Backend Engineering in Practice

By / Ahmed Khalifa

Part 1 :
Architecture patterns and principles

Ch 1 : Clean architecture

1. what is clean architecture ?
2. why clean architecture ?
3. how to design and build system in clean architecture ?

Ch 2 : Domain driven design

1. what is Domain driven design ?
2. why domain driven design ?
3. domain driven design basic building blocks :
    - 3.1 aggregation and aggregation root.
    - 3.2 entity
    - 3.3 value object
    - 3.4 domain services
    - 3.5 infrastructure services
    - 3.6 application services

Ch 3 : The importance of the bounded contexts

1. Legacy systems
2. What is bounded context ?
3. Why bounded context ?
4. Design and implementation of bounded context
5. Communications between different aggregations
6. Bounded contexts and database mapping

Part 2 : Database patterns

Ch 1 : Repository pattern

1. What is repository pattern ?
2. Why repository pattern ?
3. Repository pattern implementation
4. Repository pattern main principles
5. Problems with repository pattern :
    - 5.1 Fat repository
    - 5.2 Fat aggregation
    - 5.3 Performance issues with read only operations

Ch 2 : CQRS pattern

1. What is CQRS Pattern ?
2. Why CQRS Pattern ?
    1. Single of responsibility
    2. Separation of concern
    3. Independent Scalability
    4. Efficient Performance for database read operations.
    5. Efficient with system that has different users and different roles
3. Types and Implementation of CQRS Pattern :
    1. Single database
        1.1 Single Database Single Project
        1.2 Single Database Two Projects
    2. Multiple databases
        2.1 Same types
        2.2 different types

What is clean architecture ?

- Clean architecture is a architecture pattern that created by **Robert C. Martin**.
- It is extend to other domain centric architecture patterns like onion , hexagonal patterns.
- Usually clean architecture is applied with domain driven design principles.
- The core ideas of all domain centric architecture patterns :
  1. The business logic is the core of the system
  2. The business logic does not depend on any frameworks like user interface (UI) , database or any other infrastructures.
  3. Any changes on the UI , database or other infrastructure should not touch the business logic.

Koshary architecture :

To know how the clean architecture can be useful , we will first explore legacy architecture pattern which I see in a lot of old and real projects , I will called it “koshary pattern”.

Koshary is a delicious Egyptian food that contains a mixture of different food elements in single dish.

Koshary architecture pattern is refer to the architecture pattern which single component (usually user interface component) contain the implementation of all things like : business logic , database , notifications and .. etc.

Koshary architecture

To better understanding, let's see that with real user story and actual project with code sample :

User story : Add new employee by manager

As a manager,

I want to add new employee to the system,

So I can save important information about the employee like
- personal information : name , birth date
- contact info : email , phone number
- address : country , city , street
- skills

1.User story inputs :

Data	Data type
Employee Name	string
Employee Birthdate	Datetime
Email	string
Phone number	string
Country	string
City	string
Street and Building Number	string
Skills	List

2.User story business logic :

2.1 logic Workflow :

User	System
Manager opens screen called “Add new Employee”.	System display “Add new Employee” screen.
Manager enters the information of the employee on the UI elements and inputs.	-
Manager click on button called “Add new Employee”	System will : 1. if employee information is valid , then system will save the new employee and display success message 2. if employee information is not valid , then system will not save the new employee and display errors message

2.2 Business rules :

1. Employee age should be equal or greater than 21
2. Employee email must has correct format like test@test.com but the system should not accept other incorrect formats like test@test , test_test.com.
3. Employee phone number is required and must be unique , system can not have two employees with the same phone number.
4. Employee must has at min 1 skill.

3. User story outputs :

Case	Result
Case 1 : input by manager apply the business rules	1. Save new employee on database 2. Send Email Notification to Admin 3. Display success message on the screen for the manager
Case 2 : Input by manager does not apply the business rules	1. Display errors message on the screen for the manager

4. User story UI/UX :

4.1 Default screen :

4.2 Screen with input errors and message :

4.3 Screen with success message :

Chapter 1

Problems with repository pattern

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.

The aggregations are live on the Domain.Core

Let's take the product aggregation as example :

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 :

```
9   namespace Domain.InfrastructureInterfaces.Databases.PrimaryDatabase.Repositories
10  {
        6 references
11      public interface IProductsPrimaryDatabaseRepository : IGenericRepository<Product,long>
12      {
13          #region Write Operations
            1 reference
14          Task<Product> AddProduct(Product newProduct);
            1 reference
15          Task<Product> UpdateProduct(Product product);
            1 reference
16          Task<Product> DeleteProduct(Product product);
17          #endregion
18
19          #region Read Operations
            2 references
20          Task<Product> GetProductById(long productId);
            1 reference
21          Task<List<Product>> GetProductsList(ListParameters<Product> listParameters);
            1 reference
22          Task<List<Product>> GetProductsByCategory(long categoryId);
            1 reference
23          Task<List<Product>> GetProductsThatHaveDiscount(ListParameters<Product> listParameters);
            1 reference
24          Task<List<Product>> GetProductsThatHaveAvailableQuntityOnTheStock(ListParameters<Product> listParameters);
            1 reference
25          Task<List<Product>> GetProductsThatHaveExpiryDate(ListParameters<Product> listParameters);
            1 reference
26          Task<List<Product>> GetTopLikedProducts(ListParameters<Product> listParameters);
            1 reference
27          Task<List<Product>> GetTopSellingProducts(ListParameters<Product> listParameters);
            1 reference
28          Task<List<Product>> GetTopRatedProducts(ListParameters<Product> listParameters);
29          #endregion
30      }
31  }
```

The actual implementation of the repository pattern is live on

```
13  namespace Infrastructure.Database.SqlServer.Repositories
14  {
        2 references
15      public class ProductsSqlServerDatabaseRepository : GenericRepository<Product,long> , IProductsPrimaryDatabaseRepository
16      {
            0 references
17          public ProductsSqlServerDatabaseRepository(DatabaseContext databaseContext) ...
21
22          #region Write Operations
            1 reference
23          public Task<Product> AddProduct(Product newProduct)...
            1 reference
27          public Task<Product> DeleteProduct(Product product)...
            1 reference
31          public Task<Product> UpdateProduct(Product product)...
35          #endregion
36
37          #region Read Operations
            2 references
38          public async Task<Product> GetProductById(long productId)...
            1 reference
52          public Task<List<Product>> GetProductsByCategory(long categoryId)...
            1 reference
56          public Task<List<Product>> GetProductsList(ListParameters<Product> listParameters)...
            1 reference
60          public Task<List<Product>> GetProductsThatHaveAvailableQuntityOnTheStock(ListParameters<Product> listParameters)...
            1 reference
64          public Task<List<Product>> GetProductsThatHaveDiscount(ListParameters<Product> listParameters)...
            1 reference
68          public Task<List<Product>> GetProductsThatHaveExpiryDate(ListParameters<Product> listParameters)...
            1 reference
72          public Task<List<Product>> GetTopLikedProducts(ListParameters<Product> listParameters)...
            1 reference
76          public Task<List<Product>> GetTopRatedProducts(ListParameters<Product> listParameters)...
            1 reference
80          public Task<List<Product>> GetTopSellingProducts(ListParameters<Product> listParameters)...
84          #endregion
85      }
86  }
```

Repository pattern principles

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

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)

```
17   public ProductsSqlServerDatabaseRepository(DatabaseContext databaseContext) ...
21   #region Write Operations
     1 reference
22   public Task<Product> AddProduct(Product newProduct)...
     1 reference
26   public Task<Product> DeleteProduct(Product product)...
     1 reference
30   public Task<Product> UpdateProduct(Product product)...
     0 references
34   public Task<ProductAttribute> AddProductAttribute(ProductAttribute productAttribute)...
     0 references
38   public Task<ProductAttribute> UpdateProductAttribute(ProductAttribute productAttribute)...
     0 references
42   public Task<ProductAttribute> DeleteProductAttribute(ProductAttribute productAttribute)...
46   #endregion
47   #region Read Operations
     2 references
48   public async Task<Product> GetProductById(long productId)...
     1 reference
62   public Task<List<Product>> GetProductsByCategory(long categoryId)...
     1 reference
66   public Task<List<Product>> GetProductsList(ListParameters<Product> listParameters)...
     1 reference
70   public Task<List<Product>> GetProductsThatHaveAvailableQuntityOnTheStock(ListParameters<Product> listParameters)...
     1 reference
74   public Task<List<Product>> GetProductsThatHaveDiscount(ListParameters<Product> listParameters)...
     1 reference
78   public Task<List<Product>> GetProductsThatHaveExpiryDate(ListParameters<Product> listParameters)...
     1 reference
82   public Task<List<Product>> GetTopLikedProducts(ListParameters<Product> listParameters)...
     1 reference
86   public Task<List<Product>> GetTopRatedProducts(ListParameters<Product> listParameters)...
     1 reference
90   public Task<List<Product>> GetTopSellingProducts(ListParameters<Product> listParameters)...
     0 references
94   public Task<List<ProductAttribute>> GetProductAttributesByProductId(long productId)...
     0 references
98   public Task<ProductTax> GetProductTaxByProductId(long productId)...
     0 references
102  public Task<ProductNameAndPriceDto> GetProductNameAndPriceByProductId(long productId)...
106  #endregion
```

4- 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

```
0 references
public IQueryable<Product> GetAll()
{
    return _databaseContext.Products.AsQueryable();
}
```

You need to get golden products in two different services

Application Service 1

```
List<Product> goldenProducts =  ProductsPrimaryDatabaseRepository
                                .GetAll()
                                .Where(product =>
                                       product.TotalCountOfLikes > 1000
                                    && product.TotalCountOfViewers > 10000)
                                .ToList();
```

Application Service 2

```
List<Product> goldenProducts =  ProductsPrimaryDatabaseRepository
                                .GetAll()
                                .Where(product =>
                                       product.TotalCountOfLikes > 1000
                                    && product.TotalCountOfViewers > 10000)
                                .ToList();
```

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

```
List<Product> goldenProducts =  ProductsPrimaryDatabaseRepository
                                .GetAll()
                                .Where(product =>
                                       product.TotalCountOfLikes > 1000
                                    && product.TotalCountOfViewers > 10000
                                    && product.TotalMoneyOfSales > 100)
                                .ToList();
```

Application Service 2

```
List<Product> goldenProducts =  ProductsPrimaryDatabaseRepository
                                .GetAll()
                                .Where(product =>
                                       product.TotalCountOfLikes > 1000
                                    && product.TotalCountOfViewers > 10000
                                    && product.TotalMoneyOfSales > 100)
                                .ToList();
```

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

```
2 references
public List<Product> GetAllGoldenProducts()
{
    return  _databaseContext
            .Products
            .Where( product =>
                    product.TotalCountOfLikes > 1000
                && product.TotalCountOfViewers > 10000
                && product.TotalMoneyOfSales > 100)
            .ToList();
}
```

Application Service 1

```
List<Product> goldenProducts = ProductsPrimaryDatabaseRepository.GetAllGoldenProducts();
```

Application Service 2

```
List<Product> goldenProducts = ProductsPrimaryDatabaseRepository.GetAllGoldenProducts();
```

5- Repository pattern can return native or primitive data types like numbers , Boolean

```
0 references
public int GetCountOfProductsThatHaveDiscount()
{
    return _databaseContext
            .Products
            .Where(product => product.HasDiscount == true)
            .Count();
}

0 references
public bool CheckIsProductNameExist(string productName)
{
    return _databaseContext
            .Products
            .Where(product => product.Name == productName)
            .Any();
}
```

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.

