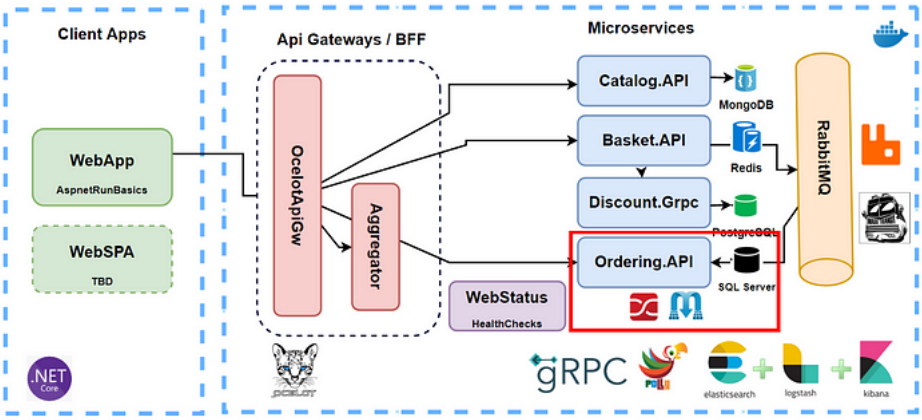
CQRS and Event Sourcing in Event Driven Architecture of Ordering Microservices

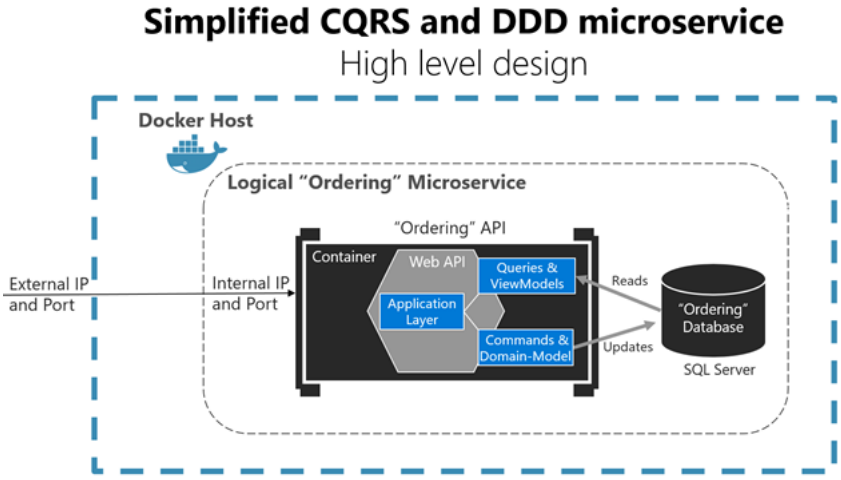
Building Ordering Microservices with Clean Architecture and CQRS Implementation with using MediatR, FluentValidation and AutoMapper.



Introduction

In this article we will show how to perform Ordering microservices operations on ASP.NET Core Web API using **Entity Framework Core with Sql Server** Database applying **Clean Architecture** and **CQRS**.

By the end of the article, we will have a Web API which implemented CRUD operations over Order entity with implementing **CQRS design pattern using MediatR, MediatR**, **FluentValidation** and **AutoMapper** packages.



Developing **Ordering microservice** which includes;

- **ASP.NET Core Web API** application
- **REST** API principles, CRUD operations
- **Entity Framework Core Code-First** Approach
- Implementing **DDD, CQRS,** and **Clean Architecture** with using Best Practices applying **SOLID principles**
- Developing **CQRS implementation** on commands and queries with using **MediatR**, **FluentValidation** and **AutoMapper** packages
- **SqlServer database** connection and containerization
- Using **Entity Framework Core** ORM and auto migrate to SqlServer when application startup
- Consuming RabbitMQ BasketCheckout event queue with using MassTransit-RabbitMQ Configuration

We will Analysis and Architecting of Ordering Microservices, applying Clean architecture and CQRS Design Pattern. **Containerize** Ordering Microservices with SqlServer database using **Docker Compose**.

So in this section, we are going to Develop Ordering.API Microservices with SqlServer.

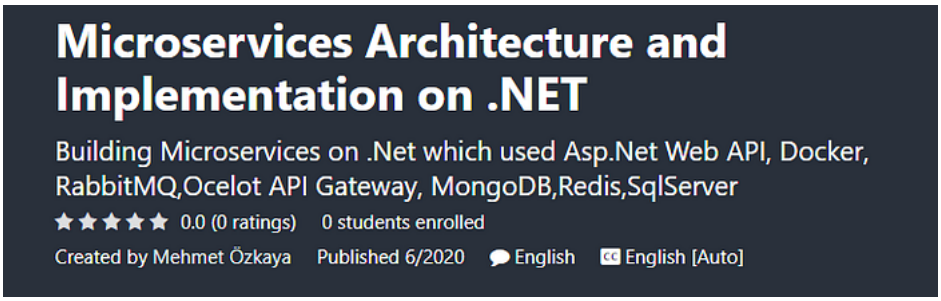
Background

You can follow the previous article which explains overall microservice architecture of this example.

Check for the previous article which explained overall microservice architecture of this repository.

We will focus on Ordering microservice from that overall e-commerce microservice architecture.

Step by Step Development w/ Udemy Course



Get Udemy Course with discounted — Microservices Architecture and Implementation on .NET.

Source Code

Get the Source Code from AspnetRun Microservices Github — Clone or fork this repository, if you like don't forget the star. If you find or ask anything you can directly open issue on repository.

