Implementacja DDD .NET - Zadanie 7

Bottega IT Minds

1. Zadanie 7 - Event Sourcing

1.1. Wprowadzenie

Stan naszego systemu możemy persystować rejestrując wszystkie *zdarzenia* a następnie je odtwarzając jedno po drugim (*projekcja*).

W module *Payments* tak został zaimplementowany agregat `Wallet` (odpowiedzialny za płatności w systemie).

```
public class Wallet : AggregateRootBaseEventSourcing
{
    private Money _balance;

    private Money _loan;

    private Money _loanLimit;

    private Money _remainingLoanLimit;

    private Wallet()
    {
        // For Marten deserialization.
    }

    private Wallet(Guid payerId)
    {
        var @event = new WalletAddedDomainEvent(
            payerId,
            Money.Of(0),
            Money.Of(0),
            Money.Of(0),
            Money.Of(0));

        Apply(@event);

        AddUncommittedEvent(@event);
    }

    public static Wallet Add(Guid payerId)
    {
        return new Wallet(payerId);
    }

    public void Apply(WalletAddedDomainEvent @event)
    {
        Id = @event.PayerId;
        _balance = Money.Of(@event.Balance);
        _loan = Money.Of(@event.Loan);
```

```
        _loanLimit = Money.Of(@event.LoanLimit);
        _remainingLoanLimit = Money.Of(@event.RemainingLoanLimit);

        Version++;
    }

    public void AddFunds(Money amount)
    {
        var @event = new FundsAddedDomainEvent(amount);
        Apply(@event);

        AddUncommittedEvent(@event);
    }

    public void Pay(Money amount)
    {
        CheckRule(new SumOfBalanceAndRemainingLoanMustBeGreaterOrEqualThanAmountRule(
            amount,
            _balance,
            _remainingLoanLimit
        ));

        if (_balance >= amount)
        {
            var @event = new BalanceDecreasedDomainEvent(amount);
            Apply(@event);

            AddUncommittedEvent(@event);
        }
        else if (_balance + _remainingLoanLimit >= amount)
        {
            var balanceDecreasedAmount = Money.Of(_balance);
            var balanceDecreasedEvent = new
BalanceDecreasedDomainEvent(balanceDecreasedAmount);
            Apply(balanceDecreasedEvent);
            AddUncommittedEvent(balanceDecreasedEvent);

            _balance = Money.Of(0);
            var toLoan = amount - balanceDecreasedAmount;

            var loanIncreasedEvent = new LoanIncreasedDomainEvent(toLoan);

            Apply(loanIncreasedEvent);
            AddUncommittedEvent(loanIncreasedEvent);
        }
    }

    public void Apply(LoanIncreasedDomainEvent @event)
    {
        _loan += @event.Amount;
        _remainingLoanLimit -= _loanLimit - _loan;
```

```
        Version++;
    }

    public void SetLoanLimit(Money loanLimit)
    {
        var @event = new LoanLimitSetDomainEvent(loanLimit);
        Apply(@event);

        AddUncommittedEvent(@event);
    }

    public void Apply(BalanceDecreasedDomainEvent @event)
    {
        _balance -= @event.Amount;

        Version++;
    }

    public void Apply(FundsAddedDomainEvent @event)
    {
        _balance += @event.Amount;

        Version++;
    }

    public void Apply(LoanLimitSetDomainEvent @event)
    {
        _loanLimit = @event.LoanLimit;
        _remainingLoanLimit = _loanLimit - _loan;

        Version++;
    }
}
```

Jako, że zdarzenia dobrze spisują się podczas zapisu, jednak przy odczycie najczęśniej potrzebujemy aktualnego stanu, została utworzona projekcja, która reprezentuje aktualny stan portfela:

```
public class WalletReadModelProjection : SingleStreamAggregation<WalletReadModel>
{
    public WalletReadModel Create(IEvent<WalletAddedDomainEvent> walletAdded)
    {
        return new WalletReadModel
        {
            PayerId = walletAdded.Data.PayerId,
            Balance = walletAdded.Data.Balance.Amount,
            Loan =  walletAdded.Data.Loan.Amount,
            LoanLimit = walletAdded.Data.LoanLimit.Amount,
            RemainingLoanLimit = walletAdded.Data.RemainingLoanLimit.Amount
```

```
        };
    }
}

public class WalletReadModel
{
    public Guid PayerId { get; set; }

    public decimal Balance { get; set; }

    public decimal Loan { get; set; }

    public decimal RemainingLoanLimit { get; set; }

    public decimal LoanLimit { get; set; }

    public void Apply(BalanceDecreasedDomainEvent balanceDecreasedDomainEvent)
    {
        Balance -= balanceDecreasedDomainEvent.Amount.Amount;
    }

    public void Apply(FundsAddedDomainEvent @event)
    {
        Balance += @event.Amount.Amount;
    }

    public void Apply(LoanIncreasedDomainEvent @event)
    {
        Loan += @event.Amount.Amount;
        RemainingLoanLimit = LoanLimit - Loan;
    }

    public void Apply(LoanLimitSetDomainEvent @event)
    {
        LoanLimit = @event.LoanLimit.Amount;
        RemainingLoanLimit = LoanLimit - Loan;
    }
}
```

1.2. Treść Zadania

Na bazie implementacji funkcjonalności związanej z portfelem (`Wallet`) w module *Payments* dodaj aggregat `Payer` i zaimplementuj go wykorzystując Event Sourcing. Każdy płatnik posiada *Nazwę* oraz flagę *Czy_aktywny (domyślnie TAK)*.

Zaimplementuj następujące przypadki użycia:

1. Tworzenie Płatnika
2. Dezaktywacja Płatnika

3. Zmiana nazwy Płatnika

Przetestuj przypadki użycia w sposób integracyjny.