2- 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

```
0 references
public void AddNewOrderItem(long productId , string productName , decimal productPrice)
{
    bool isOrderItemExist = OrderItems.Where(o => o.ProductId == productId).Count() > 0;


    if(isOrderItemExist)
    {
        string errorMessage = "the order item for product : " + productName + " is already exist before ";
        throw new Exception(errorMessage);
    }

    var newOrderItem = new OrderItem(productId, productPrice);

    this.OrderItems.Add(newOrderItem);

    this.TotalPrice += productPrice;
}
```

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.

<u>This mean the repository must load the whole aggregation.</u>

[ProductsRepository.cs]

```
1 reference
public async Task<Product> GetProductById(long productId)
{
    return await this._databaseSet
                    .Include(product => product.Price)
                    .Include(product => product.ProductAttributes)
                    .Include(product => product.ProductFiles)
                    .ThenInclude(productFile => productFile.ProductFileType)
                    .Include(product => product.ProductBarcodes)
                    .Include(product => product.ProductType)
                    .Include(product => product.ProductStocks)
                    .ThenInclude(productStock => productStock.ProductStockLocations)
                    .Include(product => product.ProductStockType)
                    .Include(product => product.ProductTransactions)
                    .Include(product => product.ProductLikes)
                    .Include(product => product.ProductViewers)
                    .Include(product => product.ProductReport)
                    .Where(product => product.Id == productId)
                    .FirstOrDefaultAsync();

}
```

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

```
0 references
public async Task<Product> GetProductWithProductAttributesOnly(long productId)
{
    return await this._databaseSet
                    .Include(product => product.ProductAttributes)
                    .Where(product => product.Id == productId)
                    .FirstOrDefaultAsync();

}
```

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

```
1 reference
public async Task<Product> GetProductById(long productId)
{
    return await this._databaseSet
                    .Include(product => product.Price)
                    .Include(product => product.ProductAttributes)
                    .Include(product => product.ProductFiles)
                    .ThenInclude(productFile => productFile.ProductFileType)
                    .Include(product => product.ProductBarcodes)
                    .Include(product => product.ProductType)
                    .Include(product => product.ProductStocks)
                    .ThenInclude(productStock => productStock.ProductStockLocations)
                    .Include(product => product.ProductStockType)
                    .Include(product => product.ProductTransactions)
                    .Include(product => product.ProductLikes)
                    .Include(product => product.ProductViewers)
                    .Include(product => product.ProductReport)
                    .Where(product => product.Id == productId)
                    .FirstOrDefaultAsync();

}
```

User Story 2 :

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

```
1 reference
public async Task<Product> GetProductById(long productId)
{
    return await this._databaseSet
                    .Include(product => product.Price)
                    .Include(product => product.ProductAttributes)
                    .Include(product => product.ProductFiles)
                    .ThenInclude(productFile => productFile.ProductFileType)
                    .Include(product => product.ProductBarcodes)
                    .Include(product => product.ProductType)
                    .Include(product => product.ProductStocks)
                    .ThenInclude(productStock => productStock.ProductStockLocations)
                    .Include(product => product.ProductStockType)
                    .Include(product => product.ProductTransactions)
                    .Include(product => product.ProductLikes)
                    .Include(product => product.ProductViewers)
                    .Include(product => product.ProductReport)
                    .Where(product => product.Id == productId)
                    .FirstOrDefaultAsync();

}
```

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

```
1 reference
public async Task<Product> GetProductById(long productId)
{
    return await this._databaseSet
                    .Include(product => product.Price)
                    .Include(product => product.ProductAttributes)
                    .Include(product => product.ProductFiles)
                    .ThenInclude(productFile => productFile.ProductFileType)
                    .Include(product => product.ProductBarcodes)
                    .Include(product => product.ProductType)
                    .Include(product => product.ProductStocks)
                    .ThenInclude(productStock => productStock.ProductStockLocations)
                    .Include(product => product.ProductStockType)
                    .Include(product => product.ProductTransactions)
                    .Include(product => product.ProductLikes)
                    .Include(product => product.ProductViewers)
                    .Include(product => product.ProductReport)
                    .Where(product => product.Id == productId)
                    .FirstOrDefaultAsync();

}
```

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.

The code and implementation of user story 4

ProductsRepository.cs

```
0 references
public ProductsSqlServerDatabaseRepository(DatabaseContext databaseContext) ...

Write Operations

#region Read Operations
1 reference
public async Task<Product> GetProductById(long productId)...
1 reference
public Task<List<Product>> GetProductsByCategory(long categoryId)...
1 reference
public Task<List<Product>> GetProductsList(ListParameters<Product> listParameters)...
1 reference
public Task<List<Product>> GetProductsThatHaveAvailableQuntityOnTheStock(ListParameters<Product> listParameters)...
1 reference
public Task<List<Product>> GetProductsThatHaveDiscount(ListParameters<Product> listParameters)...
1 reference
public Task<List<Product>> GetProductsThatHaveExpiryDate(ListParameters<Product> listParameters)...
1 reference
public Task<List<Product>> GetTopLikedProducts(ListParameters<Product> listParameters)...
1 reference
public Task<List<Product>> GetTopRatedProducts(ListParameters<Product> listParameters)...
1 reference
public Task<List<Product>> GetTopSellingProducts(ListParameters<Product> listParameters)...

#endregion
```

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

```
1 reference
public async Task<Product> GetProductById(long productId)
{
    return await this._databaseSet
                    .Include(product => product.Price)
                    .Include(product => product.ProductAttributes)
                    .Include(product => product.ProductFiles)
                    .ThenInclude(productFile => productFile.ProductFileType)
                    .Include(product => product.ProductBarcodes)
                    .Include(product => product.ProductType)
                    .Include(product => product.ProductStocks)
                    .ThenInclude(productStock => productStock.ProductStockLocations)
                    .Include(product => product.ProductStockType)
                    .Include(product => product.ProductTransactions)
                    .Include(product => product.ProductLikes)
                    .Include(product => product.ProductViewers)
                    .Include(product => product.ProductReport)
                    .Where(product => product.Id == productId)
                    .FirstOrDefaultAsync();

}
```

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

```
1 reference
public class GetProductReportForAdminApplicationService : IGetProductReportForAdminApplicationService
{
    private readonly IProductsPrimaryDatabaseRepository ProductsPrimaryDatabaseRepository;

    0 references
    public GetProductReportForAdminApplicationService(IProductsPrimaryDatabaseRepository productsPrimaryDatabaseRepository)
    {
        ProductsPrimaryDatabaseRepository = productsPrimaryDatabaseRepository;
    }

    1 reference
    public async Task<GenericResponse<GetProductReportForAdminReportDto>> Execute(long productId)
    {
        var product = await ProductsPrimaryDatabaseRepository.GetProductById(productId);

        if (product == null)
            return GenericResponseFactory.BuildNotFoundResponse<GetProductReportForAdminReportDto>();

        var reportDto = new GetProductReportForAdminReportDto();
        reportDto.TotalCountOfLikes = product.ProductReport.TotalCountOfLikes;
        reportDto.TotalCountOfViewers = product.ProductReport.TotalCountOfViewers;
        reportDto.TotalMoneyOfSales = product.ProductReport.TotalMoneyOfSales;
        reportDto.Rate = product.ProductReport.Rate;

        return GenericResponseFactory.BuildSuccessResponse<GetProductReportForAdminReportDto>(reportDto);

    }
}
```

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.

The code and implementation of second option :

ProductsRepository.cs

```
#region Read Operations
2 references
public async Task<Product> GetProductById(long productId)...
1 reference
public async Task<Product> GetProductByIdForClient(long productId)...
1 reference
public async Task<Product> GetProductByIdForSalesEmployee(long productId)...
1 reference
public async Task<Product> GetProductByIdForStockEmployee(long productId)...
1 reference
public async Task<Product> GetProductByIdForAdmin(long productId)...
```

The details of every method :

```
1 reference
public async Task<Product> GetProductByIdForClient(long productId)
{
    return await this._databaseSet
                    .Include(product => product.Price)
                    .Include(product => product.ProductAttributes)
                    .Include(product => product.ProductFiles)
                    .ThenInclude(productFile => productFile.ProductFileType)
                    .Where(product => product.Id == productId)
                    .FirstOrDefaultAsync();
}

1 reference
public async Task<Product> GetProductByIdForStockEmployee(long productId)
{
    return await this._databaseSet
                    .Include(product => product.ProductBarcodes)
                    .Include(product => product.ProductStocks)
                    .ThenInclude(productStock => productStock.ProductStockLocations)
                    .Where(product => product.Id == productId)
                    .FirstOrDefaultAsync();
}

1 reference
public async Task<Product> GetProductByIdForSalesEmployee(long productId)
{
    return await this._databaseSet
                    .Include(product => product.ProductLikes)
                    .Include(product => product.ProductViewers)
                    .Include(product => product.ProductReport)
                    .Where(product => product.Id == productId)
                    .FirstOrDefaultAsync();

}
```

The problems with the second option :

This way break the second principle 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.

  So 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 :

As stock employee,

I want to know the product barcodes

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

ProductsRepository.cs

```
1 reference
public async Task<Product> GetProductByIdForStockEmployee(long productId)
{
    return await this._databaseSet
                    .Include(product => product.ProductBarcodes)
                    .Include(product => product.ProductStocks)
                    .ThenInclude(productStock => productStock.ProductStockLocations)
                    .Where(product => product.Id == productId)
                    .FirstOrDefaultAsync();
}
```

Application Service

```
1 reference
public class GetProductBarcodesForStockEmployeeApplicationService : IGetProductBarcodesForStockEmployeeApplicationService
{
    private readonly IProductsPrimaryDatabaseRepository ProductsPrimaryDatabaseRepository;

    0 references
    public GetProductBarcodesForStockEmployeeApplicationService(IProductsPrimaryDatabaseRepository productsPrimaryDatabaseRepository)
    {
        ProductsPrimaryDatabaseRepository = productsPrimaryDatabaseRepository;
    }

    1 reference
    public async Task<GenericResponse<List<GetProductBarcodesForStockEmployeeDto>>> Execute(long productId)
    {
        var product = await ProductsPrimaryDatabaseRepository.GetProductByIdForStockEmployee(productId);
        if (product == null)
            return GenericResponseFactory.BuildNotFoundResponse<List<GetProductBarcodesForStockEmployeeDto>>();

        List<GetProductBarcodesForStockEmployeeDto> result = new List<GetProductBarcodesForStockEmployeeDto>();

        foreach (var barcode in product.ProductBarcodes)
        {
            GetProductBarcodesForStockEmployeeDto item = new GetProductBarcodesForStockEmployeeDto();
            item.BarcodeCode = barcode.BarcodeCode;
            item.Quantity = barcode.Quantity;
            result.Add(item);
        }
        return GenericResponseFactory.BuildSuccessResponse<List<GetProductBarcodesForStockEmployeeDto>>(result);
    }
}
```

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 :