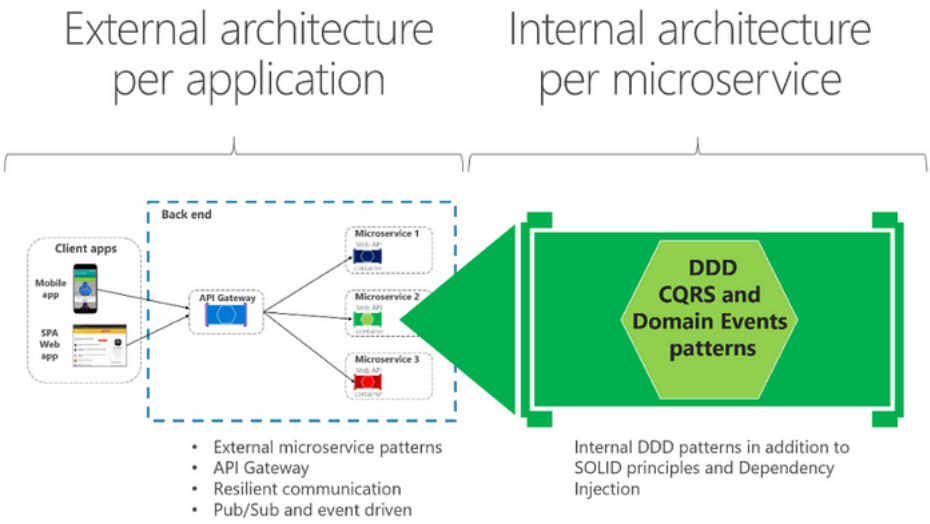
Prerequisites

- Install the .NET Core 5 or above SDK
- Install Visual Studio 2019 v16.x or above
- Docker Desktop

Ordering.API Microservices

Ordering microservice use **ASP.NET Core Web API reference application** with **Entity Framework Core**, demonstrating a layered application architecture with DDD best practices. Implements N-Layer **Hexagonal architecture** (Core, Application, Infrastructure and Presentation Layers) and **Domain Driven Design** (Entities, Repositories, Domain/Application Services, DTO's…) and aimed to be a **Clean Architecture**, with applying **SOLID principles** in order to use for a project template.

Also implements **CQRS Design Pattern** into their layers in order to separate **Queries** and **Command** for Ordering microservice.



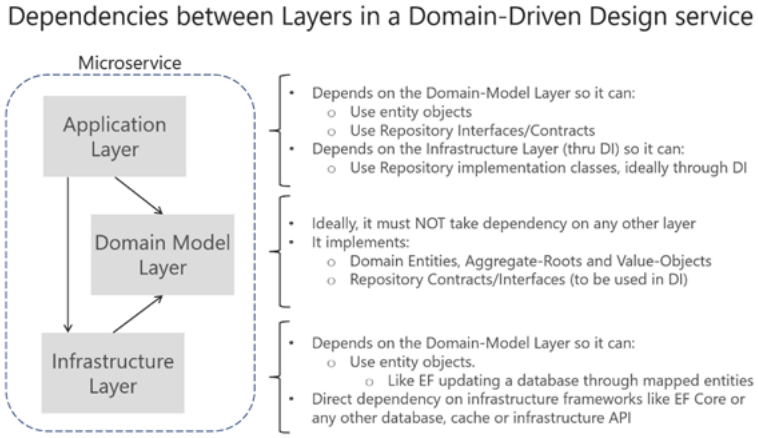
We will be following **best practices** like **loosely coupled, dependency-inverted** architecture and using **design patterns** such as **Dependency Injection**, logging, validation, exception handling, localization and so on.

First, we are going to start with remembering the **core concepts, principles of layered architecture design**. We are going to start with **small principles** and continue to **design patterns** which all these items are used our ASP.NET Core Layered Architecture project.

Before start with the architecture, we should know the **Design Principles** and some of the **Design Patterns** over the clean architecture.

Domain Driven Design (DDD)

Domain Driven Design (DDD) is not an improved technology or specific method. Domain Driven Design (DDD) is an approach that tries to bring solutions to the basic problems frequently experienced in the development of **complex software systems** and in ensuring the continuity of our applications after the implementation of these opposing projects. To understand Domain Driven Design (DDD), some basic concepts need to be mastered. With these concepts, we can enter the Domain Driven Design (DDD).



Ubiquitous Language

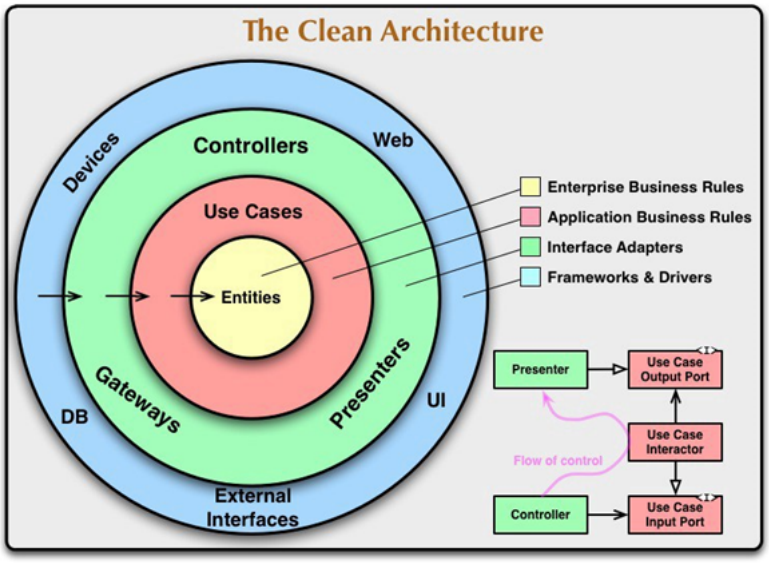
It is one of the **cornerstones** of Domain Driven Design (DDD). We need to be able to produce the desired output of the software developers and to ensure the continuity of this output to be able to **speak the same language** as the **Domain Expert.** Afterwards, we must transfer this experience to the methods and classes that we use while developing our applications by giving the names of the concepts used by experts. Every service we will use in our project must have a response in the domain. Thus, everyone involved in the project can **speak this common language** and understand each other.

Bounded Context

A recommended approach to use in Domain Driven Design (DDD) complex systems. The complex may **contain sub domains** within a domain. It should also include Domain Driven Design (DDD). Example of e-commerce site; · Order management · Customer management · Stock management · Delivery management · Payment System management · Product management · User management may contain many sub domains. As these **sub domains are grouped, Bounded Context** refers to structures in which the group of individuals most logically associated with each other in terms of the rules of the **Aggregate Roots** are **grouped together** and the **responsibilities** of this group are clearly defined.

