Kamil Grzybek

Doing The Right Things Right

Programming and designing enterprise solutions with .NET

Domain Model Validation

Introduction

In **previous post** I described how requests input data can be validated on *Application Services Layer*. I showed **FluentValidation** library usage in combination with **Pipeline Pattern** and **Problem Details** standard. In this post I would like to focus on the second type of validation which sits in the *Domain Layer – Domain Model* validation.

What is Domain Model validation

We can divide the validation of the *Domain Model* into two types based on scope – *Aggregates* scope and *Bounded Context* scope

Aggregates scope

Let's remind what the *Aggregate* is by quoting a fragment of the text from Vaughn Vernon **Domain-Driven Design Distilled** book:

> *Each Aggregate forms a transactional consistency boundary. This means that within a single Aggregate, all composed parts must be consistent, according to business rules, when the controlling transaction is committed to the database.*

The most important part of this quote in context of validation I underlined. It means that **under no circumstances** we can't persist Aggregate to database which has invalid state or breaks business rules. These rules are often called "**invariants**" and are defined by Vaughn Vernon as follows:

> *… business invariants — the rules to which the software must always adhere — are guaranteed to be consistent following each business operation.*

So in context of *Aggregates* scope, we need to protect these invariants by executing validation during our use case (business operation) processing.

Bounded Context scope

Unfortunately, validation of *Aggregates* invariants is not enough. Sometimes the business rule may apply to more than one *Aggregate* (they can be even aggregates of different types).

For example, assuming that we have Customer Entity as *Aggregate Root*, the business rule may be *"Customer email address must be unique"*. To check this rule we need to check all emails of Customers which are separated *Aggregate Roots*. It is outside of the scope of one Customer aggregate. Of course, supposedly, we could create new entity called CustomerCatalog as *Aggregate Root* and aggregate all of the Customers to it but this is not good idea for many reasons. The better solution is described later in this article.

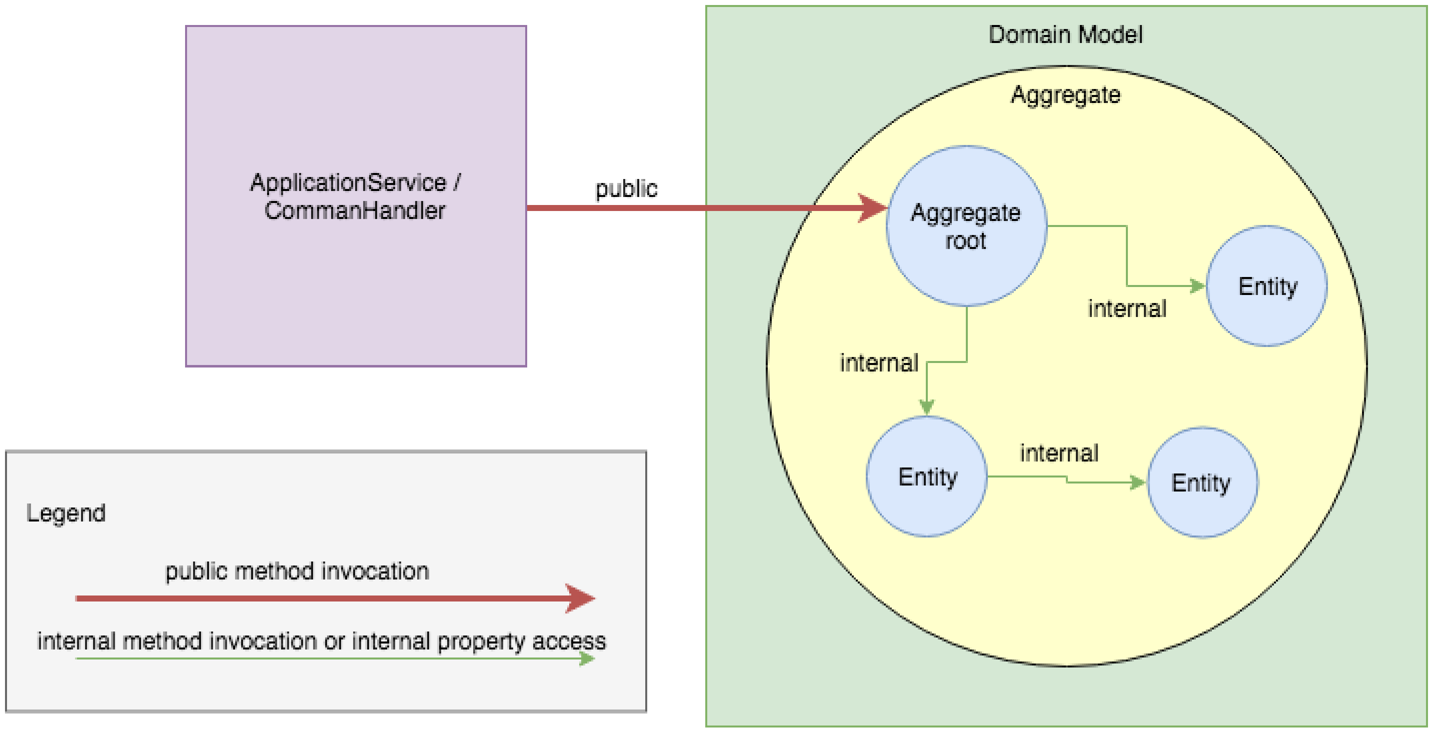
Let's see what options we have to solve both validation problems.