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

```
1 reference
public async Task<Product> GetProductById(long productId)
{
    return await this._databaseSet
                    .Include(product => product.Price)
                    .Include(product => product.ProductAttributes)
                    .Include(product => product.ProductFiles)
                    .ThenInclude(productFile => productFile.ProductFileType)
                    .Include(product => product.ProductBarcodes)
                    .Include(product => product.ProductType)
                    .Include(product => product.ProductStocks)
                    .ThenInclude(productStock => productStock.ProductStockLocations)
                    .Include(product => product.ProductStockType)
                    .Include(product => product.ProductTransactions)
                    .Include(product => product.ProductLikes)
                    .Include(product => product.ProductViewers)
                    .Include(product => product.ProductReport)
                    .Where(product => product.Id == productId)
                    .FirstOrDefaultAsync();

}
```

Application Service

- MoveProductQuantityFromLocationToOtherByStockEmployee
  - Dtos
    - MoveProductQuantityFromLocationToOtherByStockEmployeeInputModel.cs
    - MoveProductQuantityFromLocationToOtherByStockEmployeeOutputModel.cs
  - IMoveProductQuantityFromLocationToOtherByStockEmployeeApplicationService.cs
  - MoveProductQuantityFromLocationToOtherByStockEmployeeApplicationService.cs

```
1 reference
public class MoveProductQuantityFromLocationToOtherByStockEmployeeApplicationService : IMoveProductQuantityFromLocationToOtherByStockEmployeeApplicationService
{
    private readonly IProductsPrimaryDatabaseRepository ProductsPrimaryDatabaseRepository;
    private readonly MoveProductQuantityFromLocationToOtherByStockEmployeeValidator Validator;
    0 references
    public MoveProductQuantityFromLocationToOtherByStockEmployeeApplicationService(
                IProductsPrimaryDatabaseRepository productsPrimaryDatabaseRepository,
                MoveProductQuantityFromLocationToOtherByStockEmployeeValidator validator
                )
    {
        ProductsPrimaryDatabaseRepository = productsPrimaryDatabaseRepository;
        Validator = validator;
    }

    1 reference
    public async Task<GenericResponse<MoveProductQuantityFromLocationToOtherByStockEmployeeOutputModel>> Execute
                                    (MoveProductQuantityFromLocationToOtherByStockEmployeeInputModel inputModel)
    {
        var product = await ProductsPrimaryDatabaseRepository.GetProductById(inputModel.ProductId);
        if (product == null)
            return GenericResponseFactory.BuildNotFoundResponse<MoveProductQuantityFromLocationToOtherByStockEmployeeOutputModel>();

        var validatorResult = Validator.Check(product,inputModel);
        if(validatorResult.Success == false)
            return GenericResponseFactory
                    .BuildBadRequestResponse<MoveProductQuantityFromLocationToOtherByStockEmployeeOutputModel>(validatorResult.Errors);

        product.DecrementProductQuantityAtStockLocation(inputModel.StockId, inputModel.FromLocationId, inputModel.Quantity);
        product.IncrementProductQuantityAtStockLocation(inputModel.StockId, inputModel.ToLocationId, inputModel.Quantity);
        product.AddNewProductTransaction(inputModel.StockId, inputModel.FromLocationId, inputModel.ToLocationId, inputModel.Quantity);

        await ProductsPrimaryDatabaseRepository.UpdateProduct(product);

        MoveProductQuantityFromLocationToOtherByStockEmployeeOutputModel result = new MoveProductQuantityFromLocationToOtherByStockEmployeeOutputModel();
        result.NewQuantityOnSourceLocation = product.GetProductQuantityAtStockLocation(inputModel.StockId, inputModel.FromLocationId);
        result.NewQuantityOnDestinationLocation = product.GetProductQuantityAtStockLocation(inputModel.StockId, inputModel.ToLocationId);

        return GenericResponseFactory
                .BuildSuccessResponse<MoveProductQuantityFromLocationToOtherByStockEmployeeOutputModel>(result);
    }
}
```

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) :

InvoicesRepository.cs

```
0 references
public async Task<Invoice> GetInvoiceById(long invoiceId)
{
    return await this._databaseSet
                    .Include(invoice => invoice.InvoiceItems)
                    .Include(invoice => invoice.InvoicePaymentType)
                    .Include(invoice => invoice.InvoiceType)
                    .Where(invoice => invoice.Id == invoiceId)
                    .FirstOrDefaultAsync();
}
```

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 :

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

- GetInvoiceItemsOfTodayForNursesManagerApplicationService
  - Dtos
    - GetInvoiceItemsOfTodayForNursesManagerDto.cs
  - GetInvoiceItemsOfTodayForNursesManagerApplicationService.cs
  - IGetInvoiceItemsOfTodayForNursesManagerApplicationService.cs

```
1 reference
public class GetInvoiceItemsOfTodayForNursesManagerApplicationService : IGetInvoiceItemsOfTodayForNursesManagerApplicationService
{
    private readonly IInvoicesRepository InvoicesRepository;
    0 references
    public GetInvoiceItemsOfTodayForNursesManagerApplicationService(IInvoicesRepository invoicesRepository)
    {
        InvoicesRepository = invoicesRepository;
    }
    1 reference
    public async Task<GenericResponse<List<GetInvoiceItemsOfTodayForNursesManagerDto>>> Execute(long invoiceId)
    {
        var invoice = await InvoicesRepository.GetInvoiceById(invoiceId);
        if (invoice == null)
            return GenericResponseFactory.BuildNotFoundResponse<List<GetInvoiceItemsOfTodayForNursesManagerDto>>();

        List<InvoiceItem> todayInvoiceItems = invoice.InvoiceItems.Where(i => i.CreatedDate.Date == DateTime.Today).ToList();

        List<GetInvoiceItemsOfTodayForNursesManagerDto> result = new List<GetInvoiceItemsOfTodayForNursesManagerDto>();

        foreach (var invoiceItem in invoice.InvoiceItems)
        {
            GetInvoiceItemsOfTodayForNursesManagerDto item = new GetInvoiceItemsOfTodayForNursesManagerDto();
            item.ItemName = invoiceItem.Name;
            item.ItemPrice = invoiceItem.Price;
            result.Add(item);
        }
        return GenericResponseFactory.BuildSuccessResponse<List<GetInvoiceItemsOfTodayForNursesManagerDto>>(result);
    }
}
```

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.

- 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.

InvoicesRepository.cs

```
0 references
public async Task<Invoice> GetInvoiceWithTodayInvoiceItemsOnlyForNursesManagerById(long invoiceId)
{
    return await this._databaseSet
                    .Include(invoice => invoice.InvoiceItems.Where(i => i.CreatedDate.Date == DateTime.Today))
                    .Where(invoice => invoice.Id == invoiceId)
                    .FirstOrDefaultAsync();
}
```

Application Service

```
1 reference
public class GetInvoiceItemsOfTodayForNursesManagerApplicationService : IGetInvoiceItemsOfTodayForNursesManagerApplicationService
{
    private readonly IInvoicesRepository InvoicesRepository;
    0 references
    public GetInvoiceItemsOfTodayForNursesManagerApplicationService(IInvoicesRepository invoicesRepository)
    {
        InvoicesRepository = invoicesRepository;
    }
    1 reference
    public async Task<GenericResponse<List<GetInvoiceItemsOfTodayForNursesManagerDto>>> Execute(long invoiceId)
    {
        var invoice = await InvoicesRepository.GetInvoiceWithTodayInvoiceItemsForNursesManager(invoiceId);
        if (invoice == null)
            return GenericResponseFactory.BuildNotFoundResponse<List<GetInvoiceItemsOfTodayForNursesManagerDto>>();

        List<GetInvoiceItemsOfTodayForNursesManagerDto> result = new List<GetInvoiceItemsOfTodayForNursesManagerDto>();

        foreach (var invoiceItem in invoice.InvoiceItems)
        {
            GetInvoiceItemsOfTodayForNursesManagerDto item = new GetInvoiceItemsOfTodayForNursesManagerDto();
            item.ItemName = invoiceItem.Name;
            item.ItemPrice = invoiceItem.Price;
            result.Add(item);
        }
        return GenericResponseFactory.BuildSuccessResponse<List<GetInvoiceItemsOfTodayForNursesManagerDto>>(result);
    }
}
```

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.

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.

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 :

Invoice item will be have a separate repository :

InvoiceItemsRepository.cs

```
0 references
public async Task<InvoiceItem> GetInvoiceItemById(long invoiceItemId)
{
    return await this._databaseSet
                    .Where(invoiceItem => invoiceItem.Id == invoiceItemId)
                    .FirstOrDefaultAsync();
}

0 references
public async Task<List<InvoiceItem>> GetAllInvoiceItemsForInvoice(long invoiceId)
{
    return await this._databaseSet
                    .Where(invoiceItem => invoiceItem.InvoiceId == invoiceId)
                    .ToListAsync();
}

0 references
public async Task<List<InvoiceItem>> GeTodayInvoiceItemsByInvoiceId(long invoiceId)
{
    return await this._databaseSet
                    .Where(invoiceItem => invoiceItem.InvoiceId == invoiceId
                                    && invoiceItem.CreatedDate.Date == DateTime.Today)
                    .ToListAsync();
}
```

Application Service

```
1 reference
public class GetInvoiceItemsOfTodayForNursesManagerApplicationService : IGetInvoiceItemsOfTodayForNursesManagerApplicationService
{
    private readonly IInvoiceItemsRepository InvoiceItemsRepository;
    0 references
    public GetInvoiceItemsOfTodayForNursesManagerApplicationService(IInvoiceItemsRepository invoiceItemsRepository)
    {
        InvoiceItemsRepository = invoiceItemsRepository;
    }
    1 reference
    public async Task<GenericResponse<List<GetInvoiceItemsOfTodayForNursesManagerDto>>> Execute(long invoiceId)
    {
        var todayInvoiceItems = await InvoiceItemsRepository.GeTodayInvoiceItemsByInvoiceId(invoiceId);

        List<GetInvoiceItemsOfTodayForNursesManagerDto> result = new List<GetInvoiceItemsOfTodayForNursesManagerDto>();

        foreach (var invoiceItem in todayInvoiceItems)
        {
            GetInvoiceItemsOfTodayForNursesManagerDto item = new GetInvoiceItemsOfTodayForNursesManagerDto();
            item.ItemName = invoiceItem.Name;
            item.ItemPrice = invoiceItem.Price;
            result.Add(item);
        }
        return GenericResponseFactory.BuildSuccessResponse<List<GetInvoiceItemsOfTodayForNursesManagerDto>>(result);
    }
}
```

Repository pattern and bounded contexts

For now the system dose not have any bounded contexts or sub-domains.

Designing a good bounded contexts and a small aggregation is very important specially when the business logic of the system is very complex and the system has different users with different roles.

so instead of having single fat aggregation and single fat repository per aggregation root, 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 sub-domains and the ubiquitous language.