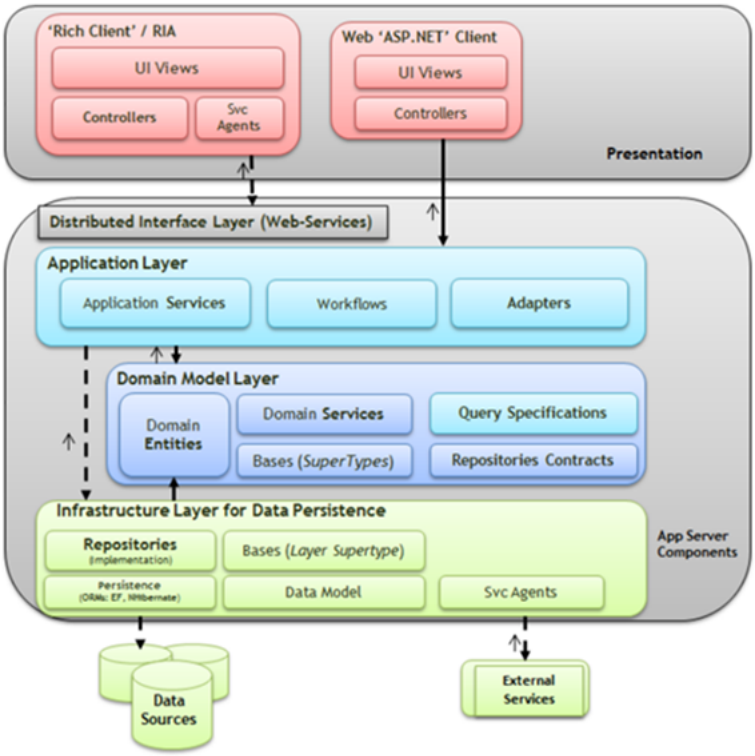
Clean Architecture (aka Ports and Adaptors)

Hexagonal Architecture (aka Ports and Adapters) is one strategy to decouple the use cases from the external details. It was coined by Alistar Cockburn more than 13 years ago, and this received improvements with the Onion and Clean Architectures.



Ordering Microservice, implements **N-Layer Hexagonal architecture (Core, Application, Infrastructure and Presentation Layers)** and **Domain Driven Design (Entities, Repositories, Domain/Application Services, DTO's…).** Also implements and provides a good infrastructure to implement best practices such as Dependency Injection, logging, validation, exception handling, localization and so on.

Aimed to be a Clean Architecture also called **Onion Architecture**, with applying **SOLID principles** in order to use for a project template. The below image represents approach of development architecture of run repository series;



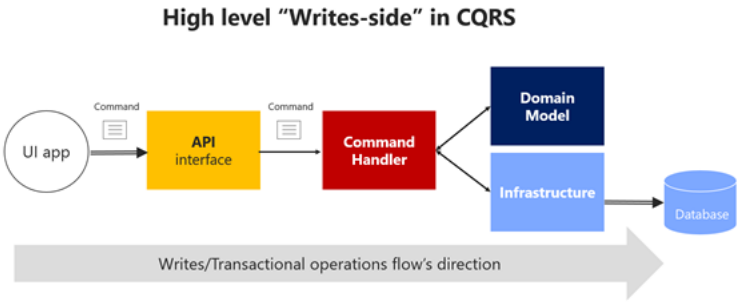
According to this diagram, we applied these layers and detail components in our project.

CQRS (Command Query Responsibility Segregation) Design Pattern

CQRS means **separation of command** and **query responsibilities**. Although it has become an increasingly popular pattern in recent years, it has increased in popularity after this article written by Martin Fowler when he saw this pattern by Greg Young.

We can say that it is based on the **CQS (Command Query Separation)** principle for the CQRS architecture. The main idea of CQS is to separate the interfaces between our operations that read the data and the operations that update the data. In CQRS, this is added to the separation of our business models.

CQRS is a software development pattern based on conducting reading and writing / updating processes on different models. The data you read and the data you write are stored in **different database tools**.



Command — First of all, command type is the only way to change something in the system. If there is no command, the state of the system remains unchanged. Command types should not return any value. Command types often act as plain objects that are sent to **CommandHandler** as parameters when used with **CommandHandler** and **CommandDispatcher** types.

Query — The only way to do similar reading is the type of Query. They cannot change the state of the system. They are generally used with **QueryHandler** and **QueryDispatcher** types.

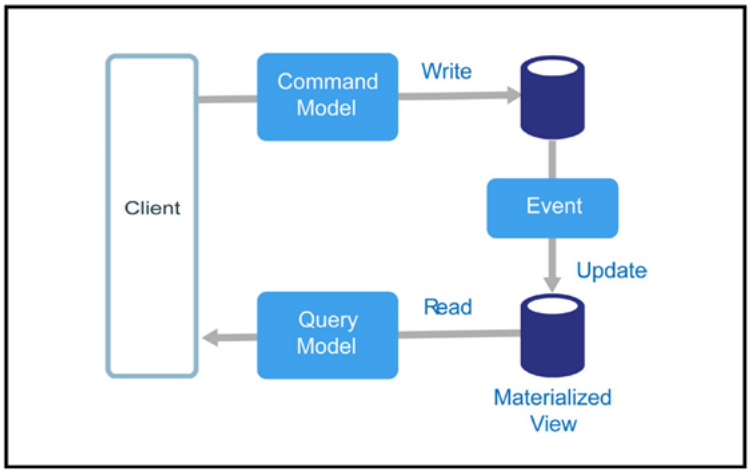
After doing Command and Query parsing in our system, it will be clear that the domain model we use for data writing is not suitable for reading data.

Eventual Consistent

All models in consistent systems are stored consistently without interruption. In **Eventual consistent** systems, the models **may be inconsistent for a while** as a result of writing / updating processes. This situation is finally resolved, and the system will eventually be consistent.

In systems developed using the **CQRS software pattern**, the write / update requests from the client are transmitted to the existing services on the system with the write model. If the request is processed, the necessary changes are made to the **reading model**. When the user queries data, the services answer the requests with the reading model.

CQRS is the updating of the reading model with **asynchronous processes** after the writing model is registered. This method is the reason why the term **Eventual Consistency** appeared. Generally, systems developed with CQRS design are eventual consistent systems.



Since there is no **transactional dependency** in eventual consistent systems, reading and writing actions do not wait for each other. For this reason, CQRS systems performance is important.

