A picture containing text, book, statue, art

Description automatically generated

CONTENTS

**1 Problems with** **repository pattern**

[1. Fat](file:///E:\My\Drive\CQRS\My-CQRS.docx#_bookmark0) repository

[2. Performance](file:///E:\My\Drive\CQRS\My-CQRS.docx#_bookmark1) issues

[3. System](file:///E:\My\Drive\CQRS\My-CQRS.docx#_bookmark2) with different users and different roles   
on the same bounded context

**2 Advantages of CQRS pattern**

1. Single of responsibility

2. Separation of concern

[3. Independent](file:///E:\My\Drive\CQRS\My-CQRS.docx#_bookmark7) Scalability

4. Efficient Performance for database read operations.

5. Efficient with system that has different users and different roles on the same bounded context.

CONTENTS

**3 Types of CQRS pattern**

1. Single database

1.1 Single Database Single Project

1.2 Single Database Two Projects

2. Multiple databases

2.1 Same databases

2.2 different databases

**4 Implementation**

1. Single Database Single Project

2. Single Database Two Projects

3. Multiple Same Databases

3. Multiple Different databases

**Chapter 1**

**Problems with repository pattern**

**A picture containing drawing, illustration, art, cartoon

Description automatically generated**

**Repository pattern is one of the most popular pattern that used today to work with the database , there are also some other patterns to work with the database like CQRS pattern.**

**every pattern has its own advantages and disadvantages or in more accurate words every pattern is designed to solve specific problems or used with specific business or technical cases.**

**the architecture and development teams must be aware about this cases , when - why - how to use this pattern ? and also the impact for using this pattern for long term.**

**Overview of the project architecture**

The project follows the clean architecture and domain driven design principles , we will not focus on the clean architecture or domain driven design principles , but focus on the repository pattern and CQRS pattern from the view of clean architecture and domain driven design.

A screenshot of a computer

Description automatically generated

The aggregations are live on the Domain.Core

A screenshot of a computer

Description automatically generated

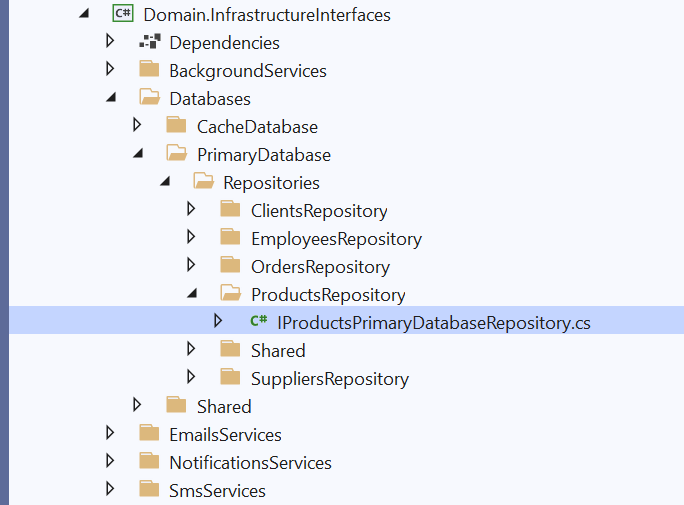
Let's take the product aggregation as example :

A screenshot of a computer

Description automatically generated with medium confidence

Product object is the aggregation root which control and manage the whole aggregation and other objects may be entities , value objects , Enums.

The product repository interface is live on :



And here the operations of products repository interface :