Bounded context	Roles	Aggregations	Features / User Stories Examples
Product Catalog	-- Client	- Client - Product	- GetTopSellingProductsForClient - GetTopLikedProductsForClient - GetTopViewedProductsForClient - GetProductDetailsForClient
Client Preferences	--- Client	- Client - Product - ProductLike - ProductView	- LikeProductByClient - DislikeProductByClient - IncrementProductViewsByClient
Ordering	--- Client	- Client - Product - Order	- CreateNewOrderByClient - AddNewProductToOrderByClient - IncrementQuantityOfProductAtOrderByClient - DecrementQuantityOfProductAtOrderByClient - RemoveProductFromOrderByClient
Supply Chain	-- Stock Employee -- Stock Manager -- Admin	- Buyer - Product - Order - Employee - Stock - SupplierCompany - SupplierEmployee - SupplierInvoice	- GetAllNewOrdersForStockManager - AcceptOrderByStockManager - AssignOrderToStockEmployeeByStockManager - ReceiveOrderByStockEmployee - AddProductFromStockToOrderByStockEmployee - ReturnProductFromOrderToStockByStockEmployee - CompleteOrderByStockEmployee - GetProductBarcodeLocationsForStockEmployee - MoveProductQuantityFromLocationToOtherByStockEmployee
Shipping	-- Client -- Delivery Employee	- Client - Product - Order - Employee - ShippingCompany	- DeliveryEmployeeTakeOrderFromStock - GetClientContactInfoForDeliveryEmployee - GetDeliveryEmployeeLocationOnMapForClient - ClientReceiveOrderSuccessfully - ClientRejectOrder

		- ShippingEmployee	
Marketing	-- Marketing Employee -- Marketing Manager -- Admin	- Client - Product - Order - Employee	- GetAllClientsWhoLikedProductForMarketingEmployee - GetTopProductsWhichOrderedByClientForMarketingEmployee - GenerateReportAboutTodayOrdersForMarketingManager

Solution Explorer
Search Solution Explorer (Ctrl+ك)
Solution 'ApplicationSolution' (40 of 40 projects)
Back-End
Marketing
Ordering
ProductsCatalog
Shared
Shipping
SupplyChain
Front-End

Solution Explorer
Search Solution Explorer (Ctrl+ك)
Solution 'ApplicationSolution' (38 of 38 projects)
Back-End
Marketing
Ordering
ProductsCatalog
Shared
Api
Api.Gateway
Domain
Infrastructure
Infrastructure.BackgroundServices.HangfireClient
Infrastructure.BackgroundServices.HangfireServer
Infrastructure.EmailsServices
Infrastructure.NotificationsServices
Infrastructure.SmsServices
Shipping
SupplyChain
Front-End

Marketing
Domain
Domain.ApplicationServices
Domain.Core
Domain.DomainServices
Domain.InfrastructureInterfaces
Infrastructure
Infrastructure.Database.SqlServer

Ordering
Domain
Domain.ApplicationServices
Domain.Core
Domain.DomainServices
Domain.InfrastructureInterfaces
Infrastructure
Infrastructure.Database.Redis
Infrastructure.Database.SqlServer

ProductsCatalog
Domain
Domain.ApplicationServices
Domain.Core
Domain.DomainServices
Domain.InfrastructureInterfaces
Infrastructure
Infrastructure.Database.Redis
Infrastructure.Database.SqlServer

Shipping
Domain
Domain.ApplicationServices
Domain.Core
Domain.DomainServices
Domain.InfrastructureInterfaces
Infrastructure
Infrastructure.Database.SqlServer

SupplyChain
Domain
Domain.ApplicationServices
Domain.Core
Domain.DomainServices
Domain.InfrastructureInterfaces
Infrastructure
Infrastructure.Database.Redis
Infrastructure.Database.SqlServer

SupplyChain
Domain
Domain.ApplicationServices
Domain.Core
Dependencies
Clients
Emplyees
Orders
Products
Shared
Domain.DomainServices
Domain.InfrastructureInterfaces
Infrastructure

Products
Product.cs
ProductBarcode.cs
ProductPrice.cs
ProductStock.cs
ProductStockLocation.cs
ProductStockType.cs
ProductTax.cs
ProductTransaction.cs
ProductType.cs

Solution Explorer
Search Solution Explorer (Ctrl+ك)
Solution 'ApplicationSolution' (38 of 38 projects)
Back-End
Marketing
Ordering
ProductsCatalog
Shared
Api
Domain
Domain.ApplicationServices
Domain.Core
Dependencies
AggregationRoot.cs
AggregationRootFullAudit.cs
Entity.cs
EntityFullAudit.cs
ValueObject.cs
Domain.DomainServices
Domain.InfrastructureInterfaces
Infrastructure
Shipping
SupplyChain
Front-End

```
1   using Domain.Core.Shared;
2   using System;
3   using System.Collections.Generic;
4   using System.Linq;
5   using System.Text;
6   using System.Threading.Tasks;
7
8   namespace SupplyChain.Domain.Core.Products
9   {
        2 references
10      public class Product : AggregationRootFullAudit<long>
11      {
```

- SupplyChain
  - Domain
    - Domain.ApplicationServices
    - Domain.Core
    - Domain.DomainServices
    - Domain.InfrastructureInterfaces
      - Dependencies
      - Databases
        - PrimaryDatabase
          - Repositories
            - ClientsRepository
            - EmployeesRepository
            - OrdersRepository
            - ProductsRepository
              - IProductsPrimaryDatabaseRepository.cs
  - Infrastructure

```
1   using Domain.InfrastructureInterfaces.Databases.Shared;
2   using SupplyChain.Domain.Core.Products;
3   using System;
4   using System.Collections.Generic;
5   using System.Linq;
6   using System.Text;
7   using System.Threading.Tasks;
8
9   namespace SupplyChain.Domain.InfrastructureInterfaces.Databases.PrimaryDatabase.Repositories
10  {
        0 references
11      public interface IProductsPrimaryDatabaseRepository
12      {
13          Write Operations
18
19          #region Read Operations
            0 references
20          Task<Product> GetProductById(long productId);
            0 references
21          Task<Product> GetProductByIdForStockEmployee(long productId);
            0 references
22          Task<Product> GetProductByIdForStockManager(long productId);
            0 references
23          Task<List<Product>> GetProductsList(ListParameters<Product> listParameters);
24          #endregion
25
26      }
27  }
28
```

- SupplyChain
  - Domain
  - Infrastructure
    - Infrastructure.Database.SqlServer
      - Dependencies
      - DatabaseContext
        - SupplyChainDatabaseContext.cs
      - Migrations
      - ModelConfigurations
      - QueryBuilders
      - Repositories
        - ClientsRepository
        - EmployeesRepository
        - OrdersRepository
        - ProductsRepository
          - ProductsPrimaryDatabaseRepository.cs

```
12  namespace SupplyChain.Infrastructure.Database.SqlServer.Repositories
13  {
        1 reference
14      public class ProductsSqlServerDatabaseRepository : IProductsPrimaryDatabaseRepository
15      {
16          private readonly SupplyChainDatabaseContext SupplyChainDatabaseContext;
            0 references
17          public ProductsSqlServerDatabaseRepository(SupplyChainDatabaseContext supplyChainDatabaseContext)
18          {
19              SupplyChainDatabaseContext = supplyChainDatabaseContext;
20          }
21          Write Operations
47          #region Read Operations
            0 references
48          public async Task<Product> GetProductById(long productId)
49          {
50              return await this.SupplyChainDatabaseContext
51                              .Products
52                              .Include(product => product.Price)
53                              .Include(product => product.ProductBarcodes)
54                              .Include(product => product.ProductType)
55                              .Include(product => product.ProductStocks)
56                              .ThenInclude(productStock => productStock.ProductStockLocations)
57                              .Include(product => product.ProductStockType)
58                              .Include(product => product.ProductTransactions)
59                              .Where(product => product.Id == productId)
60                              .FirstOrDefaultAsync();
61          }
            0 references
62          public async Task<Product> GetProductByIdForStockEmployee(long productId)...
            0 references
71          public async Task<Product> GetProductByIdForStockManager(long productId)...
            0 references
80          public Task<List<Product>> GetProductsList(ListParameters<Product> listParameters)...
84          #endregion
85      }
86
```

- SupplyChain
  - Domain
    - Domain.ApplicationServices
      - Dependencies
      - StockEmployee
        - AddProductFromStockToOrderByStockEmployee
        - CompleteOrderByStockEmployee
        - GetProductBarcodeLocationsForStockEmployee
        - MoveProductQuantityFromLocationToOtherByStockEmployee
        - ReceiveOrderByStockEmployee
        - ReturnProductFromOrderToStockByStockEmployee
      - StockManager
        - AcceptOrderByStockManager
        - AssignOrderToStockEmployeeByStockManager
        - GetAllNewOrdersForStockManager
    - Domain.Core
    - Domain.DomainServices
    - Domain.InfrastructureInterfaces
  - Infrastructure

Domain
Domain.ApplicationServices
Dependencies
OrdersManagementModule
AcceptOrder
AddProductFromStockToOrder
AssignOrder
CompleteOrder
GetAllNewOrders
ReceiveOrder
ProductsTransactionsModule
GetProductBarcodeLocations
MoveProductQuantityFromLocationToOther
ReturnProductFromOrder
Domain.Core
Domain.DomainServices
Domain.InfrastructureInterfaces
Infrastructure

Marketing
Domain
Domain.ApplicationServices
Domain.Core
Domain.DomainServices
Domain.InfrastructureInterfaces
Infrastructure
Infrastructure.Database.SqlServer

```
7   namespace Marketing.Domain.Core.Products
8   {
        0 references
9       public class ProductReport
10      {
            0 references
11          public decimal TotalCountOfViewers { get; private set; }
            0 references
12          public decimal TotalCountOfLikes { get; private set; }
            0 references
13          public long TotalNumberOfOrders { get; private set; }
            0 references
14          public decimal TotalMoneyOfSales { get; private set; }
            0 references
15          public double Rate { get; private set; }
16      }
17  }
```

```
9   namespace Marketing.Domain.InfrastructureInterfaces.Databases.PrimaryDatabase.Repositories
10  {
        0 references
11      public interface IProductsPrimaryDatabaseRepository
12      {
13          #region Read Operations
            0 references
14          Task<Product> GetProductById(long productId);
            0 references
15          Task<List<Product>> GetProductsList(ListParameters<Product> listParameters);
            0 references
16          Task<List<Product>> GetTopLikedProducts(ListParameters<Product> listParameters);
            0 references
17          Task<List<Product>> GetTopViewedProducts(ListParameters<Product> listParameters);
            0 references
18          Task<List<Product>> GetTopRatedProducts(ListParameters<Product> listParameters);
19          #endregion
20      }
21  }
```

```
8   namespace Marketing.Domain.InfrastructureInterfaces.Databases.PrimaryDatabase.Repositories
9   {
        0 references
10      public interface IProductLikesPrimaryDatabaseRepository
11      {
12
13          #region Read Operations
            0 references
14          Task<List<ProductLike>> GetAllProductsLikedByClient(long clientId);
            0 references
15          Task<List<ProductLike>> GetAllProductsLikedOnDateRange(DateTime from , DateTime to);
            0 references
16          Task<List<ProductLike>> GetTopClientsWhoLikesProducts();
            0 references
17          Task<List<ProductLike>> GetTopDislikedProducts();
18          #endregion
19
20      }
21  }
```

```
9   namespace Marketing.Domain.InfrastructureInterfaces.Databases.PrimaryDatabase.Repositories
10  {
        0 references
11      public interface IProductViewsPrimaryDatabaseRepository
12      {
13          #region Read Operations
            0 references
14          Task<List<ProductViewer>> GetAllProductsViewedByClient(long clientId);
            0 references
15          Task<List<ProductViewer>> GetAllProductsViewedOnDateRange(DateTime from, DateTime to);
            0 references
16          Task<List<ProductViewer>> GetTopClientsWhoViewProducts();
            0 references
17          Task<List<ProductViewer>> GetTopClientsWhoViewProduct(long productId);
18          #endregion
19      }
20  }
```