Three solutions

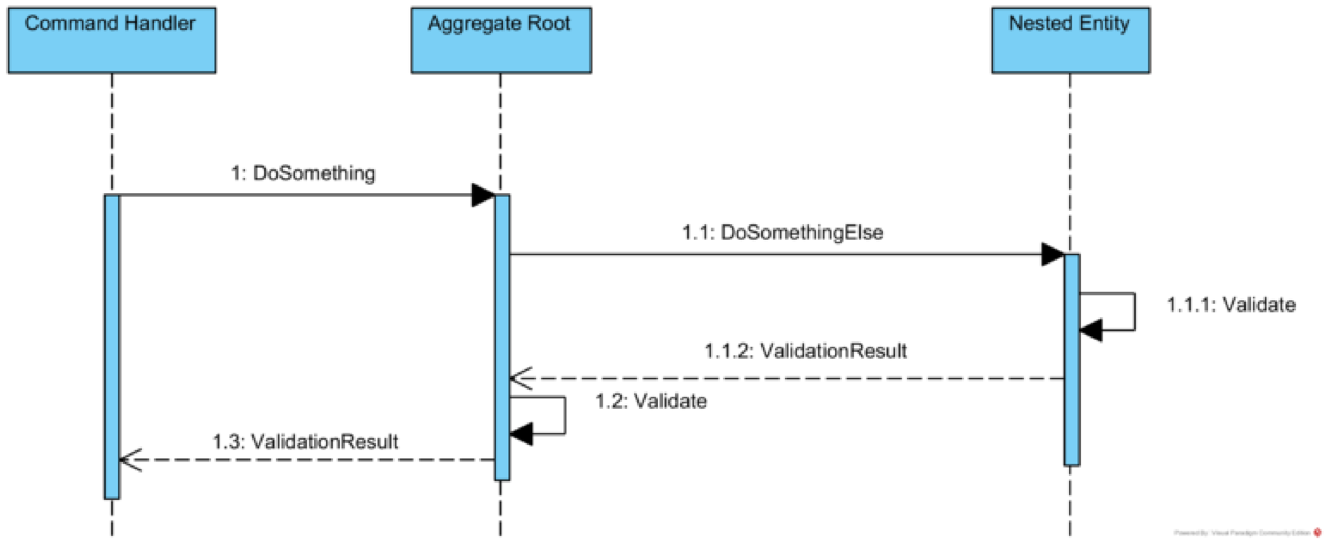
Return Validation Object

This solution is based on **Notification Pattern**. We are defining special class called Notification/ValidationResult/Result/etc which "*collects together information about errors and other information in the domain layer and communicates it*".