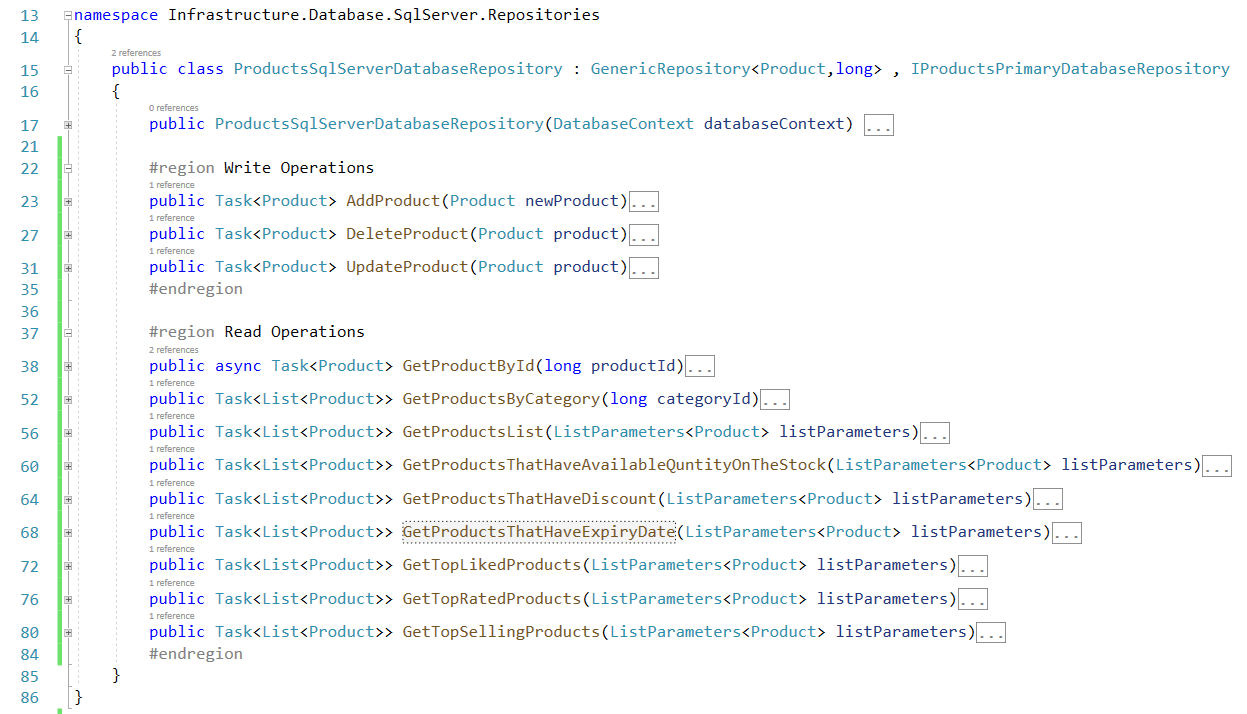
A screenshot of a computer code

Description automatically generated with low confidence

The actual implementation of the repository pattern is live on

A screenshot of a computer

Description automatically generated



**Repository pattern principles**

From Domain driven design, Repository pattern must apply two main principles:



**First principle**

**Repository pattern should be accessed via the aggregation root only.**

1. This mean repository must always **return or save** one aggregation root or list of aggregation roots ( **Product or list of Products** )
2. Repository pattern can not return normal entities directly in the aggregation like ( ProductAttribute ) , value objects like ( ProductTax ) , enums like ( ProductFileTypeEnum ).
3. Repository pattern can not return dto directly like

( ProductNameAndPriceDto )

A screenshot of a computer

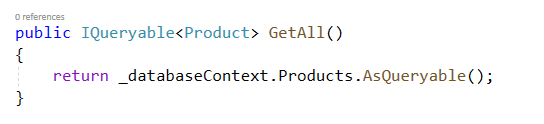
Description automatically generated with low confidence

1. Repository pattern should not return IQueryable , This is one of the most famous mistakes when using Repository pattern , because this break the main goal of repository pattern : abstract and isolate the logic or the implementation of data access layer outside the domain layer , here are example :

To get the products list which of golden category the products must satisfy the acceptance criteria :

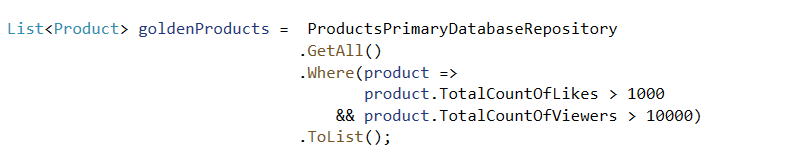
* Total number of likes > 1000
* Total number of viewers > 10,000

ProductsRepository.cs



You need to get golden products in two different services

Application Service 1



Application Service 2

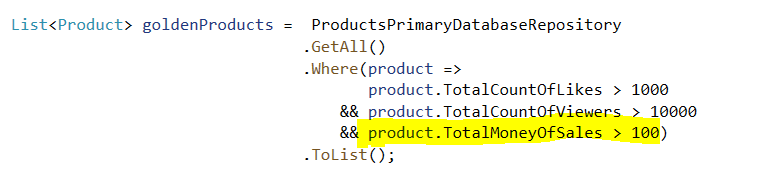
A screen shot of a computer code

Description automatically generated with low confidence

When new acceptance criteria is added or any change is occur , you must loop on all application services that get the golden products and apply this change , for example if new acceptance criteria is added to the golden product :