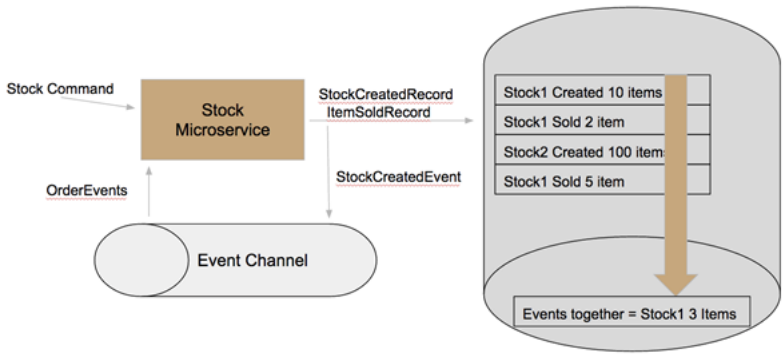
Eventual Sourcing

Event Sourcing is a method shaped on the main idea of **accumulating events** that took place in our system. Objects that are one of the main parts of the system that have an identity are called entities.

In systems developed with **Event Sourcing**, the latest status of the **assets are not recorded**. **Instead, events affecting the state of the assets are recorded.** When the query is sent by the client and the final status of the asset is requested, the system combines the existing Event information and provides the client with the necessary information.

Events occurring in order will constitute the result **Entity** to us. In other words, if an entity is expressed as the sum of the Events that will create this line, not as a data line, it can offer us an Eventual Consistent structure.

And if our system consisting of many Resource is built with **Event Sourcing** in this way, we can perform Eventual Consistent operations through these systems. As an example, we can express the transaction in a stock service as in the picture.



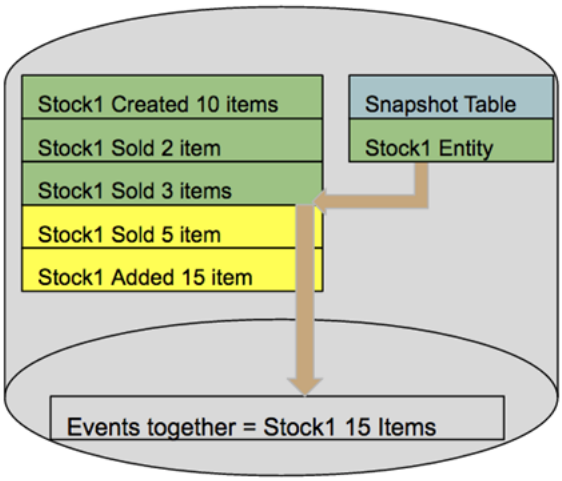
In the Event Sourcing structure, when the client requests the data, the breakdown of the events related to the asset is created. The state of the data is created from this transcript. It is obvious that the process of **generating data from the data will more slower** results compared to the traditional method where the data is kept ready.

We can solve the last two problems that occur when using Event Sourcing method (re-creating the asset every time and filtering the asset according to its areas) together with CQRS. For this reason, **Event Sourcing and CQRS** are frequently mentioned together.

CQRS and Event Sourcing

There are reading and writing models on the CQRS Design Pattern. In the Event Sourcing method, event information which affects the state of the asset, is not stored.

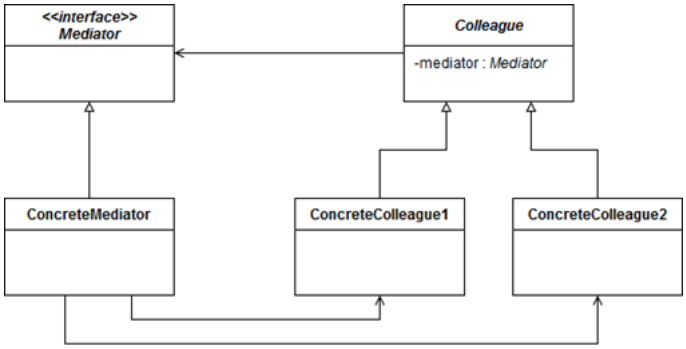
Consider event writing as a model and the final state of our existence as a reading model. The client starts an operation to change the state of the asset on the system. Event information **generated as a result** of the operation is stored in our system as a **model for writing**. In this case, our process is triggered and our **reading model** is updated depending on our writing model.



As a result, we had the Event Sourcing structure provide to the writing model in our system. It is possible for us to reach the state of our system at any time we want. On the other hand, we solve our problem by using the reading model for query operations and performance problem.

Mediator Design Pattern

As can be understood from the name of **Mediator**, it is a class created with the logic of performing this operation by using an **intermediate class** to perform the connection and other operations between the classes derived from the same interface.



One of the most frequently given examples is the tower structure, which gives the permission of the aircraft at the airports. Airplanes do not get permission from each other. All airplanes only get permission from the tower and the operations take place accordingly. The vehicle here becomes the tower.

We will use this **Mediator Pattern** when using **MediatR** nuget packages in **Ordering.Application** project.

· **MediatR** — To implement Mediator Pattern

Code Structure

According to clean architecture and cqrs implementation, we applied these layers and detail components in our project. So, the result of the project structure occurred as below;

In order to applying Clean architecture and CQRS Design Pattern, we should create these 4 layer.

- Ordering.Domain Layer
- Ordering.Application Layer
- Ordering.API Layer
- Ordering.Infrastructure Layer

Ordering.Domain Layer and Ordering.Application Layer will be the Core
Ordering.API Layer and Ordering.Infrastructure Layer will be the Periphery layer of our reference application.

> ***NOTE:*** *In this article I am going to follow only CQRS implemented part of code which is Ordering.Application layer. You can follow the whole code from github repository.*

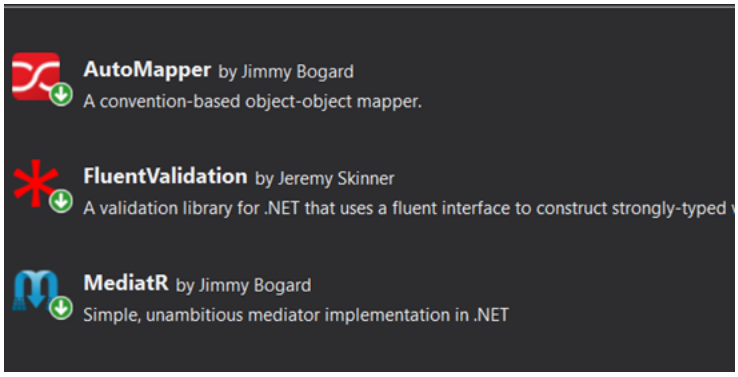
Library & Frameworks

For Basket microservices, we have to libraries in our Nuget Packages,

Ordering.Application

- MediatR.Extensions.Microsoft.DependencyInjection
- FluentValidation
- FluentValidation.DependencyInjectionExtensions
- AutoMapper
- AutoMapper.Extensions.Microsoft.Dependency
- Microsoft.Extensions.Logging.Abstractions

ProjectReference — Ordering.Domain



Developing Ordering.Application Layer with CQRS Pattern Implementation in Clean Architecture

This layer for Development of **Domain Logic** with **implementation**. Interfaces drives **business requirements** and **implementations** in this layer. Application Layer in order to implement our business logics, **use case operations**. The first point of implementation is definition of Model classes.

