EUVIC:

THE GOOD *People*

.NET + CQRS

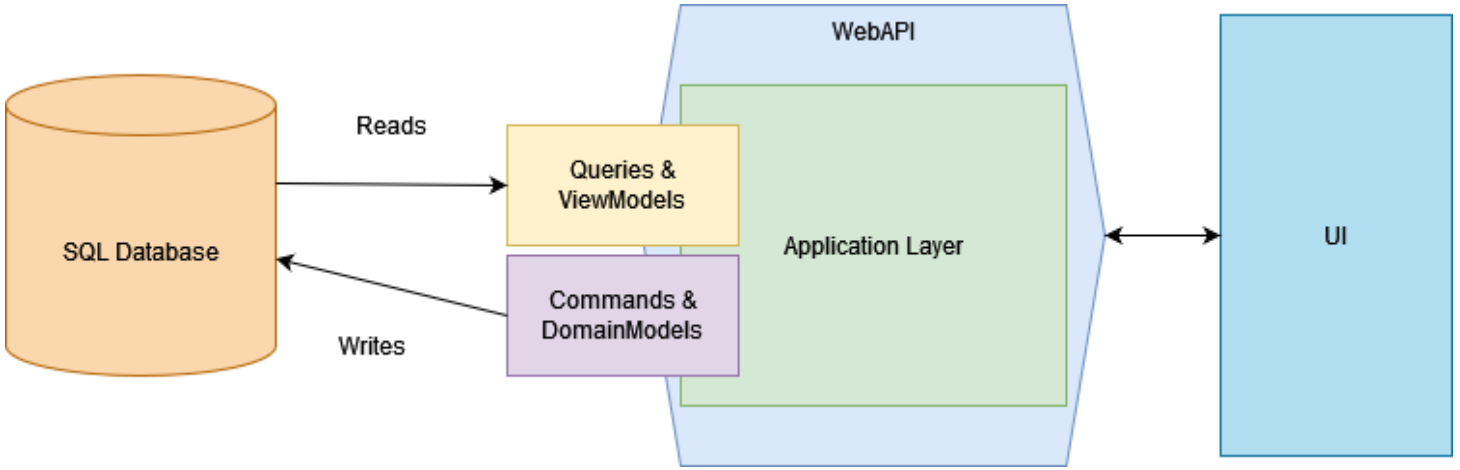
EUVIC:

Wzorzec projektowy CQRS

THE GOOD People

Czym jest CQRS

CQRS czyli Command Query Responsibility Segregation jest to wzorzec projektowy, który rozdziela odpowiedzalności zapisu i odczytu.



EUVIC:

Odczyt i zapis

THE GOOD People

Dlaczego warto rozdzielić odczyt od zapisu

- Reguły biznesowe dla zapisu są często inne niż dla odczytu
- Wydzielony kod tylko do odczytu powoduje ,że optymalizacja części systemu do odczytu jest łatwiejsza
- Rozdzielenie części systemu do zapisu powoduje ,że możemy mieć bardziej skomplikowane reguły biznesowe tylko do zapisu
- Rozdzielenie wprowadza porządek w kodzie
- Odczyt i zapis może być realizowany za pomocą 2 różnych technologii np zapis w SQL a odczyt ElasticSearch lub MongoDb
- Zazwyczaj ilość requestów dla odczytu jest wiele razy wieksza niż do zapisu. Dzięki rozdzieleniu także na poziomie aplikacji możemy skalować bardziej część tylko do oczytu

EUVIC:

CQRS

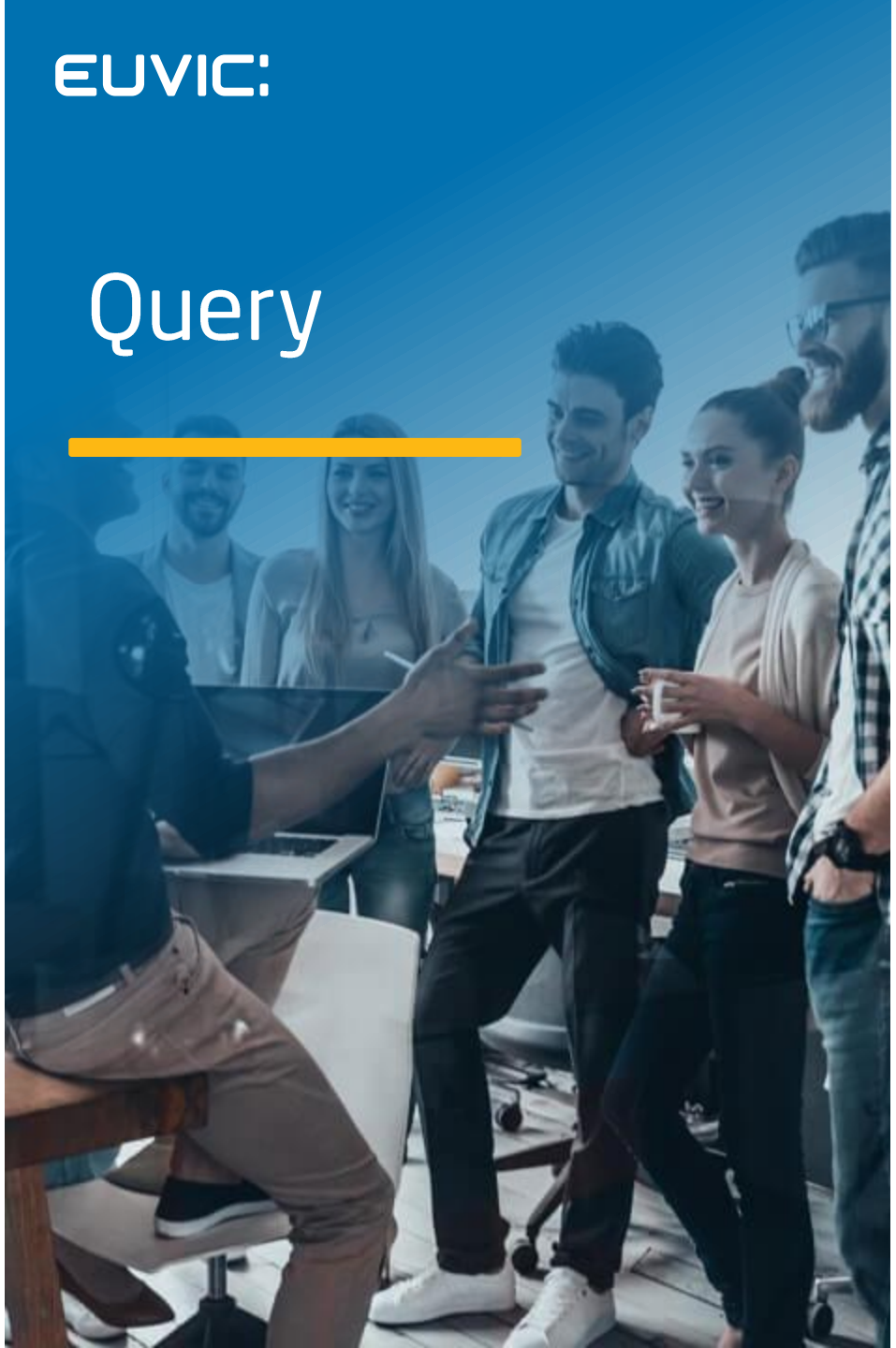
THE GOOD People

Kilka nowych pojęć

- Query – klasa reprezentująca intencje (model) użytkownika do odczytu danych
- Command – klasa reprezentująca intencje (model) użytkownika do zapisu danych
- Handler – klasa która odpowiada za wykonanie kodu przeznaczonego dla Query lub Command
- QueryHandler– klasa która implementuje jedno konkretne Query czego rezultatem jest odczyt z systemu
- CommandHandler – klasa która implementuje pojedyncze Command czego rezultatem jest zapis do systemu
- MediatR – Biblioteka za pomocą której można zrealizować implementację wzorca CQRS