Total money of sales > 100

Application Service 1

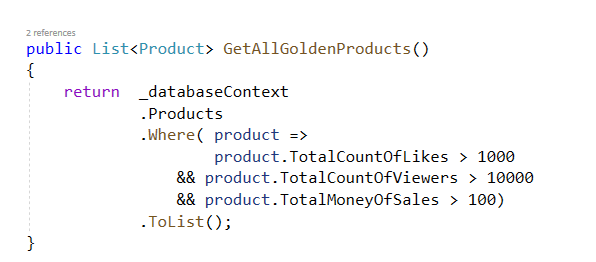


Application Service 2A picture containing text, screenshot, font, line

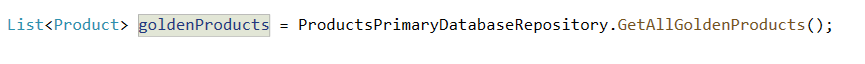
Description automatically generated

When repository pattern return IQueryable this cause a lot of duplications inside the application services , so repository pattern should not never return IQueryable , but return single or list of aggregation roots directly and put the logic or the implementation of database access inside the repository itself , so any change on the requirements will occur in single place in the repository and then will be reflected on the all application services directly.

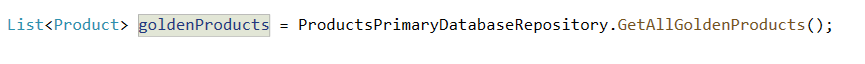
ProductsRepository.cs



Application Service 1



Application Service 2



1. Repository pattern can return native or primitive data types like numbers , Boolean

A screenshot of a computer code

Description automatically generated with low confidence

Important Note :

It usually prefers to use async/await with all operations of repository pattern or any network call , the previous examples are synchronous for just simplify.



**Second Principle :   
Repository pattern should load or save the whole aggregation together.**

From domain driven design view :

the aggregation is a group of objects that :

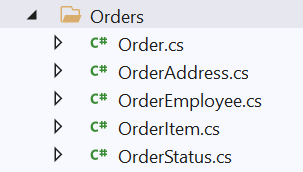
1. Logically and physically related, can product attribute exist without the product itself exist ? can product barcode , product file or any other object in the aggregation exist without the product ?

the answer is of course no.

A screenshot of a computer

Description automatically generated with medium confidence

1. affect on each other , for example on order aggregation :



when user add new order item , the total price property in the order is recalculated to add the new order item price to the previous total price.

Order.cs

A screen shot of a computer code

Description automatically generated with low confidence

From the code above :

* Order aggregation root is needed to load all order items to check some business rule like prevent duplications of order items.
* Any change on the order items list ( add – update – delete ) has an effect on total price on order.

So the whole aggregation objects is work as a single transaction.

This mean the repository must load the whole aggregation.

ProductsRepository.cs

A picture containing text, screenshot, font, number

Description automatically generated

Note : Product Tax is not loaded because it is value object and mapped to loaded automatically with the product.

Product Expression is not business entity but contain static methods only that return some conditions and expressions.

repository pattern can not load part only from the aggregation like product with product attributes only or product with product barcodes only and so on.

ProductsRepository.cs

A screenshot of a computer program

Description automatically generated with low confidence

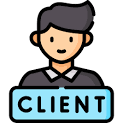
**Repository pattern and System with different roles**

In most of the enterprise systems , the system has users with different roles like : admin , client , manager , employee and so on.

Every role has permission to access some data and has not for others data.

Here are some examples from our project :

**User Story 1 :**



as a Client ,

I want to see the product name , price , attributes , images when I click on the product card on the home page,

So this help me to buy the product which I search about.

This are the data required from the product aggegation for the client to achive this story :

ProductsRepository.cs

A screenshot of a computer code

Description automatically generated with low confidence

**User Story 2 :**

A person driving a forklift

Description automatically generated

As a Warehouse Employee

I want to know the product stocks and every stock locations

So I can know where to put and store the products

This are the data required from the product aggegation for the warehouse employee to achive this story :

ProductsRepository.cs

A screenshot of a computer code

Description automatically generated with low confidence

**User Story 3 :**



As Sales Employee

I want to know the clients who like , view the products

So I can know extract reports and analysis information’s to increase sales of the company.