What does it mean for us? Is means that for every entity method which **mutates the state** of *Aggregate* we should return this validation object. The keyword here is **entity** because we can have (and we likely will have) nested invocations of methods inside *Aggregate*. Recall the diagram from the **post** about Domain Model encapsulation:



The program flow will look like:



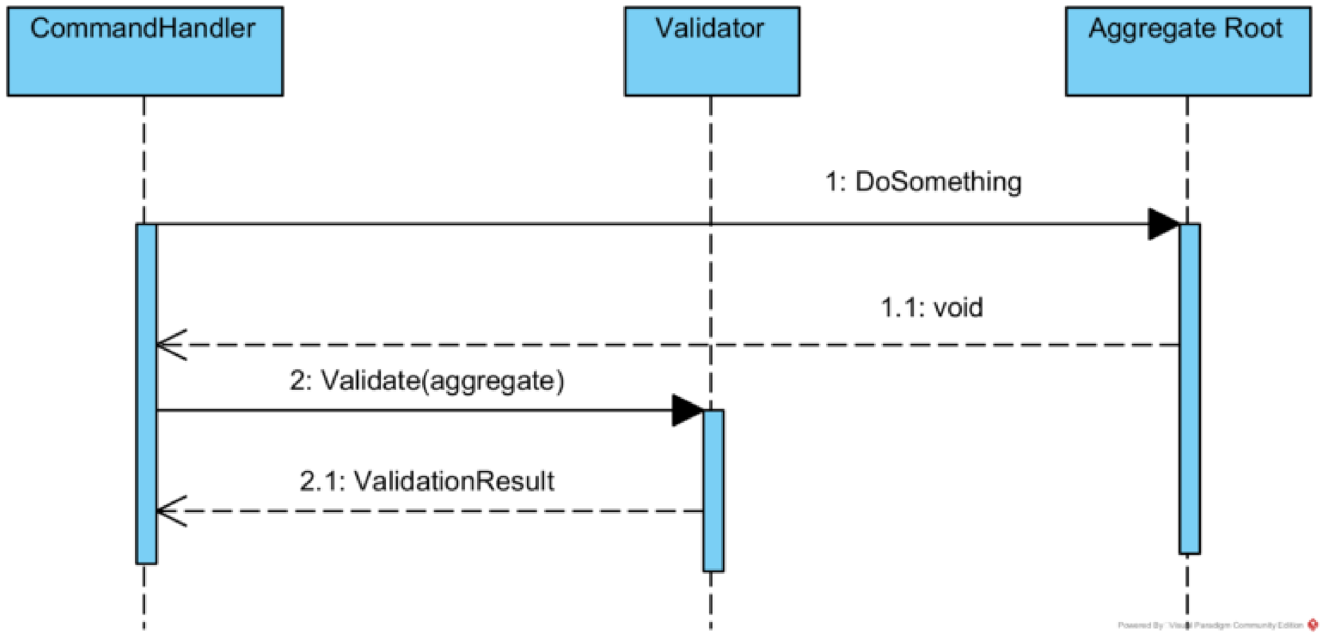
and the code structure (simplified):

```
public class AggregateRoot
{
    private NestedEntity _nestedEntity;

    public ValidationResult DoSomething()
    {
        // logic..

        ValidationResult nestedEntityValidationResult = _nestedEntity.DoSomethingElse();
        return Validate(nestedEntityValidationResult);
    }

    private ValidationResult Validate(ValidationResult nestedEntityValidationResult)
    {
        // Validate AggregateRoot and check nestedEntityValidationResult.
    }
}

public class NestedEntity
{
    public ValidationResult DoSomethingElse()
    {
        // logic..
        return Validate();
    }

    private ValidationResult Validate()
    {
        //NestedEntity validation...
    }
}
```

However, if we don't like to return `ValidationResult` from every method which mutates the state we can apply different approach which I described in **article** about publishing *Domain Events*. In short, in this solution we need to add `ValidationResult` property for every *Entity* (as Domain Events collection) and after *Aggregate* processing we have to examine these properties and decide if the whole *Aggregate* is valid.