EUVIC:

Query



THE GOOD People

CQRS - Query

- Query służy wyłącznie do odczytu i nie powinno zmieniać stanu systemu
- Jeżeli z jakiegoś powodu Query miało by zmienić system należy wtedy stworzyć event a zmianę systemu zrealizować w evencie a nie w QueryHandlerze
- Jeżeli korzystamy z SQL na potrzeby odczytu polecam, aby korzystać z EntityFramework wraz z bezpośrednią projekcją na model. Po czasie, pojedyncze problematyczne zapytania możemy przepisać na czysty SQL i użyć Dappera, ale tylko tam gdzie jest to naprawdę potrzebne

EUVIC:

Command

THE GOOD People

CQRS - Command

- Command służy do zapisu i nie powinno zwracać danych
- Osobiście dopuszczam zwracanie Id stworzonej encji podczas wykonywania command CreateSomething
- Oczywiście nie dajmy się zwariować i w pojedynczych przypadkach tam gdzie miało by to znaczny wplyw na wydajność, możemy zaakceptować zwracanie danych przez Command

EUVIC:

MediatR

Biblioteka MediatR

THE GOOD People

Jimmy Bogard stworzył świetną bibliotekę która umożliwa implementację wzorcja CQRS. Sam MediatR posiada generyczne nazwy na Query i Command nazwane „Request" dlatego warto zrobić sobie własną abstrakcję na interfejsy dostarczone przez MediatR

Originalne interfejsy

```
namespace MediatR;

public interface IRequest : IRequest<Unit> { }

public interface IRequest<out TResponse> : IBaseRequest { }
```

Proponononowana abstrakcja

```
namespace Euvic.Cqrs.Primitives
{
    15 references | 0 changes | 0 authors, 0 changes
    public interface ICommand : MediatR.IRequest { }
    5 references | 0 changes | 0 authors, 0 changes
    public interface ICommand<out TResult> : MediatR.IRequest<TResult> { }
}
```

```
namespace Euvic.Cqrs.Primitives
{
    12 references | 0 changes | 0 authors, 0 changes
    public interface IQuery<out TResult> : MediatR.IRequest<TResult> { }
}
```

EUVIC:

MediatR

Jak uzyć MediatR

THE GOOD People

Aby automatycznie zarejestrować wszystkie Query oraz Command należy użyć komendy:

```
0 references | 0 changes | 0 authors, 0 changes
public void ConfigureServices(IServiceCollection services)
{
    services.AddMediatR(typeof(Startup));
}
```