This are the data required from the product aggegation for the sales employee to achive this story :

ProductsRepository.cs

A screenshot of a computer program

Description automatically generated with low confidence

Repository pattern can work with different roles by two options :



First option:

Single operation to load the whole aggregation from the database , then map the only data required for every role or user story on the application service.

Example :

**User Story 4 :**



As admin,

I want to know the product report information like total numbers of likes, total numbers of views and …

So, I can know the products which interested by the clients.

A picture containing cartoon, screenshot, design

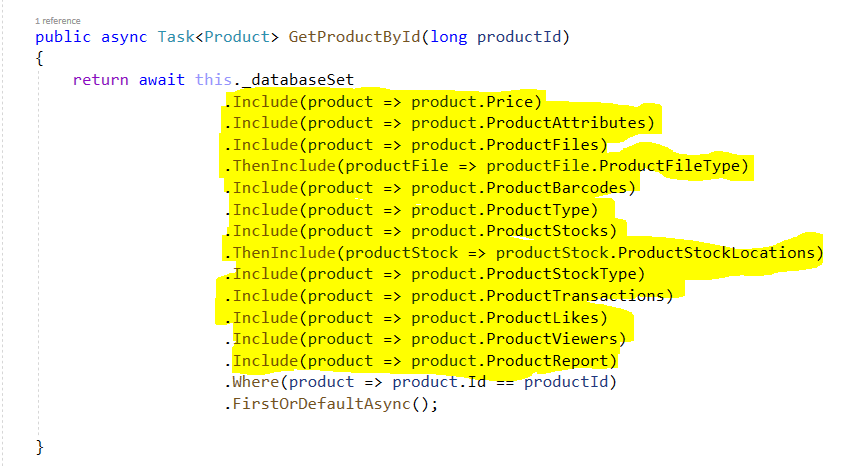
Description automatically generated

The code and implementation of user story 4

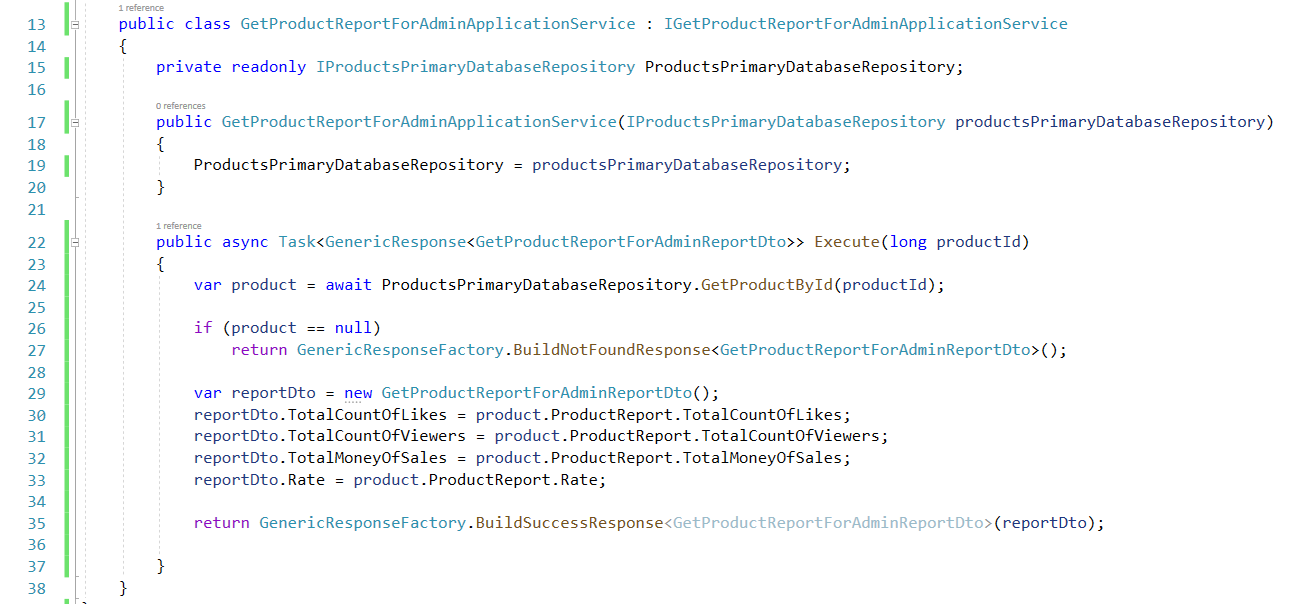
ProductsRepository.cs



GetProductById method is only method that used with all roles and load the whole aggregation from the database :