Deferred validation

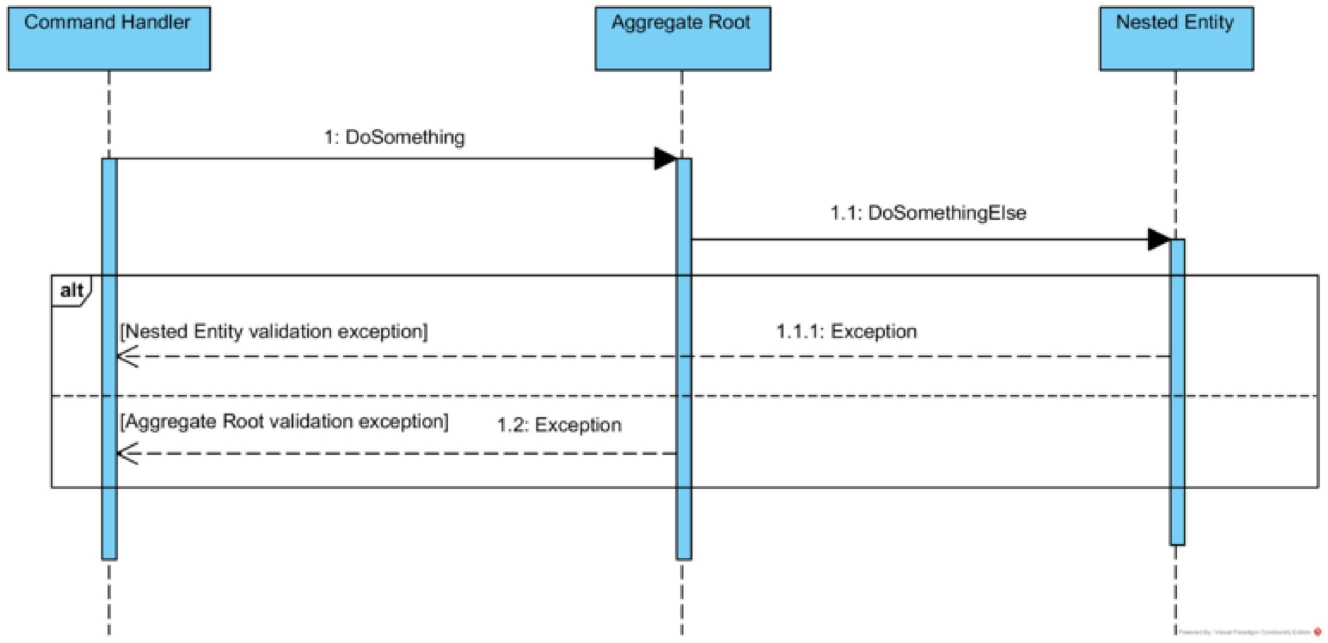
Second solution how to implement validation is to execute checking after whole *Aggregate's* method is processed. This approach is presented for example by Jeffrey Palermo in his **article**. The whole solution is pretty straightforward:



```
public async Task<Unit> Handle(AddCustomerOrderCommand request, CancellationToken cancellationToken)
{
    var customer = await this._customerRepository.GetByIdAsync(request.CustomerId);

    // ....

    var order = new Order(orderProducts);

    customer.AddOrder(order);

    ValidationResult validationResult = ValidateCustomer(customer);

    await this._customerRepository.UnitOfWork.CommitAsync(cancellationToken);

    return Unit.Value;
}

private ValidationResult ValidateCustomer(Customer customer)
{
    Validatior validator = new Validator();
    return validator.Validate(customer);
}
```