Marketing
Domain
Infrastructure
Infrastructure.Database.SqlServer
Dependencies
DatabaseContext
MarketingDatabaseContext.cs
Migrations
ModelConfigurations
QueryBuilders
Repositories
ClientsRepository
EmployeesRepository
OrderEmployeesRepository
OrderItemsRepository
OrdersRepository
ProductLikesRepository
ProductLikesPrimaryDatabaseRepository.cs
ProductsRepository
ProductsPrimaryDatabaseRepository.cs
ProductViewersRepository
ProductViewersPrimaryDatabaseRepository.cs

```
12  namespace Marketing.Infrastructure.Database.SqlServer.Repositories
13  {
        1 reference
14      public class ProductsSqlServerDatabaseRepository : IProductsPrimaryDatabaseRepository
15      {
16          private readonly MarketingDatabaseContext MarketingDatabaseContext;
            0 references
17          public ProductsSqlServerDatabaseRepository(MarketingDatabaseContext databaseContext)
18          {
19              MarketingDatabaseContext = databaseContext;
20          }
            0 references
21          public async Task<Product> GetProductById(long productId)
22          {
23              return await MarketingDatabaseContext
24                          .Products
25                          .Include(p => p.Price)
26                          .Include(p => p.ProductReport)
27                          .Where(p => p.Id == productId)
28                          .FirstOrDefaultAsync();
29          }
            0 references
30          public async Task<List<Product>> GetProductsList(ListParameters<Product> listParameters)
31          {
32              return await  MarketingDatabaseContext
33                           .Products
34                           .Include(p => p.Price)
35                           .Include(p => p.ProductReport)
36                           .Skip((listParameters.PageNumber - 1) * listParameters.PageSize)
37                           .Take(listParameters.PageSize)
38                           .ToListAsync();
39          }

            0 references
40          public async Task<List<Product>> GetTopLikedProducts(ListParameters<Product> listParameters)
41          {
42              return await MarketingDatabaseContext
43                          .Products
44                          .Include(p => p.Price)
45                          .Include(p => p.ProductReport)
46                          .Skip((listParameters.PageNumber - 1) * listParameters.PageSize)
47                          .Take(listParameters.PageSize)
48                          .OrderByDescending( p => p.ProductReport.TotalCountOfLikes)
49                          .ToListAsync();
50          }
51
            0 references
52          public async Task<List<Product>> GetTopViewedProducts(ListParameters<Product> listParameters)
53          {
54              return await MarketingDatabaseContext
55                         .Products
56                         .Include(p => p.Price)
57                         .Include(p => p.ProductReport)
58                         .Skip((listParameters.PageNumber - 1) * listParameters.PageSize)
59                         .Take(listParameters.PageSize)
60                         .OrderByDescending(p => p.ProductReport.TotalCountOfViewers)
61                         .ToListAsync();
62          }
63
64      }
```

Marketing
Domain
Domain.ApplicationServices
Dependencies
MarketingEmployee
GetAllClientsWhoLikedProductForMarketingEmployeeApplicationService
GetAllClientsWhoMakeOrdersForMarketingEmployeeApplicationService
GetTopOrdersByDateTimeForMarketingEmployeeApplicationService
GetTopOrdersByTotalCostForMarketingEmployeeApplicationService
MarketingManager
GenerateReportAboutLastYearOrdersForMarketingManagerApplicationService
GetTopLikedProductsForMarketingManagerApplicationService
GetTopSellingProductsForMarketingManagerApplicationService
GetTopViewedProductsForMarketingManagerApplicationService
Domain.Core
Domain.DomainServices
Domain.InfrastructureInterfaces
Infrastructure

Marketing
Domain
Domain.ApplicationServices
Dependencies
ClientsAnalysis
GetAllClientsWhoLikedProductApplicationService
GetAllClientsWhoMakeOrdersApplicationService
OrdersAnalysis
GenerateReportAboutLastYearOrdersApplicationService
GetTopOrdersByDateTimeApplicationService
GetTopOrdersByTotalCostApplicationService
ProductsAnalysis
GetTopLikedProductsApplicationService
GetTopSellingProductsApplicationService
GetTopViewedProductsApplicationService
Domain.Core
Domain.DomainServices
Domain.InfrastructureInterfaces
Infrastructure

Communications between different aggregations in the same bounded context

General rules of the communications :

Aggregation should be always accessed via It's Aggregation Root only from any place on the application like repository , another Aggregation in the same bound context or different bounded context.

Case 1 : Valid

Aggregation root can access another Aggregation root

Case 2 : Valid

Normal Entity or Object can access another Aggregation root

Case 3 : Not Valid

Aggregation root can not access normal entity in another Aggregation directly

Case 4 : Valid

Aggregation root can access normal entity in another Aggregation via its Aggregation root

Case 5 : Not Valid

Normal Entity can not access normal entity in another Aggregation directly

Case 6 : Valid

Normal Entity or Object in Aggregation can access normal entity in another Aggregation via its Aggregation root

A) Communications for Read operations :

a lot of business requirements or user stories need data from different aggregations in the same application service like

User Story :

As marketing employee,

I want to know the products and clients who liked these products At specific time.

So, I can extract some analysis information.

Product name	Client name	Client age	Like datetime
iphone 14	lila	21	1-7-2023 1:00AM
iphone 14	nancy	22	1-7-2023 1:30AM
iphone 14	adam	19	2-7-2023 2:30AM
Le nove labtob	michael	27	1-7-2023 2:00PM
Le nove labtob	ahmed	28	3-7-2023 1:00PM

This user story need data from three different aggregations ProductLikes , Products and Clients aggregations , they are three main methods to achive that, every method has its own advantages and disadvantages.

The first method :
using repository pattern for each aggregation when each aggregation has not any navigation reference to other aggregations.

Marketing Bounded Context

First aggregation

```
namespace Marketing.Domain.Core.ProductLikes
{
    17 references
    public class ProductLike : AggregationRootFullAudit<long>
    {
        2 references
        public long ProductId { get; private set; }
        2 references
        public long ClientId { get; private set; }

        Operations

    }
}
```

Second aggregation

```
namespace Marketing.Domain.Core.Products
{
    3 references
    public class Product : AggregationRootFullAudit<long>
    {
        0 references
        public string Name { get; private set; }
        0 references
        public string Description { get; private set; }
        0 references
        public bool HasDiscount { get; private set; }
        0 references
        public decimal Discount { get; private set; }
        4 references
        public ProductPrice Price { get; private set; }
        6 references
        public ProductReport ProductReport { get; private set; }

        Operations
    }
}
```

Third aggregation

```
namespace Marketing.Domain.Core.Clients
{
    4 references
    public class Client : AggregationRootFullAudit<long>
    {
        1 reference
        public string Name { get; private set; }
        0 references
        public DateTime BirthDate { get; private set; }
        1 reference
        public long TotalNumberOfAllOrders { get; private set; }
        0 references
        public decimal TotalMoneyOfAllOrders { get; private set; }
        0 references
        public long TotalProductsLikes { get; private set; }
        0 references
        public long TotalProductsViews { get; private set; }
        0 references
        public bool IsActive { get; private set; }

        private readonly List<ClientAddress> _clientAddresses = new List<ClientAddress>();
        0 references
        public IReadOnlyCollection<ClientAddress> ClientAddresses => _clientAddresses.AsReadOnly();

        private readonly List<ClientNationality> _clientNationalities = new List<ClientNationality>();
        0 references
        public IReadOnlyCollection<ClientNationality> ClientNationalities => _clientNationalities.AsReadOnly();

        Operations
    }
}
```

First aggregation ProductLikes :

```
10  namespace Marketing.Domain.Core.ProductLikes
11  {
        17 references
12      public class ProductLike : AggregationRootFullAudit<long>
13      {
            2 references
14          public long ProductId { get; private set; }
            2 references
15          public long ClientId { get; private set; }
16
17          Operations
19
20      }
21  }
```

Second aggregation Products :

```
8   namespace Marketing.Domain.Core.Products
9   {
        3 references
10      public class Product : AggregationRootFullAudit<long>
11      {
            0 references
12          public string Name { get; private set; }
            0 references
13          public string Description { get; private set; }
            0 references
14          public bool HasDiscount { get; private set; }
            0 references
15          public decimal Discount { get; private set; }
            4 references
16          public ProductPrice Price { get; private set; }
            6 references
17          public ProductReport ProductReport { get; private set; }
18
19          Operations
21      }
22  }
23
```

Third aggregation Clients :

```
8   namespace Marketing.Domain.Core.Clients
9   {
        4 references
10      public class Client : AggregationRootFullAudit<long>
11      {
            1 reference
12          public string Name { get; private set; }
            0 references
13          public DateTime BirthDate { get; private set; }
            1 reference
14          public long TotalNumberOfAllOrders { get; private set; }
            0 references
15          public decimal TotalMoneyOfAllOrders { get; private set; }
            0 references
16          public long TotalProductsLikes { get; private set; }
            0 references
17          public long TotalProductsViews { get; private set; }
            0 references
18          public bool IsActive { get; private set; }
19
20          private readonly List<ClientAddress> _clientAddresses = new List<ClientAddress>();
            0 references
21          public IReadOnlyCollection<ClientAddress> ClientAddresses => _clientAddresses.AsReadOnly();
22
23          private readonly List<ClientNationality> _clientNationalities = new List<ClientNationality>();
            0 references
24          public IReadOnlyCollection<ClientNationality> ClientNationalities => _clientNationalities.AsReadOnly();
25
26          Operations
32      }
33  }
```

ApplicationService :

```
13  namespace Marketing.Domain.ApplicationServices.ProductLikeAnalysis
14  {
        1 reference
15      public class GetAllProductsAndClientsWhoLikedByDataTimeRangeApplicationService
16                  :IGetAllProductsAndClientsWhoLikedByDataTimeRangeApplicationService
17      {
18          private readonly IProductLikesPrimaryDatabaseRepository ProductLikesPrimaryDatabaseRepository;
19          private readonly IProductsPrimaryDatabaseRepository ProductsPrimaryDatabaseRepository;
20          private readonly IClientsPrimaryDatabaseRepository ClientsPrimaryDatabaseRepository;
21          private readonly ClientBirthDateToAgeDomainService ClientBirthDateToAgeDomainService;
            0 references
22          public GetAllProductsAndClientsWhoLikedByDataTimeRangeApplicationService(
23                      IProductLikesPrimaryDatabaseRepository productLikesPrimaryDatabaseRepository,
24                      IProductsPrimaryDatabaseRepository productsPrimaryDatabaseRepository,
25                      IClientsPrimaryDatabaseRepository clientsPrimaryDatabaseRepository,
26                      ClientBirthDateToAgeDomainService clientBirthDateToAgeDomainService)
27          {
28              ProductLikesPrimaryDatabaseRepository = productLikesPrimaryDatabaseRepository;
29              ProductsPrimaryDatabaseRepository = productsPrimaryDatabaseRepository;
30              ClientsPrimaryDatabaseRepository = clientsPrimaryDatabaseRepository;
31              ClientBirthDateToAgeDomainService = clientBirthDateToAgeDomainService;
32          }
33
```

```
34          public async Task<GenericResponse<List<GetAllProductsAndClientsWhoLikedByDataTimeRangeOutputModel>>>
                        1 reference
35                      Execute(GetAllProductsAndClientsWhoLikedByDataTimeRangeInputModel inputModel)
36          {
37              var productLikes = await ProductLikesPrimaryDatabaseRepository
38                                      .GetAllByCreatedDatetimeRange(inputModel.From,inputModel.To);
39
40              List<long> productsIds = productLikes.Select(p => p.ProductId).Distinct().ToList();
41              var productsList = await ProductsPrimaryDatabaseRepository
42                                      .GetProductsListByIds(productsIds);
43              var productsDictionary = productsList.ToDictionary(product => product.Id , product => product);
44
45              List<long> clientsIds = productLikes.Select(p => p.ClientId).Distinct().ToList();
46              var clientsList = await ClientsPrimaryDatabaseRepository
47                                      .GetClientsListByIds(clientsIds);
48              var clientsDictionary = clientsList.ToDictionary(client => client.Id, client => client);
49
```

```
49
50          var result = new List<GetAllProductsAndClientsWhoLikedByDataTimeRangeOutputModel>();
51          foreach (var productLike in productLikes)
52          {
53              var product = productsDictionary[productLike.ProductId];
54              var client =  clientsDictionary[productLike.ClientId];
55
56              var item = new GetAllProductsAndClientsWhoLikedByDataTimeRangeOutputModel();
57              item.ProductId = product.Id;
58              item.ProductName = product.Name;
59              item.ClientId = client.Id;
60              item.ClientName = client.Name;
61              item.ClientAge = ClientBirthDateToAgeDomainService.Convert(client.BirthDate);
62              item.LikedDatetime = productLike.CreatedAt;
63              result.Add(item);
64          }
65
66          return GenericResponseFactory
67                .BuildSuccessResponse<List<GetAllProductsAndClientsWhoLikedByDataTimeRangeOutputModel>>(result);
68      }
69  }
70 }
```

Disadvantages of the first method :

1- High latency :

The response time of application service will be increased due to there are a sperate network call or round trip to fetch every aggegation seperatly.