Then in the application service map only the data of report :



A red light with rays of light

Description automatically generated with low confidence

The problems with the first option :

1. The GetProductById method cause performance issues on both the database server and the application server , performance issue on the database because the query is very complex and contain a lot of join , performance issue on the application server because the memory of server will store a lot of unused data or objects for every user story or application service.
2. make the throughput of the application is very low.
3. make the latency of the application is very high because The query is very complex and slow.



Second option:

Every role has its own operation or method in the repository to load the only data which he has permission.

A picture containing cartoon, screenshot, design

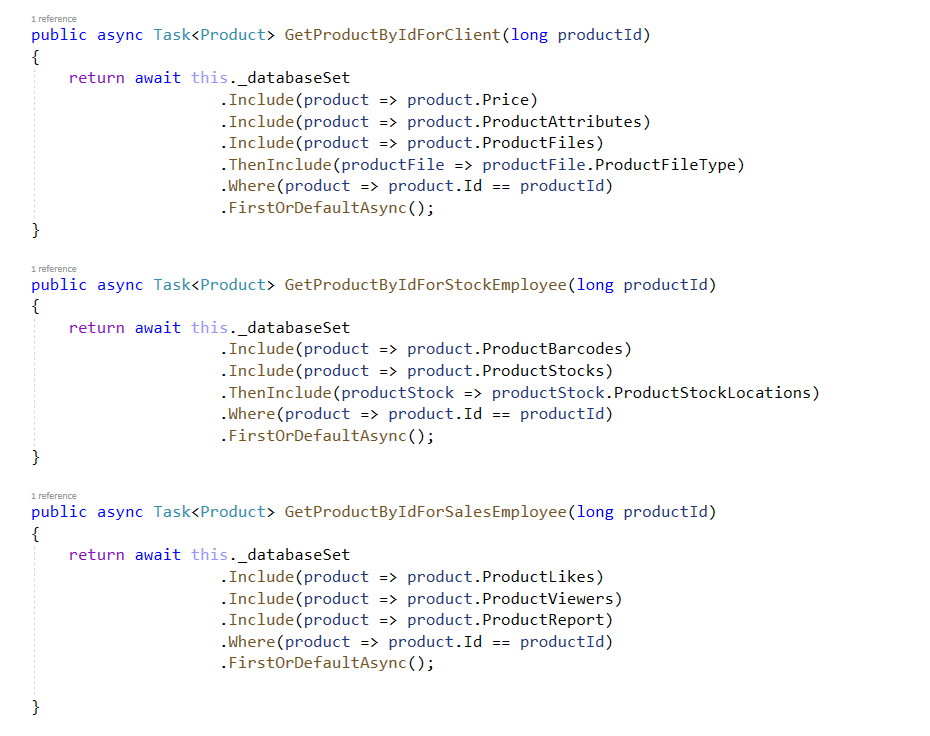
Description automatically generated

The code and implementation of second option :

ProductsRepository.cs

A picture containing text, screenshot, font

Description automatically generatedThe details of every method :



A red light with rays of light

Description automatically generated with low confidence

The problems with the second option :

1. this way break the second principles of repository pattern :

I search a lot about how repository pattern can works with the system that has different roles that require different data for every role , and after a lot of search , asking architects , read answers about this topic on the community website like stack over flow :

* The first opinion :

agree with the second option and break the second principle of the repository pattern and gain the performance.

* The second opinion :

Agree with the second option if only the operations is read only and dose not make any modification or change the data or used inside another transaction but agree only in case the data that returned from the repository will be returned directly to the end users.

To summarize the second opinion :

load the whole aggregation when will the aggregation will be modified or used with the write operation.

Load part of the aggregation with read only operations.