The use case of projects should be handled by Application layer. So, we are following to **CQRS Design pattern** when creating classes.

Application layer should be cover all business use cases with abstractions. So this layer should responsible for business use cases, business validations, flows and so on.

Now we can create main folders of Ordering.Application project.

Main Folders — Create Folders

Contracts
Features
Behaviours

Create all these 3 folders.
These are main objectives should handle in application.
Application Contracts
Application Features
Application Behaviours

Application Contracts
This folder should cover application capabilities. This should include Interfaces for abstracting use case implementations.

Application Features
This folder will apply CQRS design patterns for handling business use cases. We will create sub folder according to use case definitions like get orders by user name or checkout order and so on..

Application Behaviours
This folder will responsible for Application Behaviours that apply when performing use case implementations. In example validations, logging, cross-cutting concerns an so on.

Developing Ordering.Application Layer — Application Contracts

We are going to Develop Application Contracts of Ordering.Application Layer with CQRS Pattern Implementation in Clean Architecture.

Go to “Contracts” folder

After that examine “Contracts”

Create sub folders
“Persistence”
“Infrastructure”

— Create Persistence classes

“Persistence”
IAsyncRepository
IOrderRepository

```
IAsyncRepositorypublic interface IAsyncRepository<T> where T : EntityBase
 {
 Task<IReadOnlyList<T>> GetAllAsync();
 Task<IReadOnlyList<T>> GetAsync(Expression<Func<T, bool>> predicate);
 Task<IReadOnlyList<T>> GetAsync(Expression<Func<T, bool>> predicate = null,
 Func<IQueryable<T>, IOrderedQueryable<T>> orderBy = null,
 string includeString = null,
 bool disableTracking = true);
 Task<IReadOnlyList<T>> GetAsync(Expression<Func<T, bool>> predicate = null,
 Func<IQueryable<T>, IOrderedQueryable<T>> orderBy = null,
 List<Expression<Func<T, object>>> includes = null,
 bool disableTracking = true);
 Task<T> GetByIdAsync(int id);
 Task<T> AddAsync(T entity);
 Task UpdateAsync(T entity);
 Task DeleteAsync(T entity);
 }IOrderRepositorypublic interface IOrderRepository : IAsyncRepository<Order>
 {
 Task<IEnumerable<Order>> GetOrdersByUserName(string userName);
 }
```

With these repository classes we handled database related actions and abstracted with creating interfaces.
These interface will implement in Infrastructure layer with uisng ef.core and sql server.
But it doesnt important from Application layer, this interface could be handle any orm and databases not affecting application layer.
By this abstraction these infrasturcture changes could be made very easy.

Think about that, you can create 2 infrasturcutre layer 1 for ef.core-sqlserver other coukld be dapper-postgresql and you can change your infrastructure layer according to configuration.
By this way you can easly compare performance and decide to go your implementations.

After that we can develop other external infrasturture contracts.

“Infrastructure”
IEmailService

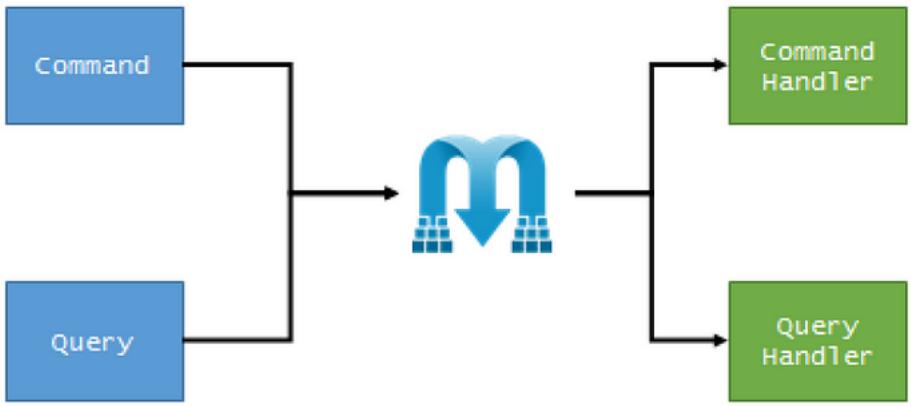
```
public interface IEmailService
 {
 Task<bool> SendEmail(Email email);
 }
```

This interface also will implement on Infrasturcture layer with any external mail sender package. We dont interested with that in Application. We focus on customer requirements.

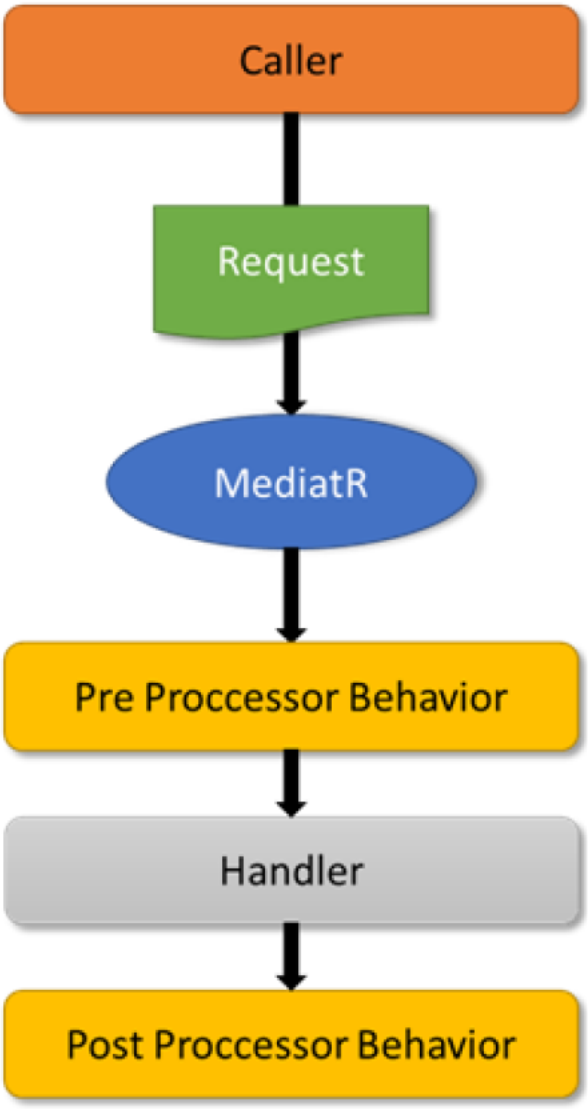
Customer wants from us to send mail when new order coming. So we will do it with this interface. After that these implementations will configure on presentation layer in aspnet built-in dependency injection.