Advantages of the first method :

1- ddd prefer to isolate the different the aggregations from each other , beause this help to keep the aggregations small as possible and prevents a lot of conflicts when the different aggregations reference each other's.

2- Every repository sends simple query to fetch its aggregation instead of sending one complex query that has a lot of SQL joins to fetch all aggregations together in one network call or round trip.

The second method :
using repository pattern for one of the aggregation and fetch the other aggregations.

Marketing Bounded Context

First aggregation

```
10  namespace Marketing.Domain.Core.ProductLikes
11  {
        17 references
12      public class ProductLike : AggregationRootFullAudit<long>
13      {
            2 references
14          public long ProductId { get; private set; }
            0 references
15          public Product Product { get; }
            2 references
16          public long ClientId { get; private set; }
            0 references
17          public Client Client { get; }
18
19          Operations
21
22      }
23  }
```

Second aggregation

```
8   namespace Marketing.Domain.Core.Products
9   {
        3 references
10      public class Product : AggregationRootFullAudit<long>
11      {
            0 references
12          public string Name { get; private set; }
            0 references
13          public string Description { get; private set; }
            0 references
14          public bool HasDiscount { get; private set; }
            0 references
15          public decimal Discount { get; private set; }
            4 references
16          public ProductPrice Price { get; private set; }
            6 references
17          public ProductReport ProductReport { get; private set; }
18
19          Operations
21      }
22  }
23
```

Third aggregation

```
8   namespace Marketing.Domain.Core.Clients
9   {
        4 references
10      public class Client : AggregationRootFullAudit<long>
11      {
            1 reference
12          public string Name { get; private set; }
            0 references
13          public DateTime BirthDate { get; private set; }
            1 reference
14          public long TotalNumberOfAllOrders { get; private set; }
            0 references
15          public decimal TotalMoneyOfAllOrders { get; private set; }
            0 references
16          public long TotalProductsLikes { get; private set; }
            0 references
17          public long TotalProductsViews { get; private set; }
            0 references
18          public bool IsActive { get; private set; }
19
20          private readonly List<ClientAddress> _clientAddresses = new List<ClientAddress>();
            0 references
21          public IReadOnlyCollection<ClientAddress> ClientAddresses => _clientAddresses.AsReadOnly();
22
23          private readonly List<ClientNationality> _clientNationalities = new List<ClientNationality>();
            0 references
24          public IReadOnlyCollection<ClientNationality> ClientNationalities => _clientNationalities.AsReadOnly();
25
26          Operations
32      }
33  }
```

ProductLikes Aggregation :

```
namespace Marketing.Domain.Core.ProductLikes
{
    17 references
    public class ProductLike : AggregationRootFullAudit<long>
    {
        2 references
        public long ProductId { get; private set; }
        0 references
        public Product Product { get; }
        2 references
        public long ClientId { get; private set; }
        0 references
        public Client Client { get; }

        Operations

    }
}
```

ProductLikeRepository :

```
2 references
public async Task<List<ProductLike>> GetAllByCreatedDatetimeRange(DateTime from , DateTime to)
{
    return await MarketingDatabaseContext
                .ProductLikes
                .Where(p => p.CreatedAt >= from && p.CreatedAt <= to)
                .ToListAsync();
}

1 reference
public async Task<List<ProductLike>> GetAllByCreatedDatetimeRangeWithProductAndClient(DateTime from, DateTime to)
{
    return await MarketingDatabaseContext
                .ProductLikes
                .Include(p => p.Product)
                .Include(p => p.Client )
                .Where(p => p.CreatedAt >= from && p.CreatedAt <= to)
                .ToListAsync();
}
```

Application Service :

```
13  namespace Marketing.Domain.ApplicationServices.ProductLikeAnalysis
14  {
        1 reference
15      public class GetAllProductsAndClientsWhoLikedByDataTimeRangeApplicationService
16                  :IGetAllProductsAndClientsWhoLikedByDataTimeRangeApplicationService
17      {
18          private readonly IProductLikesPrimaryDatabaseRepository ProductLikesPrimaryDatabaseRepository;
19          private readonly ClientBirthDateToAgeDomainService ClientBirthDateToAgeDomainService;
            0 references
20          public GetAllProductsAndClientsWhoLikedByDataTimeRangeApplicationService(
21                      IProductLikesPrimaryDatabaseRepository productLikesPrimaryDatabaseRepository,
22                      ClientBirthDateToAgeDomainService clientBirthDateToAgeDomainService)
23          {
24              ProductLikesPrimaryDatabaseRepository = productLikesPrimaryDatabaseRepository;
25              ClientBirthDateToAgeDomainService = clientBirthDateToAgeDomainService;
26          }
27
```

```
28          public async Task<GenericResponse<List<GetAllProductsAndClientsWhoLikedByDataTimeRangeOutputModel>>>
                        1 reference
29                      Execute(GetAllProductsAndClientsWhoLikedByDataTimeRangeInputModel inputModel)
30          {
31              var productLikes = await ProductLikesPrimaryDatabaseRepository
32                                      .GetAllByCreatedDatetimeRangeWithProductAndClient(inputModel.From,inputModel.To);
33
34              var result = new List<GetAllProductsAndClientsWhoLikedByDataTimeRangeOutputModel>();
35              foreach (var productLike in productLikes)
36              {
37                  var product = productLike.GetTheProduct();
38                  var client =  productLike.GetTheClient();
39
40                  var item = new GetAllProductsAndClientsWhoLikedByDataTimeRangeOutputModel();
41                  item.ProductId = product.Id;
42                  item.ProductName = product.Name;
43                  item.ClientId = client.Id;
44                  item.ClientName = client.Name;
45                  item.ClientAge = ClientBirthDateToAgeDomainService.Convert(client.BirthDate);
46                  item.LikedDatetime = productLike.CreatedAt;
47                  result.Add(item);
48              }
49
50              return GenericResponseFactory
51                    .BuildSuccessResponse<List<GetAllProductsAndClientsWhoLikedByDataTimeRangeOutputModel>>(result);
52          }
53      }
54  }
```

Advantages of the second method :

1- lower latency :
single network call to database to fetch all aggegations.

Disadvantages of the second method :

1- if there are a lot of different aggregations, this mean there will be a lot of SQL joins which cause the query is very complex and then sending seperatly query as the first method may be better for the performane , so developers need to trade off between the two methods based on the number and size of the aggregations.

2- If the application use no sql database like Mongodb or redis which dose not support join directly as sql databases , so the developers need to find some workaround solutions for this problem , you can read about embedding vs

references documents on mongodb , and redis lua script for more details.

3- Some opinions which I read on some articles say " its okay or acceptable to load the aggregation by reference from the other aggregation but the repository should load also the complete other aggregation , not only the aggregation root or partial view from the aggregation to ensure the consistency "
there are no complete confirmation for the correctness of this opinion but I can not say it correct or not and I will try to search and read more about this conflict point , but my personal idea : " i agree with this on write-read operations but I dose not agree on read only operations."

Example of loading the whole product and client aggregation on ProductLikesRepository :

```
1 reference
public async Task<List<ProductLike>> GetAllByCreatedDatetimeRange(DateTime from , DateTime to)
{
    return await MarketingDatabaseContext
                .ProductLikes
                .Where(p => p.CreatedAt >= from && p.CreatedAt <= to)
                .ToListAsync();
}
```

```
2 references
public async Task<List<ProductLike>> GetAllByCreatedDatetimeRangeWithProductAndClient(DateTime from, DateTime to)
{
    return await MarketingDatabaseContext
                .ProductLikes
                .Include( productLike => productLike.Product)
                .ThenInclude( product => product.Price)
                .ThenInclude( product => product.ProductReport)
                .Include( productLike => productLike.Client)
                .ThenInclude( client => client.ClientAddresses)
                .ThenInclude( client => client.ClientNationalities)
                .Where(p => p.CreatedAt >= from && p.CreatedAt <= to)
                .ToListAsync();
}
```

The whole product aggregation

The whole client aggregation

This will lead to load a lot of unused data and objects and make the query more complex and slower and also the developers need to update the other repositories when new object is added to the other aggregation.

The third method :
using CQRS pattern.

As we note , repository pattern has a lot of advantages and one of this main advantages is ensure consistency for the aggregation on write or read-write operations but also repository pattern has a lot of problems with read only operations like load unused objects or attributes and can has performance issue and query slower specially where the business logic is very complex.

On the other side CQRS pattern solve all this problems with read only operations , we will discuss CQRS Pattern in details next chapter.

B) Communications for Write operations :

From the domain driven design perfect view : “ single transaction will be represented by single aggregation “

But in a lot of business requirements or user stories , the changing on one of the aggregation can effect on one or more other aggregations.

User Story :

As stock employee,

I want to move the quantity of product order item from stock to order.

So, I can decrement the product quantity from the stock location and increase my bonus points.

Let’s discuss the business on more details and see the workflow of the business :

When client create order , the product quantity of each order item will be not decremented automatically from the stock until the stock employee move it from the stock location to the order , for example if the stock contain 10 quantity of product like le novo laptop and one of the client create order with 2 product quantity of le novo laptop , the current balance of le novo laptop will be remain 10 until the stock employee already take two 2 laptop and add them to the client order.

The company to increase the productivity of the stock employees give money bonus (10 $) to every stock employee

when he completes every 100 product moving from stock location to order.

So in this user story , the changing on order aggregation will be effect on both product and employee aggregations.

Actually there are tow main methods of the communication to achieve this.