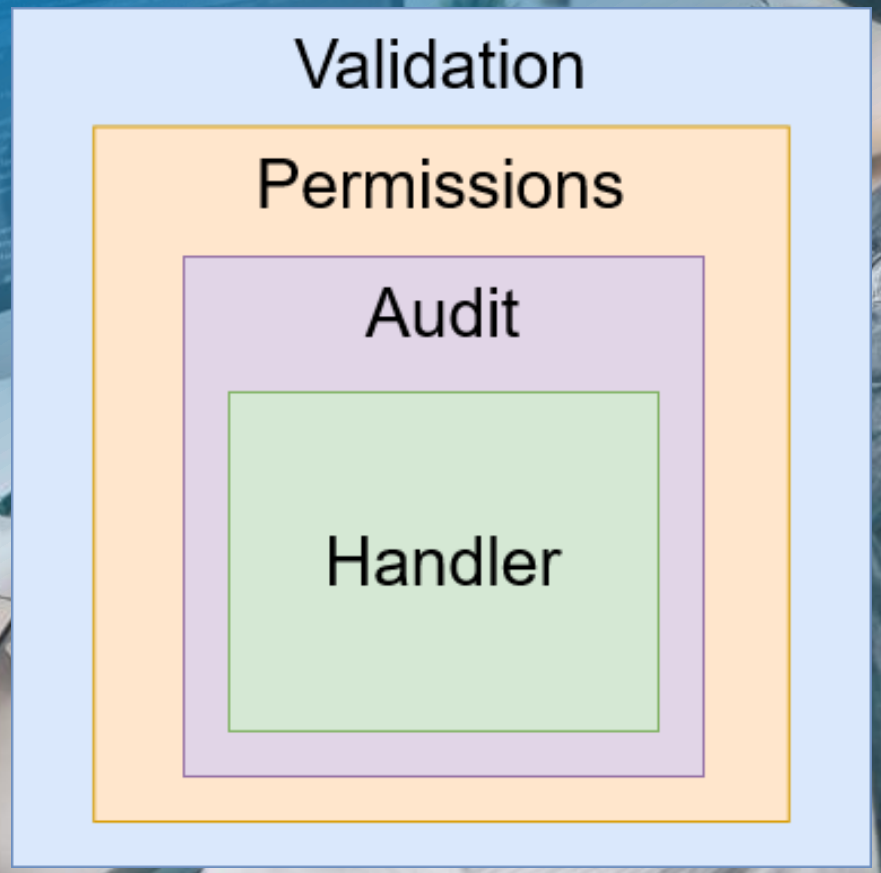
Wykonanie tej komendy powiąże nam wszystkie IQuery oraz ICommand z Handlerami które to implementują.

Następnie w dowolnym miejscu możemy wstrzyknąć Interfejs „*IMediator*" i wywołać metodę Send()

```
private readonly IMediator _mediator;

0 references | 0 changes | 0 authors, 0 changes
public AttendeesController(IMediator mediator)
{
    _mediator = mediator;
}

[HttpGet("me/profile")]
0 references | 0 changes | 0 authors, 0 changes
public async Task<IActionResult> GetAttendeeProfile()
{
    var profile = await _mediator.Send(new GetAttendeeProfile.Query());

    return Ok(profile);
}
```

EUVIC:

Pipeline Behavior



THE GOOD People

Co to jest pipeline behavior

- Pipeline behavior pozwala na wykonanie dodatkowego kodu zanim odpowiedni handler zostanie wywołany
- Pipeline behaviors wywołują się w kolejności w jakiej zostały zarejestrowane
- W przypadku gdy jakies reguły biznesowe zostały złamane polecam rzucać wyjątki konkretnych typów a następnie filtrami łapać je na poziomie WebAPI