MediatR Nuget Package

We will use this Mediator Pattern when using MediatR nuget packages in Ordering.Application project.
· MediatR — To implement Mediator Pattern



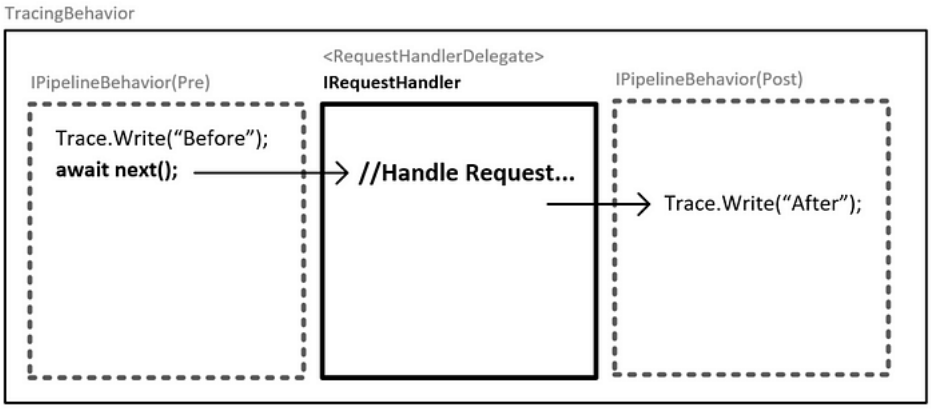
MediatR allows us to easily create and send Command and Query objects to the correct Command/Query Handlers.
You can see here, MediatR takes Command and Query objects and trigger to Handler classes.



Also you can find a MediatR request pipeline lifecycle for particular request.
As you can see that when Request accommodating in MediatR,
we can put some interceptors as a pre and post processor behaviors.

MediatR Pipeline Behaviour

You can see in this picture more clearly,



Here is **Tracing behavior** adding on **mediatr** as a **pipeline behaviour**.
Pre and Post request handles in Pipeline behaviours and perform tracing operations between request handler operations.

Also we will Apply the Validation pipeline behavior with using Fluent Validator nuget package.

High level “Writes-side” in CQRS

Here is an example of 1 Command Request to Ordering.Application layer.
A request comes from API call from Ordering.API and call the CommandHandler class in Ordering.Application with helping from MediatR nuget package.

In command handler class, it uses Infrastructure objects that perform database related operations.
In our case we will checkout the order and save order record.

As you can see that we have seen that CQRS Implementation with Mediator Design Pattern.

Developing Ordering.Application Layer — Application Features — GetOrdersListQuery

We are going to Develop Application Features of Ordering.Application Layer with CQRS Pattern Implementation in Clean Architecture.

We have seperated Commands and Queries, because we are going to apply CQRS design pattern.
This pattern basically explains that seperating read model and write model.
So we will obey this best practices on this Ordering Application layer.

Queries
Create folder
GetOrdersList

Add New Class into that folder;
GetOrdersListQuery.cs

```
 public class GetOrdersListQuery : IRequest<List<OrdersVm>>
 {
 public string UserName { get; set; }public GetOrdersListQuery(string userName)
 {
 UserName = userName ?? throw new ArgumentNullException(nameof(userName));
 }
 }
```

According to MediatR implementation, every IRequest implementation should have Handler classes. MediatR trigger this handler classes when Request comes.

GetOrdersListQueryHandler.cs

```
public class GetOrdersListQueryHandler : IRequestHandler<GetOrdersListQuery, List<OrdersVm>>
 {
 private readonly IOrderRepository _orderRepository;
 private readonly IMapper _mapper;public GetOrdersListQueryHandler(IOrderRepository orderRepository, IMapper mapper)
 {
 _orderRepository = orderRepository ?? throw new ArgumentNullException(nameof(orderRepository));
 _mapper = mapper ?? throw new ArgumentNullException(nameof(mapper));
 }public async Task<List<OrdersVm>> Handle(GetOrdersListQuery request, CancellationToken cancellationToken)
 {
 var orderList = await _orderRepository.GetOrdersByUserName(request.UserName);
 return _mapper.Map<List<OrdersVm>>(orderList);
 }
 }
```

This class needs to Dto object for query operation.
Its best practice to seperate your dto objects as per commands and queries.

By this way you can seperate your dependencies with original entities.

```
public class OrdersVm
 {
 public int Id { get; set; }
 public string UserName { get; set; }
 public decimal TotalPrice { get; set; }// BillingAddress
 public string FirstName { get; set; }
 public string LastName { get; set; }
 public string EmailAddress { get; set; }
 public string AddressLine { get; set; }
 public string Country { get; set; }
 public string State { get; set; }
 public string ZipCode { get; set; }// Payment
 public string CardName { get; set; }
 public string CardNumber { get; set; }
 public string Expiration { get; set; }
 public string CVV { get; set; }
 public int PaymentMethod { get; set; }
 }
```

Developing Ordering.Application Layer — Application Command Features — CheckoutOrder

We are going to Develop Application Command Features of Ordering.Application Layer with CQRS Pattern Implementation in Clean Architecture.