The first method :
direct or normal communications

Application Service :

```
11  namespace SupplyChain.Domain.ApplicationServices.OrdersManagementModule
12  {
        1 reference
13      public class AddProductFromStockToOrderApplicationService : IAddProductFromStockToOrderApplicationService
14      {
15          private readonly AddProductFromStockToOrderValidator Validator;
16          private readonly IOrdersPrimaryDatabaseRepository OrdersPrimaryDatabaseRepository;
17          private readonly IProductsPrimaryDatabaseRepository ProductsPrimaryDatabaseRepository;
18          private readonly IEmployeesPrimaryDatabaseRepository EmployeesPrimaryDatabaseRepository;
19          private readonly IUnitOfWork UnitOfWork;
            0 references
20          public AddProductFromStockToOrderApplicationService(
21              AddProductFromStockToOrderValidator addProductFromStockToOrderValidator,
22              IOrdersPrimaryDatabaseRepository ordersPrimaryDatabaseRepository,
23              IProductsPrimaryDatabaseRepository productsPrimaryDatabaseRepository,
24              IEmployeesPrimaryDatabaseRepository employeesPrimaryDatabaseRepository,
25              IUnitOfWork unitOfWork)
26          {
27              Validator = addProductFromStockToOrderValidator;
28              OrdersPrimaryDatabaseRepository = ordersPrimaryDatabaseRepository;
29              ProductsPrimaryDatabaseRepository = productsPrimaryDatabaseRepository;
30              EmployeesPrimaryDatabaseRepository = employeesPrimaryDatabaseRepository;
31              UnitOfWork = unitOfWork;
32          }
```

```
            1 reference
33          public async Task<GenericResponse<AddProductFromStockToOrderOutputModel>> Execute(AddProductFromStockToOrderInputModel inputModel)
34          {
35              var validatorResult = Validator.Check(inputModel);
36
37              if (!validatorResult.Success)
38                  return GenericResponseFactory.BuildBadRequestResponse<AddProductFromStockToOrderOutputModel>(validatorResult.Errors);
39
40              using (UnitOfWork.Start())
41              {
42                  #region Order
43                  var order = await OrdersPrimaryDatabaseRepository.GetOrderById(inputModel.OrderId);
44                  order.MoveProductFromStockToOrder(inputModel.StockId, inputModel.ProductStockLocationId, inputModel.ProductId);
45                  #endregion
46
47                  #region Product
48                  var product = await ProductsPrimaryDatabaseRepository.GetProductById(inputModel.ProductId);
49                  product.DecrementProductQuantityAtStockLocation(inputModel.StockId, inputModel.ProductStockLocationId, inputModel.ProductQuantity);
50                  #endregion
51
52                  #region StockEmployee
53                  var orderStockEmployee = order.GetOrderStockEmployee();
54                  var stockEmployee = await EmployeesPrimaryDatabaseRepository.GetEmployeeById(orderStockEmployee.EmployeeId);
55                  stockEmployee.IncrementEmployeePoints((int)StockEmployeePointsTypesEnum.AddProductFromStockToOrder);
56                  #endregion
57
58                  UnitOfWork.Commit();
59
60                  return GenericResponseFactory.BuildSuccessResponse<AddProductFromStockToOrderOutputModel>(new AddProductFromStockToOrderOutputModel());
61              }
62          }
63      }
64  }
```

Disadvantages of the first method :

Dirty Communications When There are alot of Single Transactions Require Multiple Aggegrations

The second method :
domain events communications

In our user story :

architecture with domain events

Event data is live on :

Event Handlers is live on :

```
namespace SupplyChain.Domain.DomainEvents.Orders
{
    1 reference
    public class DecrementProductQuantityAtStockLocationEventHandler : IDomainEventHandler<AddProductFromStockToOrderEvent>
    {
        private readonly IProductsPrimaryDatabaseRepository ProductsPrimaryDatabaseRepository;
        0 references
        public DecrementProductQuantityAtStockLocationEventHandler(
            IProductsPrimaryDatabaseRepository productsPrimaryDatabaseRepository)
        {
            ProductsPrimaryDatabaseRepository = productsPrimaryDatabaseRepository;
        }
        0 references
        public async Task Handle(AddProductFromStockToOrderEvent @event)
        {
            var product = await ProductsPrimaryDatabaseRepository.GetProductById(@event.ProductId);
            product.DecrementProductQuantityAtStockLocation(@event.StockId, @event.ProductStockLocationId, @event.ProductQuantity);
        }
    }
}
```

```
12  namespace SupplyChain.Domain.DomainEvents.Orders
13  {
        1 reference
14      public class IncrementStockEmployeePointsEventHandler : IDomainEventHandler<AddProductFromStockToOrderEvent>
15      {
16          private readonly IEmployeesPrimaryDatabaseRepository EmployeesPrimaryDatabaseRepository;
            0 references
17          public IncrementStockEmployeePointsEventHandler(
18              IEmployeesPrimaryDatabaseRepository employeesPrimaryDatabaseRepository)
19          {
20              EmployeesPrimaryDatabaseRepository = employeesPrimaryDatabaseRepository;
21          }
            0 references
22          public async Task Handle(AddProductFromStockToOrderEvent @event)
23          {
24              var orderStockEmployee = @event.Order.GetOrderStockEmployee();
25              var stockEmployee = await EmployeesPrimaryDatabaseRepository.GetEmployeeById(orderStockEmployee.EmployeeId);
26              stockEmployee.IncrementEmployeePoints((int)StockEmployeePointsTypesEnum.AddProductFromStockToOrder);
27          }
28      }
29  }
```

Domain Event Bus Can be inside The Application Service :

```
namespace SupplyChain.Domain.ApplicationServices.OrdersManagementModule
{
    1 reference
    public class AddProductFromStockToOrderApplicationService : IAddProductFromStockToOrderApplicationService
    {
        private readonly AddProductFromStockToOrderValidator Validator;
        private readonly IOrdersPrimaryDatabaseRepository OrdersPrimaryDatabaseRepository;
        private readonly IUnitOfWork UnitOfWork;
        0 references
        public AddProductFromStockToOrderApplicationService(
            AddProductFromStockToOrderValidator addProductFromStockToOrderValidator,
            IOrdersPrimaryDatabaseRepository ordersPrimaryDatabaseRepository,
            IUnitOfWork unitOfWork)
        {
            Validator = addProductFromStockToOrderValidator;
            OrdersPrimaryDatabaseRepository = ordersPrimaryDatabaseRepository;
            UnitOfWork = unitOfWork;
        }
```

```
        1 reference
        public async Task<GenericResponse<AddProductFromStockToOrderOutputModel>> Execute(AddProductFromStockToOrderInputModel inputModel)
        {
            var validatorResult = Validator.Check(inputModel);

            if (!validatorResult.Success)
                return GenericResponseFactory.BuildBadRequestResponse<AddProductFromStockToOrderOutputModel>(validatorResult.Errors);

            using (UnitOfWork.Start())
            {

                var order = await OrdersPrimaryDatabaseRepository.GetOrderById(inputModel.OrderId);
                order.MoveProductFromStockToOrder(inputModel.StockId, inputModel.ProductStockLocationId, inputModel.ProductId);

                DomainEventsBus.Raise(
                    new AddProductFromStockToOrderEvent
                    {
                        Order = order,
                        ProductId = inputModel.ProductId,
                        StockId = inputModel.StockId,
                        ProductStockLocationId = inputModel.ProductStockLocationId,
                        ProductQuantity = inputModel.ProductQuantity
                    }
                );

                UnitOfWork.Commit();

                return GenericResponseFactory.BuildSuccessResponse<AddProductFromStockToOrderOutputModel>(new AddProductFromStockToOrderOutputModel());
            }
        }
    }
}
```

But it more better to locate Domain Event inside The Aggregations , there are a lot of pattern and libraries that help to do this :

```
1 reference
public async Task<GenericResponse<AddProductFromStockToOrderOutputModel>> Execute(AddProductFromStockToOrderInputModel inputModel)
{
    var validatorResult = Validator.Check(inputModel);

    if (!validatorResult.Success)
        return GenericResponseFactory.BuildBadRequestResponse<AddProductFromStockToOrderOutputModel>(validatorResult.Errors);

    using (UnitOfWork.Start())
    {

        var order = await OrdersPrimaryDatabaseRepository.GetOrderById(inputModel.OrderId);
        order.MoveProductFromStockToOrder(inputModel.StockId, inputModel.ProductStockLocationId, inputModel.ProductId, inputModel.ProductQuantity);

        UnitOfWork.Commit();

        return GenericResponseFactory.BuildSuccessResponse<AddProductFromStockToOrderOutputModel>(new AddProductFromStockToOrderOutputModel());
    }
}
```

Product Aggregation Root :

```
1 reference
public void MoveProductFromStockToOrder(long stockId, long locationId, long productId , decimal productQuantity)
{
    LogicOfTheMethodWillBeFirstHereonThisRegion

    #region EventWillBeTriggeredAtEnd
    DomainEvents.Raise(
            new AddProductFromStockToOrderEvent
            {
                Order = this,
                ProductId = productId,
                StockId = stockId,
                ProductStockLocationId = locationId,
                ProductQuantity = productQuantity
            }
        );
    #endregion
}
```

advantages of the second method :

Using domain events can help to build decoupled system components and ensure the consistency for the business rules when different aggregations effect on each other.

when action 1 is happened on aggregation 1 :

action 2 must be happened on aggregation 2

action 3 must be happened on aggregation 3

action n must be happened on aggregation n

specially when action 1 can called from different application services.

for example :

our system allow client , cashier and admin to create order , every role has its own application service and send different data to create order, but the business rules and business workflow is the same.

When order is created , it effect on the client aggregation and product aggregation , so instead of duplicate the same code on every application service as we see on the first method, encapsulate the business logic of other aggregations inside event handlers , and when Order.Create() method is called from any application service , the domain events bus will automatically call the event handlers of this event.

Single Event With Multiple Application Services

Also domain events help us to make our code more flexible and reusable , for example when new business rule is added to create order operation that effect on Employee aggregation and instead of looping on all application services that create orders and apply this change , by using domain events we can add new event handler that apply this business rule and it automatically called every time Order.Create() is called on any application service.

New Event Handler Will Automatically Triggered

Bounded contexts and database mapping :

as we note there are some unique aggregations that exist only in single bounded context like :

- Supplier Company (Supply Chain)
- Supplier Employee (Supply Chain)
- Supplier Invoice (Supply Chain)
- Shipping Company (Shipping)
- Shipping Employee (Shipping)

also there are some shared aggregations that exist in more than bounded contexts like :

- Product
- Order
- Client

Let’s take shared aggregation like Client for example in(Marketing – SupplyChain – Shipping) :

- There are shared attributes like :
    1. Name Attribute : exist on Three Different bounded context (Marketing – Supply Chain – Shipping)
    2. PhoneNumber Attribute : exist on two

Different bounded context (Marketing – Shipping)

- There are shared entities like

  1. ClientAddress entity : exist on two Different bounded context (Marketing – Shipping)

```
namespace Marketing.Domain.Core.Clients
{
    5 references
    public class Client : AggregationRootFullAudit<long>
    {
        1 reference
        public string Name { get; private set; }
        1 reference
        public DateTime BirthDate { get; private set; }
        0 references
        public string PhoneNumber { get; private set; }
        0 references
        public long TotalProductsLikes { get; private set; }
        0 references
        public long TotalProductsViews { get; private set; }
        0 references
        public bool IsActive { get; private set; }

        private readonly List<ClientAddress> _clientAddresses = new List<ClientAddress>();
        0 references
        public IReadOnlyCollection<ClientAddress> ClientAddresses => _clientAddresses.AsReadOnly();

        private readonly List<ClientNationality> _clientNationalities = new List<ClientNationality>();
        0 references
        public IReadOnlyCollection<ClientNationality> ClientNationalities => _clientNationalities.AsReadOnly();

        Operations
    }
}
```

```
9   namespace Shipping.Domain.Core.Clients
10  {
        0 references
11      public class Client : AggregationRootFullAudit<long>
12      {
            0 references
13          public string Name { get; private set; }
            0 references
14          public string PhoneNumber { get; private set; }
15
16          private readonly List<ClientAddress> _clientAddresses = new List<ClientAddress>();
            0 references
17          public IReadOnlyCollection<ClientAddress> ClientAddresses => _clientAddresses.AsReadOnly();
18
19          operations
21
22      }
23  }
```

```
8   namespace SupplyChain.Domain.Core.Clients
9   {
        0 references
10      public class Client : AggregationRootFullAudit<long>
11      {
            0 references
12          public string Name { get; private set; }
            0 references
13          public long TotalCurrentInProgressOrdersAtStocks { get; private set; }
            0 references
14          public bool IsActive { get; private set; }
15
16          Operations
18
19      }
20  }
```

A) Same database same schema :

on this method , there are single database and single schema that all bounded contexts will be mapped to it , this mean all shared aggregations , attributes and entities will be mapped to the same physical database unit (table).