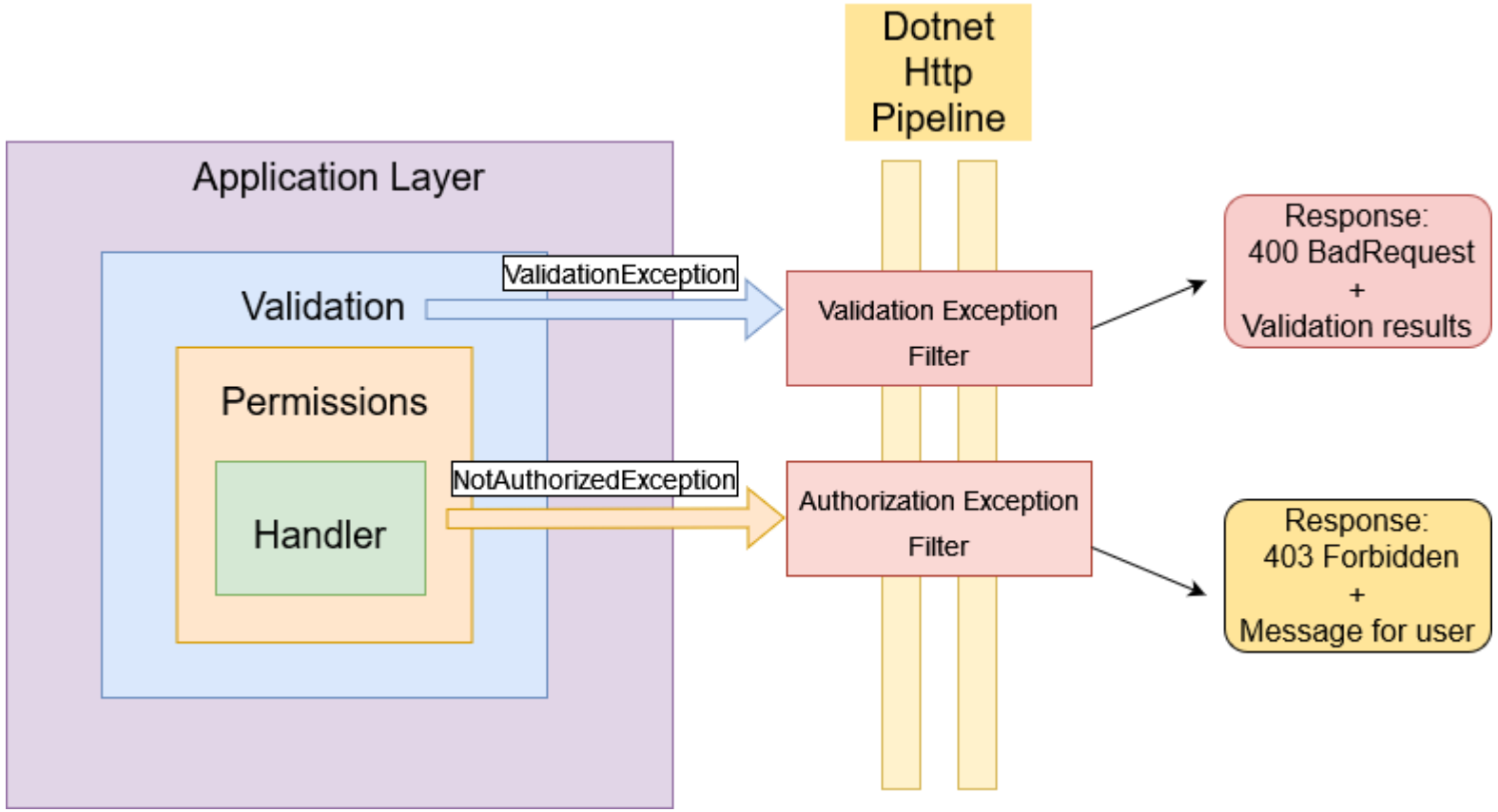
Przykladowe zastosowania pipeline behaviors:

- Walidacja
- Autoryzacja użytkownika związana z logiką biznesową
- Mierzenie czasu wykonania Query lub Command

Dodatkowo używałem pipeline behavior do:

- Stworzenia audytu dla każdego command
- Akcji kompensacyjnych, aby zachować spójność i odwrócić stan po błędzie w przypadku gdy nie możemy mieć tranzakcji (np. komunikacja pomiędzy systemami)

Schemat komunikacji z WebAPI



EUVIC:

Command

THE GOOD People

Zalety CQRS

- Każdy feature reprezentowany jako query lub command może mieć inną własną implementacje i własne zależności. W tej kwestii dostajemy całkowitą wolność implementacji per feature.
- Łatwo jest się odnaleźć osobom, które przychodzą do projektu porównując to z typową architekturą warstwową
- Każdy feature ma własną klasę przez to cały projekt jest łatwy w utrzymaniu w przeciwieństwie do serwisów które maja po 1000 lini i są odpowiedzialne za wiele rzeczy a każda metoda ma wiele referencji przez co strach ją modyfikować
- Stosowanie pipeline behavior otwiera nieskończony wahlarz możliwości aby dopinać dodatkową logikę dla każdego query lub command
- Możliwość zastosowania jednej technologi do zapisu danych a innej do odczytu

EUVIC:

THE GOOD *People*

www.euvic.com

Przewozowa 32 | 44-100 Gliwice
+48 32 279 49 42 | info@euvic.pl