**User Story 5 :**

A person driving a forklift

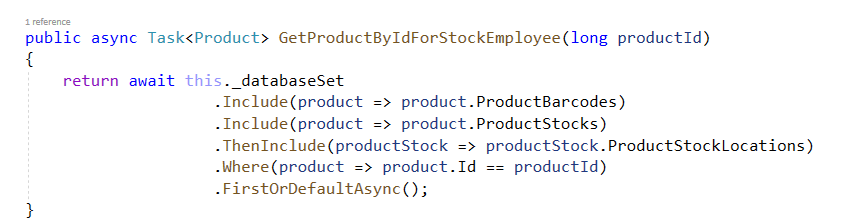
Description automatically generated

As stock employee,

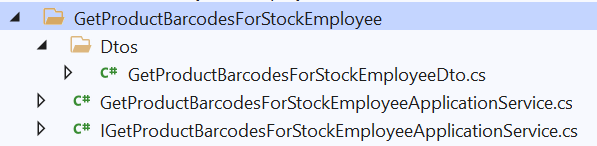
I want to know the product barcodes

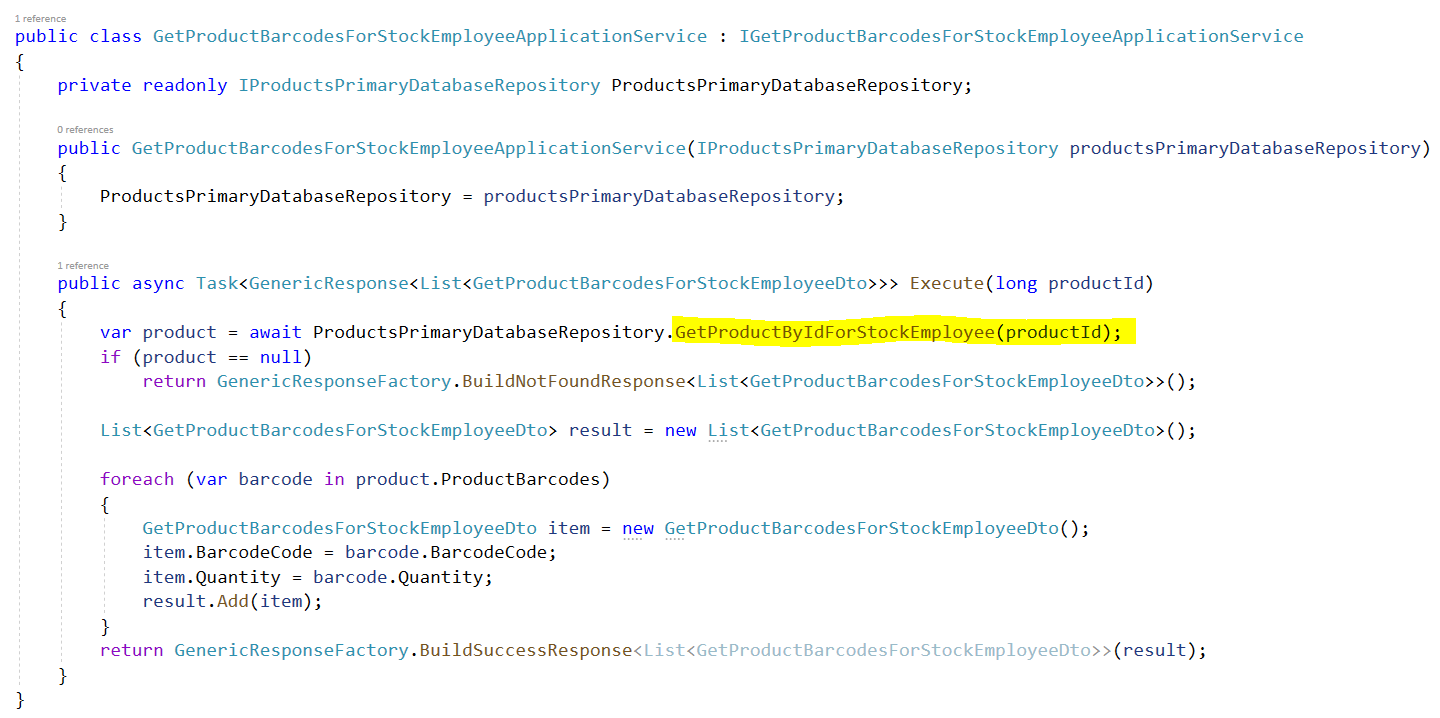
So, I can know the quantity of the product barcode code.

ProductsRepository.cs



Application Service





The application service dose not change or modify the data returned from the repository , just return it to display for the stock employee , so this application service is read only operation , so loading part of the aggregation in this case can not cause any problem.