Go to “Orders” -> “Commands” folder

Create “CheckoutOrder” folder

Develop
CheckoutOrderCommand.cs
CheckoutOrderCommandHandler.cs

—

CheckoutOrderCommand.cs

```
public class CheckoutOrderCommand : IRequest<int>
{
public string UserName { get; set; }
public decimal TotalPrice { get; set; }// BillingAddress
public string FirstName { get; set; }
public string LastName { get; set; }
public string EmailAddress { get; set; }
public string AddressLine { get; set; }
public string Country { get; set; }
public string State { get; set; }
public string ZipCode { get; set; }// Payment
public string CardName { get; set; }
public string CardNumber { get; set; }
public string Expiration { get; set; }
public string CVV { get; set; }
public int PaymentMethod { get; set; }
}
```

— Now according to mediatR we should create handler class for this request and implement business logic. Our business logic is creating order and send mail.

CheckoutOrderCommandHandler.cs

```
public class CheckoutOrderCommandHandler : IRequestHandler<CheckoutOrderCommand, int>
{
private readonly IOrderRepository _orderRepository;
private readonly IMapper _mapper;
private readonly IEmailService _emailService;
private readonly ILogger<CheckoutOrderCommandHandler> _logger;public CheckoutOrderCommandHandler(IOrderRepository orderRepository, IMapper mapper, IE
{
_orderRepository = orderRepository ?? throw new ArgumentNullException(nameof(orderRepository));
_mapper = mapper ?? throw new ArgumentNullException(nameof(mapper));
_emailService = emailService ?? throw new ArgumentNullException(nameof(emailService));
_logger = logger ?? throw new ArgumentNullException(nameof(logger));
}public async Task<int> Handle(CheckoutOrderCommand request, CancellationToken cancellationToken)
{
var orderEntity = _mapper.Map<Order>(request);
var newOrder = await _orderRepository.AddAsync(orderEntity);

_logger.LogInformation($"Order {newOrder.Id} is successfully created.");

await SendMail(newOrder);return newOrder.Id;
}private async Task SendMail(Order order)
{
var email = new Email() { To = "ezozkme@gmail.com", Body = $"Order was created.", Subject = "Order was created" };try
{
await _emailService.SendEmail(email);
}
catch (Exception ex)
{
_logger.LogError($"Order {order.Id} failed due to an error with the mail service: {ex.Message}");
}
}
}
```

Lastly, I am going to add Validator for checkout order command.

CheckoutOrderCommandValidator.cs

```
public class CheckoutOrderCommandValidator : AbstractValidator<CheckoutOrderCommand>
{
public CheckoutOrderCommandValidator()
{
RuleFor(p => p.UserName)
.NotEmpty().WithMessage("{UserName} is required.")
.NotNull()
.MaximumLength(50).WithMessage("{UserName} must not exceed 50 characters.");RuleFor(p => p.EmailAddress)
.NotEmpty().WithMessage("{EmailAddress} is required.");RuleFor(p => p.TotalPrice)
.NotEmpty().WithMessage("{TotalPrice} is required.")
.GreaterThan(0).WithMessage("{TotalPrice} should be greater than zero.");
}
}
```

This validator will work when request comes to MediatR, before running handler class, this will run this validations.
For validation library, we are going to use FluentValidation. Thats why we have inherited AbstractValidator class.
We have used RuleFor function which comes from FluentValidation and provide to define some set of validations. We mostly using empty check and can able to give messages when validation not passed.

Developing Ordering.Application Layer — Application Command Features — UpdateOrder

We are going to Develop Update Application Command Features of Ordering.Application Layer with CQRS Pattern Implementation in Clean Architecture.

Create "UpdateOrder" folder

Develop
UpdateOrderCommand
UpdateOrderCommandHandler.cs

— **UpdateOrderCommand.cs**

```
public class UpdateOrderCommand : IRequest
{
public int Id { get; set; }
public string UserName { get; set; }
public decimal TotalPrice { get; set; }// BillingAddress
public string FirstName { get; set; }
public string LastName { get; set; }
public string EmailAddress { get; set; }
public string AddressLine { get; set; }
public string Country { get; set; }
public string State { get; set; }
public string ZipCode { get; set; }// Payment
public string CardName { get; set; }
public string CardNumber { get; set; }
public string Expiration { get; set; }
public string CVV { get; set; }
public int PaymentMethod { get; set; }
}
```

— Now according to mediatR we should create handler class for this request and implement business logic. Our business logic is updating order.

UpdateOrderCommandHandler.cs

```
public class UpdateOrderCommandHandler : IRequestHandler<UpdateOrderCommand>
{
private readonly IOrderRepository _orderRepository;
private readonly IMapper _mapper;
private readonly ILogger<UpdateOrderCommandHandler> _logger;public UpdateOrderCommandHandler(IOrderRepository orderRepository, IMapper mapper, ILogge
{
_orderRepository = orderRepository ?? throw new ArgumentNullException(nameof(orderRepository));
_mapper = mapper ?? throw new ArgumentNullException(nameof(mapper));
_logger = logger ?? throw new ArgumentNullException(nameof(logger));
}public async Task<Unit> Handle(UpdateOrderCommand request, CancellationToken cancellationToken)
{
var orderToUpdate = await _orderRepository.GetByIdAsync(request.Id);
if (orderToUpdate == null)
{
throw new NotFoundException(nameof(Order), request.Id);
}

_mapper.Map(request, orderToUpdate, typeof(UpdateOrderCommand), typeof(Order));await _orderRepository.UpdateAsync(orderToUpdate);_logger.LogInformati
}
}
```

In UpdateOrderCommandHandler class, we have updating order record with repository object.

Lastly, I am going to add Validator for update order command.

UpdateOrderCommandValidator.cs

```
public class UpdateOrderCommandValidator : AbstractValidator<UpdateOrderCommand>
{
public UpdateOrderCommandValidator()
{
RuleFor(p => p.UserName)
.NotEmpty().WithMessage("{UserName} is required.")
.NotNull()
.MaximumLength(50).WithMessage("{UserName} must not exceed 50 characters.");RuleFor(p => p.EmailAddress)
.NotEmpty().WithMessage("{EmailAddress} is required.");RuleFor(p => p.TotalPrice)
.NotEmpty().WithMessage("{TotalPrice} is required.")
.GreaterThan(0).WithMessage("{TotalPrice} should be greater than zero.");
}
}
```

This validator will work when request comes to MediatR, before running handler class, this will run this validations.
For validation library, we are going to use FluentValidation. Thats why we have inherited AbstractValidator class.

As you can see that we have Developed CheckoutOrderCommand and UpdateOrderCommand Application Features of Ordering.Application Layer in Clean Architecture. As the same way, you can develop to DeleteOrderCommand objects.