Always Valid

Last but not least solution is called “Always Valid” and it’s just about throwing exceptions inside *Aggregate* methods. It means that we finish processing of the business operation **with the first violation** of the *Aggregate invariant*. In this way, we are assured that our *Aggregate* is always valid:



Comparison of solutions

I have to admit that I don’t like *Validation Object* and *Deferred Validation* approach and I recommend *Always Valid* strategy. My reasoning is as follows.

Returning Validation Object approach pollutes our methods declarations, adds accidental complexity to our *Entities* and is against **Fail-Fast** principle. Moreover, Validation Object becomes part of our *Domain Model* and it is for sure not part of **ubiquitous language**. On the other hand Deferred Validation implies not encapsulated *Aggregate*, because the validator object **must have access to aggregate internals** to properly check invariants.

However, both approaches have one advantage – they do not require throwing exceptions which should be thrown only when something **unexpected** occurs. Business rule broken is not unexpected.

Nevertheless, I think this is one of the rare exception when we can break this rule. For me, throwing exceptions and having always valid *Aggregate* is the best solution. *"The ends justify the means"* I would like to say. I think of this solution like implementation of **Publish-Subsribe Pattern**. *Domain Model* is the Publisher of broken invariants messages and Application is the Subscriber to this messages. The main assumption is that after publishing message the publisher stops processing because this is how exceptions mechanism works.

Always Valid Implementation

Exception throwing is built into the C# language so practically we have everything. Only thing to do is create specific `Exception` class, I called it `BusinessRuleValidationException`:

```
public class BusinessRuleValidationException : Exception
{
    public string Details { get; }

    public BusinessRuleValidationException(string message) : base(message)
    {

    }

    public BusinessRuleValidationException(string message, string details) : base(message)
    {
        this.Details = details;
    }
}
```

Suppose we have a business rule defined that you cannot order more than 2 orders on the same day. So it looks implementation:

```
// Customer aggregate root.
public void AddOrder(Order order)
{
    if (this._orders.Count(x => x.IsOrderedToday()) >= 2)
    {
        throw new BusinessRuleValidationException("You cannot order more than 2 orders on the same day");
    }

    this._orders.Add(order);

    this.AddDomainEvent(new OrderAddedEvent(order));
}
```

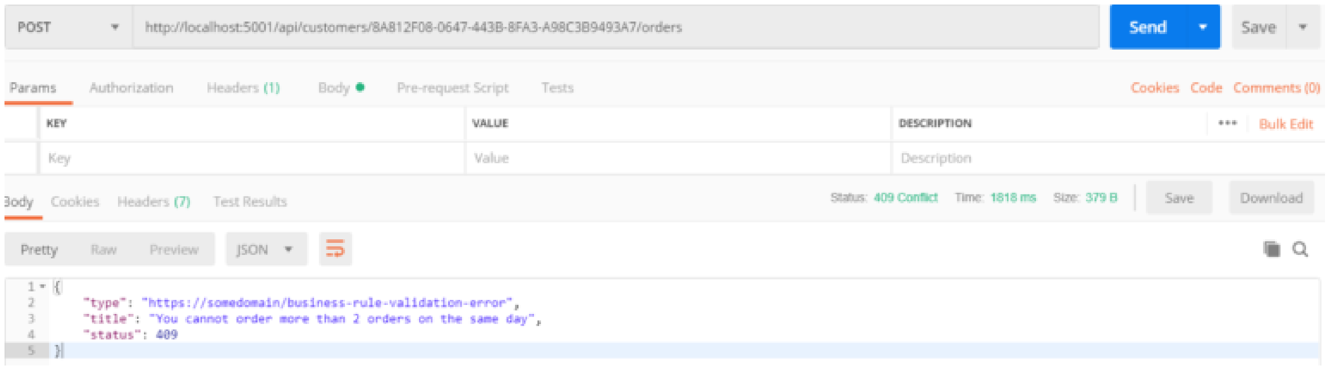
```
// Order entity.
internal bool IsOrderedToday()
{
   return this._orderDate.Date == DateTime.UtcNow.Date;
}
```