**User Story 6 :**

A person driving a forklift

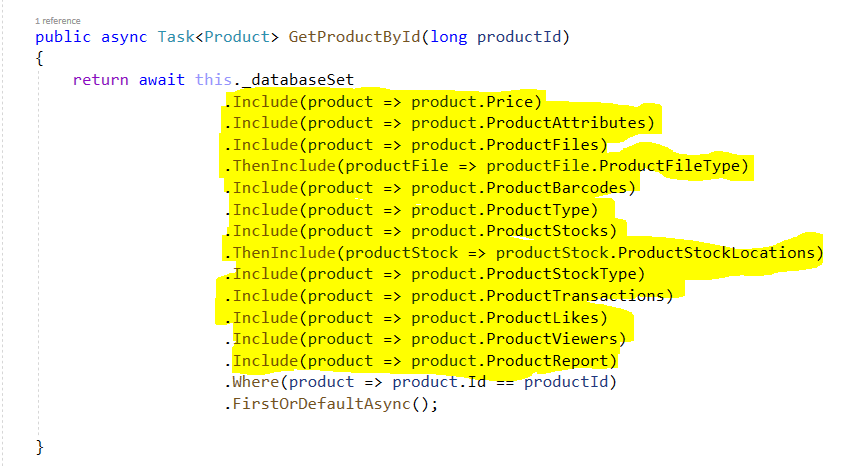
Description automatically generated

As stock employee,

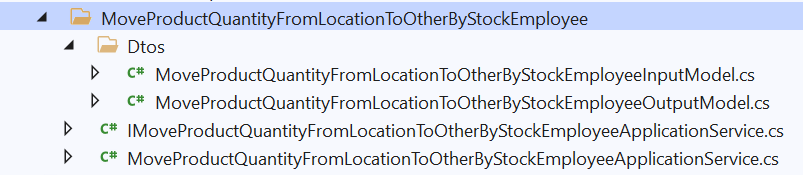
I want to move some quantity of product from location to another location inside the same stock

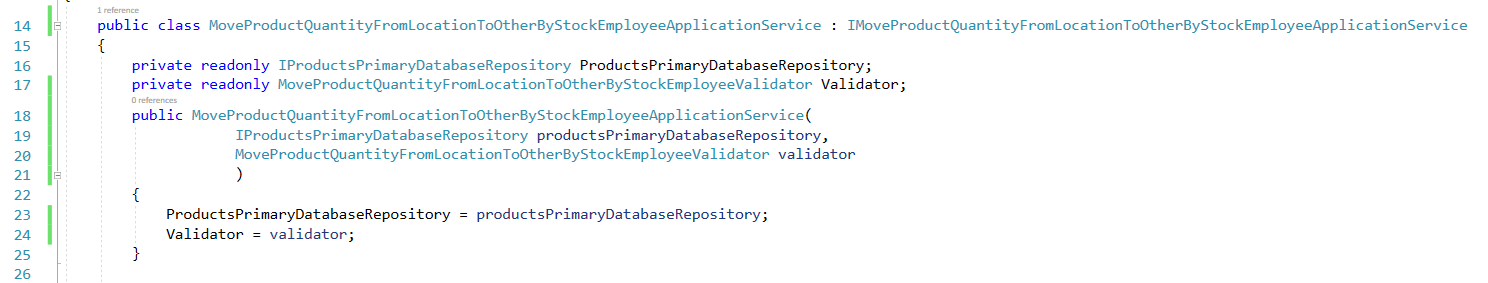
So, I can balance the quantity of the product on different locations

ProductsRepository.cs



Application Service







The application service is this case update or modify the aggregation root , and then save this update again on the database , so the application service use GetProductById on the repository pattern to load the whole aggregation.

* The third opinion :

Completely reject to load part of the aggregation in any case , instead of this design a good and small aggregation with small number of objects as possible , the section Repository pattern with bounded contexts cover this topic in details with examples.

but still after designing aggregation with small numbers of objects , there can be some performance issues when children objects have high number of items and here similar example from real life work experience :