Developing Ordering.Application Layer — Application Behaviours

We are going to Develop Application Behaviours of Ordering.Application Layer with CQRS Pattern Implementation in Clean Architecture.

As you remember that we had 3 main folders in Application layer.

+ Application Contracts
+ Application Features
— Application Behaviours

Now we have focus on Application Behaviours.

— We can add Application Behaviours by using MediatR.
— MediatR provides us to IPipelineBehavior interface that we can intercept requests and perform any behavior before executing handle classes.

— Create ValidationBehaviour class

ValidationBehaviour.cs

```
using ValidationException = Ordering.Application.Exceptions.ValidationException;public class ValidationBehaviour<TRequest, TResponse> : IPipelineBehav
 where TRequest : IRequest<TResponse>
 {
 private readonly IEnumerable<IValidator<TRequest>> _validators;public ValidationBehaviour(IEnumerable<IValidator<TRequest>> validators)
 {
 _validators = validators;
 }public async Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken, RequestHandlerDelegate<TResponse> next)
 {
 if (_validators.Any())
 {
 var context = new ValidationContext<TRequest>(request);var validationResults = await Task.WhenAll(_validators.Select(v => v.ValidateAsync(context, ca
 var failures = validationResults.SelectMany(r => r.Errors).Where(f => f != null).ToList();if (failures.Count != 0)
 throw new ValidationException(failures);
 }
 return await next();
 }
 }
```

This class basically, run all validation classes if exist and collect results. If there is any unvalidate result throwing Custom ValidationException.
This class searching FluentValidator — AbstractValidator objects with refllection. And run all validators before executing request.
Very good features for performing validation middleware by intercepting mediatR cqrs requests.

Developing Ordering.Application Layer — Application Service Registrations

We are going to Develop Application Service Registrations of Ordering.Application Layer with CQRS Pattern Implementation in Clean Architecture.

As you know that, In aspnet applications we have registered our objects into aspnet built-in depdency injection tool.
This register operation handled in;
Startup — ConfigureServices method

```
public void ConfigureServices(IServiceCollection services)
 {
 services.AddApplicationServices();
```

So this is get IServiceCollection and adding services with using Add methods.
In order to create our extention method we should developed our extention method in Application Layer. It is good practice to handle every dependencies into their own layer.
— Create new Class

ApplicationServiceRegistration.cs

```
public static class ApplicationServiceRegistration
 {
 public static IServiceCollection AddApplicationServices(this IServiceCollection services)
 {
 services.AddAutoMapper(Assembly.GetExecutingAssembly());
 services.AddValidatorsFromAssembly(Assembly.GetExecutingAssembly());
 services.AddMediatR(Assembly.GetExecutingAssembly());

 services.AddTransient(typeof(IPipelineBehavior<,>), typeof(UnhandledExceptionBehaviour<,>));
 services.AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehaviour<,>));return services;
 }
 }
```

In order to use extention methods, we should install below packages

Install Packages :

- Install-Package AutoMapper.Extensions.Microsoft.DependencyInjection
- Install-Package FluentValidation.DependencyInjectionExtensions

Developing Ordering.API Presentation Layer in Clean Architecture

We are going to start to Developing Ordering.API Presentation Layer in Clean Architecture.

As you know that, we have developed Ordering. Core and Application layer.
With application layer we covered all business use cases without any infrastructure dependencies.

So now, I am pushing one step further and say that we can also develop Presentation API Controller classes witout any external infrastructure related objects.
Its one of the best practices that you can handle your apis without any external infrasturcture objects.
So this is very important to seperate infrastructure and your actual business logic. Not let me directly develop our API Controller classes without any external dependencies.

Verify dependencies :

Add Project References:
Ordering.Application
Ordering.Infrastructure

For now we only use Ordering.Application references.
After that we can create new controller for exposing our apis.

Ordering.API — Controllers
Add OrderController.cs

```
[ApiController]
 [Route(“api/v1/[controller]”)]
 public class OrderController : ControllerBase
 {
 private readonly IMediator _mediator;public OrderController(IMediator mediator)
 {
 _mediator = mediator ?? throw new ArgumentNullException(nameof(mediator));
 }

 [HttpGet(“{userName}”, Name = “GetOrder”)]
 [ProducesResponseType(typeof(IEnumerable<OrdersVm>), (int)HttpStatusCode.OK)]
 public async Task<ActionResult<IEnumerable<OrdersVm>>> GetOrdersByUserName(string userName)
 {
 var query = new GetOrdersListQuery(userName);
 var orders = await _mediator.Send(query);
 return Ok(orders);
 }// testing purpose
 [HttpPost(Name = “CheckoutOrder”)]
 [ProducesResponseType((int)HttpStatusCode.OK)]
 public async Task<ActionResult<int>> CheckoutOrder([FromBody] CheckoutOrderCommand command)
 {
 var result = await _mediator.Send(command);
 return Ok(result);
 }[HttpPut(Name = “UpdateOrder”)]
 [ProducesResponseType(StatusCodes.Status204NoContent)]
 [ProducesResponseType(StatusCodes.Status404NotFound)]
 [ProducesDefaultResponseType]
 public async Task<ActionResult> UpdateOrder([FromBody] UpdateOrderCommand command)
 {
 await _mediator.Send(command);
 return NoContent();
 }[HttpDelete(“{id}”, Name = “DeleteOrder”)]
 [ProducesResponseType(StatusCodes.Status204NoContent)]
 [ProducesResponseType(StatusCodes.Status404NotFound)]
 [ProducesDefaultResponseType]
 public async Task<ActionResult> DeleteOrder(int id)
 {
 var command = new DeleteOrderCommand() { Id = id };
 await _mediator.Send(command);
 return NoContent();
 }
 }
```

As you can see that we have very low code when performing main logics.
We only create mediatR cqrs request object and send this request with mediatR. Behind this action MediatR creates pipeline for request and trigger to handle method.

No infrastructure dependencies no application business logics. Presentation layer only responsible for exposing apis. Business rules handles in application layer.
But wait, where is implementations, how mediatR get order, insert update delete orders ?

If you run the application you will get error for not registering objects on dependency injection. We should register interfaces with implementations. You can continue to rest of the development with following the resources below.

Get Udemy Course with discounted — Microservices Architecture and Implementation on .NET.

Get the Source Code from AspnetRun Microservices Github