What we should do with the thrown exception? We can use approach from **REST API Data Validation** and return appropriate message to the client as Problem Details object standard. All we have to do is to add another `ProblemDetails` class and set up mapping in `Startup`:

```
public class BusinessRuleValidationExceptionProblemDetails : Microsoft.AspNetCore.Mvc.ProblemDetails
{
    public BusinessRuleValidationExceptionProblemDetails(BusinessRuleValidationException exception)
    {
        this.Title = exception.Message;
        this.Status = StatusCodes.Status409Conflict;
        this.Detail = exception.Details;
        this.Type = "https://somedomain/business-rule-validation-error";
    }
}
```

```
services.AddProblemDetails(x =>
{
    x.Map<InvalidCommandException>(ex => new InvalidCommandProblemDetails(ex));
    x.Map<BusinessRuleValidationException>(ex => new BusinessRuleValidationExceptionProblemDetails(ex));
});
```

The result returned to client:



http://localhost:5001/api/customers/8A812F08-0647-443B-8FA3-A98C3B9493A7/orders

Status: 409 Conflict Time: 1818 ms Size: 379 B

```
{
    "type": "https://somedomain/business-rule-validation-error",
    "title": "You cannot order more than 2 orders on the same day",
    "status": 409
}
```

For simpler validation like checking for nulls, empty lists etc you can create library of guards (see **Guard Pattern**) or you can use external library. See **GuardClauses** created by Steve Smith for example.

BC scope validation implementation

What about validation which spans multiple *Aggregates* (*Bounded Context* scope)? Let's assume that we have a rule that there cannot be 2 Customers with the same email address. There are two approaches to solve this.

The first way is to get required aggregates in `CommandHandler` and then pass them to aggregate's method/constructor as arguments:

```
public async Task<CustomerDto> Handle(RegisterCustomerCommand request, CancellationToken cancellationToken)
{
    var allCustomers = await _customerRepository.GetAll();
    var customer = new Customer(request.Email, request.Name, allCustomers);

    await this._customerRepository.AddAsync(customer);

    await this._customerRepository.UnitOfWork.CommitAsync(cancellationToken);

    return new CustomerDto { Id = customer.Id };
}
```

```
public Customer(string email, string name, List<Customer> allCustomers)
{
    if (allCustomers.Contains(email))
    {
        throw new BusinessRuleValidationException("Customer with this email already exists.");
    }
    this.Email = email;
    this.Name = name;

    this.AddDomainEvent(new CustomerRegisteredEvent(this));
}
```

However, this is not always a good solution because as you can see we need to **load all** Customer Aggregates to memory. This could be serious performance issue. If we can not afford it then we need to introduce second approach – create *Domain Service* which is defined as (source – **DDD Reference**):

> *When a significant process or transformation in the domain is not a natural responsibility of an entity or value object, add an operation to the model as a standalone interface declared as a service*

So, for that case we need to create `ICustomerUniquenessChecker` service interface:

```
public interface ICustomerUniquenessChecker
{
    bool IsUnique(Customer customer);
}
```

