our repository will serve. As you can see there are special methods for each CRUD operation. Below I pasted an implementation and longer description of each method.

public class MongoRepository<TDocument> : IMongoRepository<TDocument>

where TDocument : IDocument

{

private readonly IMongoCollection<TDocument> \_collection;

public MongoRepository(IMongoDbSettings settings)

{

var database = new MongoClient(settings.ConnectionString).GetDatabase(settings.DatabaseName);

\_collection = database.GetCollection<TDocument>(GetCollectionName(typeof(TDocument)));

}

private protected string GetCollectionName(Type documentType)

{

return ((BsonCollectionAttribute) documentType.GetCustomAttributes(

typeof(BsonCollectionAttribute),

true)

.FirstOrDefault())?.CollectionName;

}

public virtual IQueryable<TDocument> AsQueryable()

{

return \_collection.AsQueryable();

}

public virtual IEnumerable<TDocument> FilterBy(

Expression<Func<TDocument, bool>> filterExpression)

{

return \_collection.Find(filterExpression).ToEnumerable();

}

public virtual IEnumerable<TProjected> FilterBy<TProjected>(

Expression<Func<TDocument, bool>> filterExpression,

Expression<Func<TDocument, TProjected>> projectionExpression)

{

return \_collection.Find(filterExpression).Project(projectionExpression).ToEnumerable();

}

public virtual TDocument FindOne(Expression<Func<TDocument, bool>> filterExpression)

{

return \_collection.Find(filterExpression).FirstOrDefault();

}

public virtual Task<TDocument> FindOneAsync(Expression<Func<TDocument, bool>> filterExpression)

{

return Task.Run(() => \_collection.Find(filterExpression).FirstOrDefaultAsync());

}

public virtual TDocument FindById(string id)

{

var objectId = new ObjectId(id);

var filter = Builders<TDocument>.Filter.Eq(doc => doc.Id, objectId);

return \_collection.Find(filter).SingleOrDefault();

}

public virtual Task<TDocument> FindByIdAsync(string id)

{

return Task.Run(() =>

{

var objectId = new ObjectId(id);

var filter = Builders<TDocument>.Filter.Eq(doc => doc.Id, objectId);

return \_collection.Find(filter).SingleOrDefaultAsync();

});

}

public virtual void InsertOne(TDocument document)

{

\_collection.InsertOne(document);

}

public virtual Task InsertOneAsync(TDocument document)

{

return Task.Run(() => \_collection.InsertOneAsync(document));

}

public void InsertMany(ICollection<TDocument> documents)

{

\_collection.InsertMany(documents);

}

public virtual async Task InsertManyAsync(ICollection<TDocument> documents)

{

await \_collection.InsertManyAsync(documents);

}

public void ReplaceOne(TDocument document)

{

var filter = Builders<TDocument>.Filter.Eq(doc => doc.Id, document.Id);

\_collection.FindOneAndReplace(filter, document);

}

public virtual async Task ReplaceOneAsync(TDocument document)

{

var filter = Builders<TDocument>.Filter.Eq(doc => doc.Id, document.Id);

await \_collection.FindOneAndReplaceAsync(filter, document);

}

public void DeleteOne(Expression<Func<TDocument, bool>> filterExpression)

{

\_collection.FindOneAndDelete(filterExpression);

}

public Task DeleteOneAsync(Expression<Func<TDocument, bool>> filterExpression)

{

return Task.Run(() => \_collection.FindOneAndDeleteAsync(filterExpression));

}

public void DeleteById(string id)

{

var objectId = new ObjectId(id);

var filter = Builders<TDocument>.Filter.Eq(doc => doc.Id, objectId);

\_collection.FindOneAndDelete(filter);

}

public Task DeleteByIdAsync(string id)

{

return Task.Run(() =>

{

var objectId = new ObjectId(id);

var filter = Builders<TDocument>.Filter.Eq(doc => doc.Id, objectId);

\_collection.FindOneAndDeleteAsync(filter);

});

}

public void DeleteMany(Expression<Func<TDocument, bool>> filterExpression)

{

\_collection.DeleteMany(filterExpression);

}

public Task DeleteManyAsync(Expression<Func<TDocument, bool>> filterExpression)

{

return Task.Run(() => \_collection.DeleteManyAsync(filterExpression));

}

}

**Repository implementation explained**

Constructor**:**it connects to our database server by MongoClient and gets the proper database by GetDatabase method. Next, we get collection data and store it in private field called \_collection.

GetCollectionName: this method uses our previously prepared attribute and gets documents collection by type of document provided in parameter (*ex. when we pass Person as the parameter, this method will return “people” as a result*).

AsQueryable: this method let’s us to get whole non-materialized collection and perform some operations later (it’s slower than other methods because adds extra abstraction layer).

FilterBy: those methods allows us to filter data by sending expressions in parameters. This works similar to *LINQ*, but operations are performed on database, so it’s much faster than doing it later in code-side. I provided extra parameter called projectionExpression, this parameter is useful, when you want to get only some fields, rather than full objects from filtered results. You can also add sorting when necessary, simply using Sort method provided by IFindFluent interface.

FindOne and FindById: those methods are here to get single objects from database, I provided here also asynchronous implementations.

InsertOne and InsertMany: methods are used to insert documents to our database, also implemented in synchronous and asynchronous ways.

ReplaceOne: this method is used to replace documents in our application. You can also use Update method if you need.

DeleteOne and DeleteById: those are for hard-deleting documents in our app. When it comes to business application, i’d also suggest you using soft-delete, by simply updating some field called documentState and setting it to deleted. It’ll secure your data from being deleted by accident or some other cases.

**Usage of Repository**

**Registering repository in Startup**

As we finished implementation, the last thing to do, is to register our repository in Startup.cs in ConfigureServices method.

...

services.AddScoped(typeof(IMongoRepository<>), typeof(MongoRepository<>));

services.AddControllers();

...

**Usage in API**

Eventually, we can use our mechanism to perform some operations in actual application. Here, I have sample controller, you should never use your repository there, but it’s just to make this article shorter. When it comes to real application, consider adding some application services to communicate with your **data layer** which repository belongs to.

[ApiController]

[Route("[controller]")]

public class SampleController : ControllerBase

{

private readonly IMongoRepository<Person> \_peopleRepository;

public SampleController(IMongoRepository<Person> peopleRepository)

{

\_peopleRepository = peopleRepository;

}

[HttpPost("registerPerson")]

public async Task AddPerson(string firstName, string lastName)

{

var person = new Person()

{

FirstName = "John",

LastName = "Doe"

};

await \_peopleRepository.InsertOneAsync(person);

}

[HttpGet("getPeopleData")]

public IEnumerable<string> GetPeopleData()

{

var people = \_peopleRepository.FilterBy(

filter => filter.FirstName != "test",

projection => projection.FirstName

);

return people;

}

}

**Conclusion**

That’s all for today, we implemented together Mongo Generic Repository, using MongoDb Driver. You can also make it even faster by creating correct indexes for each collection, but this is a topic for separate article.

If you appreciate my work, don’t forget to react for this story by positive feedback. I have also other interesting articles on my profile, so make sure to check them out!

**Thanks.**