Single scheme example on SQL Server :

For shared aggregation roots that exist on multiple bounded contexts like Client , there will be **single table on the database** that contain all attributes for all bounded contexts and every aggregation root on different bounded context will mapped only to its attributes.

```
namespace SupplyChain.Domain.Core.Clients
{
    0 references
    public class Client : AggregationRootFullAudit<long>
    {
        0 references
        public string Name { get; private set; }
        0 references
        public long TotalCurrentInProgressOrdersAtStocks { get; private set; }
        0 references
        public bool IsActive { get; private set; }

        Operations

    }
}
```

Clients Database Table

Column Name	Data Type	Allow Nulls
Id	bigint	☐
Name	nvarchar(255)	☐
BirthDate	datetime	☑
MobileNumber	nvarchar(255)	☑
TotalCurrentInProgressOrdersAtStocks	int	☑
TotalProductsLikes	int	☑
TotalProductsViews	int	☑
IsActive	bit	☑
IsDeleted	bit	☑
		☐

Frame 1

```
namespace Marketing.Domain.Core.Clients
{
    5 references
    public class Client : AggregationRootFullAudit<long>
    {
        1 reference
        public string Name { get; private set; }
        1 reference
        public DateTime BirthDate { get; private set; }
        0 references
        public string PhoneNumber { get; private set; }
        0 references
        public long TotalProductsLikes { get; private set; }
        0 references
        public long TotalProductsViews { get; private set; }
        0 references
        public bool IsActive { get; private set; }

        private readonly List<ClientAddress> _clientAddresses = new List<ClientAddress>();
        0 references
        public IReadOnlyCollection<ClientAddress> ClientAddresses => _clientAddresses.AsReadOnly();

        private readonly List<ClientNationality> _clientNationalities = new List<ClientNationality>();
        0 references
        public IReadOnlyCollection<ClientNationality> ClientNationalities => _clientNationalities.AsReadOnly();

        Operations
    }
}
```

Clients Database Table

Column Name	Data Type	Allow Nulls
Id	bigint	☐
Name	nvarchar(255)	☐
BirthDate	datetime	☑
MobileNumber	nvarchar(255)	☑
TotalCurrentInProgressOrdersAtStocks	int	☑
TotalProductsLikes	int	☑
TotalProductsViews	int	☑
IsActive	bit	☑
IsDeleted	bit	☑
		☐

- Advantages of Single Database Single Schema :

1. Very simple.

2.No data duplication.

3. Strong consistency between shared aggregations ,

Insert – Update – Delete Row from bounded context will be reflected directly to other bounded contexts.

- Disadvantages of Single Database Single Schema :

1. shared attributes can cause some conflicts between different bounded contexts :

For ex : Marketing team want to divide client name to first name , middle name and last name but other bounded contexts that share also client name like supply chain and shipping want to work with client name as single unit.

So changing in bounded context on shared attribute can effect other bounded context.

```
8   namespace Marketing.Domain.Core.Clients
9   {
        5 references
10      public class Client : AggregationRootFullAudit<long>
11      {
            0 references
12          public string FirstName { get; private set; }
            0 references
13          public string MiddleName { get; private set; }
            0 references
14          public string LastName { get; private set; }
            1 reference
```

```
8   namespace SupplyChain.Domain.Core.Clients
9   {
        0 references
10      public class Client : AggregationRootFullAudit<long>
11      {
            0 references
12          public string Name { get; private set; }
```

```
9   namespace Shipping.Domain.Core.Clients
10  {
        0 references
11      public class Client : AggregationRootFullAudit<long>
12      {
            0 references
13          public string Name { get; private set; }
```

2. Concurrent transactions for shared aggregation in different bounded contexts can slow the performance :

Although client update different data that required by stock employee , the operation by stock employee to read order and order items details will blocked until the update operation by client is executed.

you can solve this problem by choosing the appropriate isolation level for sql databases or choosing a good no sql database that can handle this problem.

B) Same database different schemes :

In this method , every bounded context is mapped to a sperate schema on the database.

Single database different schemes example on SQL Server :

- ProductCatalog.Client
- ProductCatalog.Product
- ProductCatalog.ProductAttribute
- ProductCatalog.ProductFile
- ProductCatalog.ProductFileType

- SupplyChain.Client
- SupplyChain.Employee
- SupplyChain.Order
- SupplyChain.OrderEmployee
- SupplyChain.OrderItem
- SupplyChain.Product
- SupplyChain.ProductBarcode
- SupplyChain.ProductPrice
- SupplyChain.ProductStock
- SupplyChain.ProductStockLocation
- SupplyChain.ProductStockType
- SupplyChain.ProductTransaction
- SupplyChain.ProductType
- SupplyChain.Stock
- SupplyChain.StockLocation
- SupplyChain.SupplierCompany
- SupplyChain.SupplierEmployee
- SupplyChain.SupplierInvoice
- SupplyChain.SupplierInvoiceItem

- ClientPreferences.Client
- ClientPreferences.Product
- ClientPreferences.ProductLike
- ClientPreferences.ProductView

- Marketing.Client
- Marketing.ClientAddress
- Marketing.ClientNationality
- Marketing.Employee
- Marketing.Order
- Marketing.OrderAddress
- Marketing.OrderEmployee
- Marketing.OrderItem
- Marketing.Product
- Marketing.ProductLike
- Marketing.ProductPrice
- Marketing.ProductReport
- Marketing.ProductView

- Ordering.Client
- Ordering.Order
- Ordering.OrderAddress
- Ordering.OrderEmployee
- Ordering.OrderItem
- Ordering.Product

- Shipping.Client
- Shipping.ClientAddress
- Shipping.Employee
- Shipping.Order
- Shipping.OrderAddress
- Shipping.OrderItem
- Shipping.Product
- Shipping.ShippingCompany
- Shipping.ShippingEmployee

let’s take example of shared aggregation root like client

Marketing.Client Table

Id	Name	BirthDate	PhoneNumber	TotalProductsLikes	TotalProductsViews	IsActive
1	Sara	12-1-2002	01234556	4	19	true
2	john	3-6-1990	04356789	3	15	true
3	ahmed	5-1-1993	03123987	2	20	true

SupplyChain.Client Table

Id	Name	TotalCurrentInProgressOrdersAtStocks	IsActive
1	Sara	3	True
2	john	2	True
3	ahmed	4	True

- there will be a separate table for client on every bounded context contain it.
- Every table will contain only the attributes required by it’s bounded context.
- Shared attributes like name , isActive need to be consistency with each other , when shared attribute is updated in one of the bounded context , the new value of this attribute must be published to other bounded contexts that share it.
- Usually there are single bounded context that responsible about creating the aggregation and then publish it to other bounded contexts that share it usually through event bus or message queue.

Example of creating aggregation in bounded context and then published it to all other bounded contexts that share the whole aggregation :

Example of update aggregation in bounded context and then publish new updated value to other bounded context that only share this updated attribute :

Order delivery reception date is exist only on Ordering.Order and Shipping.Order , so when client update this attribute on ordering bounded context , the new updated value must be published also to shipping bounded context only but order in other bounded contexts like marketing , supply chain dose not contain this attribute.

- Advantages of Single Database Multiple Schemes :

- every bounded context data is isolated separately, so changing on the shared attributes on one of the bounded context does not cause any conflicts to other bounded contexts.

Marketing.Client

Id	FirstName	MiddleName	LastName	...
1	sara	luis	salama	...
2	ahmed	mohamed	elshrief	.....

SupplyChain.Client

Id	Name	...
1	sara luis salama	...
2	ahmed mohamed elshrief	.....

- concurrent transactions on shared aggregations on different bounded context does not block each other because every transaction is mapped to different table.

Disadvantages of Single Database Multiple Schemes

- Data duplication
- Complexity and cost of the atomicity, consistency and handling failure :

Depending on the business rules , developers have to choose between strong consistency or eventual consistency for shared aggregations.

Strong consistency for shared aggregation means the transaction must be committed and successful response return to client only after it successfully committed on all bounded contexts that share it.

The strong consistency has a lot of advantages , it achieve the atomicity, but the cost of this is the performance , client need to wait until the new order is published to all bounded contexts that share it.

Eventual consistency for shared aggregation mean the successful response is returned directly to the client after the transaction only just committed on the current bounded context , and then in asynchronous or background publish the transaction to other bounded contexts.

With eventual consistency the system can has more throughput , the performance increased compared to strong consistency , but eventual consistency do this with cost for other thing (handling failures) , in the happy scenario there will be no problem , the problem start when the transaction is committed successfully on some bounded context and failure on other , which make the system components inconsistent with each other , in our example the client create new order in ordering bounded context but when the transaction is failed on supply chain bounded context , the stock employee can not know any thing about the new

created order.

At the end every type of consistency has advantages and also cost for other thing , the developers , architecture and business team should choose the most suitable solution that serve the business of their system.

If they choose the strong consistency , they must think about solutions for the performance issues , and how to increase the throughput of the system and make it has low latency.

If they choose the eventual consistency , they must think about solutions for handling partial failures , and how to fix the inconsistent data.

Different bounded context different database :

On this method , every bounded context is mapped to different database , the databases can be of the same type or every bounded context can choose the most suitable database for it’s features and business needs.

- Advantages of database per bounded context :

- high throughput

For example Instead of having single database that can serve 1000 request per second , the system now have 6 database and they can serve 6000 request per second.

- Flexibility and high performance

The type of operations that executed on every bounded context can be different than other bounded contexts for example :

- Most of the operations that executed in the product catalog bounded context are read operations : searching about products , view product details .
  So if you have a huge volume of data with complex model , may be it better to use NoSQL document database like elastic search or mongoDB.

- Most of the operations that executed in the supply chain bounded context are write operations that require strong consistency , so Relational or SQL database like SQL Server or MySQL can be the most suitable database for this bounded context.

Disadvantages of database per bounded context :

Distributed databases have a lot of advantages like Scalability , performance , data model flexibility and a lot of other advantages , but on other side there will cost for that , and below some of famous challenges that need smart solutions on distributed data systems :

- Distributed transaction complexity.
- How to handle partial failures , transaction is committed in some database and failure on other.
- Strong consistency performance .
- Concurrency control model.