This is the implementation of that interface:

```
public class CustomerUniquenessChecker : ICustomerUniquenessChecker
{
    private readonly ISqlConnectionFactory _sqlConnectionFactory;

    public CustomerUniquenessChecker(ISqlConnectionFactory sqlConnectionFactory)
    {
        _sqlConnectionFactory = sqlConnectionFactory;
    }

    public bool IsUnique(Customer customer)
    {
        using (var connection = this._sqlConnectionFactory.GetOpenConnection())
        {
            const string sql = "SELECT TOP 1 1" +
                               "FROM [orders].[Customers] AS [Customer] " +
                               "WHERE [Customer].[Email] = @Email";
            var customersNumber = connection.QuerySingle<int?>(sql,
                            new
                            {
                                customer.Email
                            });

            return !customersNumber.HasValue;
        }
    }
}
```

Finally, we can use it inside our Customer Aggregate:

```
public Customer(string email, string name, ICustomerUniquenessChecker customerUniquenessChecker)
{
    this.Email = email;
    this.Name = name;

    var isUnique = customerUniquenessChecker.IsUnique(this);
    if (!isUnique)
    {
        throw new BusinessRuleValidationException("Customer with this email already exists.");
    }

    this.AddDomainEvent(new CustomerRegisteredEvent(this));
}
```

The question here is whether pass *Domain Service* as an argument to aggregate's constructor/method or execute validation in *Command Handler* itself. As you can see above I am fan of former approach because I like keep my command handlers very thin. Another argument for this option is that if I ever need to register Customer from a different use case I will not be able to bypass and forget about this uniqueness rule because I will have to pass this service.

Summary

A lot of was covered in this post in context of Domain Model Validation. Let's summarize:
– We have two types of *Domain Model* validation – *Aggregates* scope and *Bounded Context* scope
– There are generally 3 methods of *Domain Model* validation – using *Validation Object*, *Deferred Validation* or *Always Valid* (throwing exceptions)
– *Always Valid* approach is preferred
– For *Bounded Context* scope validation there are 2 methods of validations – passing all required data to aggregate's method or constructor or create *Domain Service* (generally for performance reason).

Source code

If you would like to see full, working example – check my **GitHub repository**.

Additional Resources

Validation in Domain-Driven Design (DDD) – Lev Gorodinski
Validation in a DDD world– Jimmy Bogard

Related posts

REST API Data Validation
Domain Model Encapsulation and PI with Entity Framework 2.2
Simple CQRS implementation with raw SQL and DDD
How to publish and handle Domain Events

SHARE THIS:

Print Facebook LinkedIn Twitter Pocket

March 4, 2019 Kamil Grzybek Design DDD, Design patterns, REST API, Validation

TAKŻE NA **KAMIL GRZYBEK**

Strangling .NET Framework App to …
rok temu • 10 komentarzy
The incremental approach to migrate from .NET Framework legacy system …

Using Database Project and DbUp …
2 lata temu • 28 komentarzy
Using .NET Database Project and DbUp library for database management.

The Outbox Pattern
2 lata temu • 15 komentarzy
Implementation of Outbox Pattern

Modular M Integratio
10 miesięcy ten
Description o Styles used to modules in M

Komentarze **Społeczność** **polityką prywatności**

1 **Zaloguj**

Poleć 4 Tweet Udostępnij

Sortuj według najlepszych

Dołącz do dyskusji…

ZALOGUJ SIĘ ZA POMOCĄ ALBO ZAREJESTRUJ W DISQUS ?

Nazwa

Luís Barbosa • 2 lata temu

Hi Kamil, great article.

I particularly like this phrase: "I think of this solution as implementation of Publish-Subsribe Pattern." Domain Model is the Publisher of broken invariants messages and Application is the Subscriber to this messages. this is how exceptions mechanism works.". I never thought of it that way, but it makes perfect sense.

2 ^ | ⌄ • Odpowiedz • Udostępnij ›

Mateusz Nowak • 9 miesięcy temu

Hi Kamil,
I have a question for last example.
As I see your business rule checked in CustomerRegisteredEvent can be broken. Before the check for uniqueness is not atomic. It's can be unique while invoking isUnique and in the moment of saving new customer it's can be broken. How do you handle such situation - for email uniqueness (etc)?