**Invoice is a aggregation with a small number of objects ( 4 objects ) :**

A screenshot of a computer

Description automatically generated with low confidence

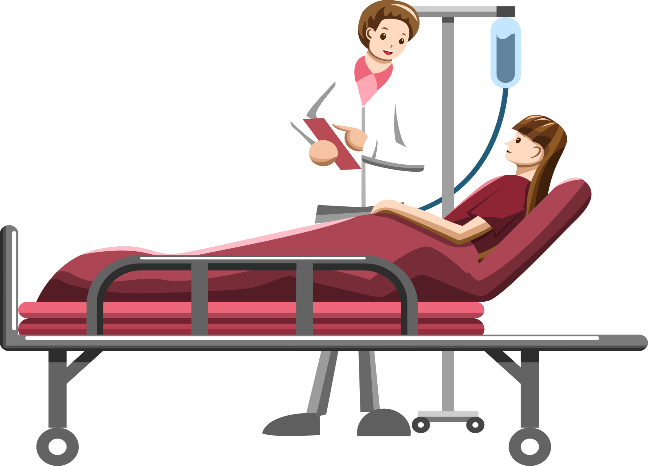
InvoicesRepository.cs

A picture containing text, screenshot, font

Description automatically generated

In most cases the GetInvoiceById is good and has not any problems , but in some scenario this method cause performance issues for example :

**Case Study :**



A patient who stayed in the hospital for three months, took a lot of medicines and other services, the number of invoice items reached more than 6000.

**User Story :**

Cartoon of a nurse holding a clipboard

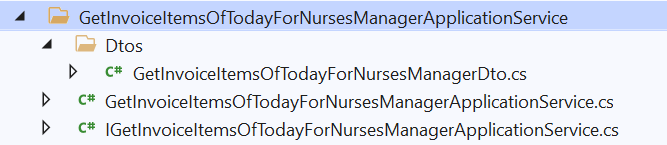
Description automatically generated with low confidence

As nurses manger,

I want to see the invoice items of today only for my patient.

So, I can know the drugs and other services of my patient for today.

Application Service



A screenshot of a computer

Description automatically generated with low confidence

Loading all invoice items ( 6000 ) for the three month for this patient from the database and then filtering the invoice items on the application server memory cause performance issues for our systems , the system is follow monolithic and multi-tenancy architecture with single SQL Server database , specially there are more than 1000 concurrent employees for 10 hospitals in three different countries and capacity for about 3000 patients and 100 visitors per minute on the application and website.

A picture containing circle, black and white, coil spring, spring

Description automatically generated

* try always to make the filtration of data in the database not on the application server
* try always to load the minimum data as possible from the database
* try always to minimize database round trips as possible.
* Try always to make the query is simple as possible , no a lot of joins , no business logic inside the query.

To solve this problem and after searching and discuss with the architects and research teams , we have three different solutions :

First Solution ( Simplest )

Apply the second option : Read Only Operations for every role or use case , we choose this solution although it can break the second principle of the repository pattern , but it is the simplest one.

A light bulb with a black background

Description automatically generated with low confidence

InvoicesRepository.cs

A picture containing text, screenshot, font, line

Description automatically generated

Application Service

A screenshot of a computer

Description automatically generated with low confidence

A light bulb with a black background

Description automatically generated with low confidence

Second Solution ( Excellent )

Using CQRS pattern , and we will discuss it in depth in the next chapter.

using CQRS pattern is Excellent for this case as we will see in the next chapter.

A light bulb with a black background

Description automatically generated with low confidence

Third Solution ( Very good for specific cases )

When most of the invoice items number per invoice is very low, we can add acceptance criteria to limit the max number of the invoice items per invoice for example 100 , and if the patient need more drugs or services create new invoice and calculate the total at end.

A light bulb with a black background

Description automatically generated with low confidence

Fourth Solution ( Good for specific cases )

Make Invoice Item as a Separate Aggregation.

This solution is good when the most or average of invoice items number per invoice is high.

But it isn’t very good because this can conflict with the concept of aggregation as we discuss before.

Invoice item will be a separate aggregation :

A screenshot of a computer

Description automatically generated with low confidence

Invoice item will be have a separate repository :

A screenshot of a computer

Description automatically generated with medium confidence

InvoiceItemsRepository.cs

A screenshot of a computer code

Description automatically generated with low confidence

Application Service

A screenshot of a computer

Description automatically generated with low confidence

Designing a good bounded contexts can help to solve the first part of this problem by instead of having single fat aggregation and single fat repository , the project will having different bounded contexts with different versions of aggregations and repositories based on the business requirements or in the words of domain driven design based on the subdomains and the ubiquitous language.

but still can in the same bounded context if the business logic is complex , the problem will be still exist and here the cqrs pattern to solve second part of the problem.

We will discuss the details of all this points in the next sections.