^ | ⌄ • Odpowiedz • Udostępnij ›

Kamil Grzybek Mod ➦ Mateusz Nowak
• 9 miesięcy temu

Hi @**Mateusz Nowak**

Yes, I agree - there is a little, finite period of time between check and save. In that case you can:
a) Create an unique key on database (treating db as last line of defense) pro: performance con: your rule leaked a little to the infrastructure
b) Create an Aggregate and use concurrency control. Pros and cons: opposite to a) - this will not be

efficient (you must load all emails to Aggregate) but your rule will not leak to infrastructure.

^ | ⌄ • Odpowiedz • Udostępnij ›

Arthur Hoffmann ➦ Kamil Grzybek • 6 miesięcy temu

What you think about a approach c): call customer.Activate(customerUniquenessCheck an then repeat the verification (will find another customer created simultaneously). My ideia is that the customer will initially in the state PENDING/DEACTIVATED and after call Active will transit to ACTIVATED.

^ | ⌄ • Odpowiedz • Udostępnij ›

Kamil Grzybek Mod ➦ Arthur Hoffmann • 5 miesięcy temu

I think repeating does not add value here because always will be period of time between check and save.

So to be one hundred percent sure that this will not be broken is to read record with lock and then save. And this is how db unique constraint works - so why not to use if it is the most efficient way to do that (the time between read and check is the shortest)

^ | ⌄ • Odpowiedz • Udostępnij ›

Shooresh Golzari • 9 miesięcy temu

Hello, I cannot find the library containing the AddProblemDetails middleware as you have it in your solution. Can you point me in the right direction? Thank you.

^ | ⌄ • Odpowiedz • Udostępnij ›

Kamil Grzybek Mod ↪ Shooresh Golzari
• 9 miesięcy temu

Sure: https://www.nuget.org/packa...

• Odpowiedz • Udostępnij ›

Arthur Borisow • rok temu

Hi Kamil, nice article.

But I've got a question about last example. I understand why you pass the service to the domain model (though this is the first time I see it), but how should repository then build you entity. Turns out repository will have to know about services and it breaks the architecture I guess. And even if so, when you return a customer from repository you will still have to provide this service to entity despite the fact that you might need it ony for reading

• Odpowiedz • Udostępnij ›

Kamil Grzybek Mod ↪ Arthur Borisow
• 9 miesięcy temu • edited

Hi @**Arthur Borisow**

Repository uses a different constructor. Entity framework can use private parameterless constructor to create the Entity. Only public/internal constructor are used by application.

Now even I **try to keep all constructors private and use Factory Methods to create entities** for the application. This is the best approach in my opinion.

• Odpowiedz • Udostępnij ›

Arthur Borisow ↪ Kamil Grzybek
• 9 miesięcy temu

Thank you for the answer. Didn't think you

would reply =)
Yep, I saw your projects on github concerning DDD - they answered my question as well

• Odpowiedz • Udostępnij ›

Rabbal • 2 lata temu

Hi Kamil,

Exception handling for flow control is EVIL!
Exceptions for flow control in C#
Replacing Throwing Exceptions with Notification in Validations

So we can use a factory method for section "BC scope validation implementation" like this:

```
public class Customer
{
public string Email { get; private set; }
public string Name { get; private set; }

private Customer(email, name)
{
Email = email;
Name = name;
}

public static Result<customer> New(string email, string name,
INewCustomerPolicy policy)
{
var isUnique = policy.IsUnique(email);
if (!isUnique)
{
return Result.Fail<customer>("Customer with this email
already exists.");
}

var customer = new Customer(email, name);

//customer.AddDomainEvent(new CustomerRegistered(customer));

return Result.Ok(customer);
}
}
```

Result class accessible in DNTFrameworkCore

• Odpowiedz • Udostępnij ›

Kamil Grzybek Mod Rabbal • 2 lata temu

Proudly powered by WordPress