Занятие 17. CQRS

Теория

CQRS (Command Query Responsibility Segregation) — это паттерн проектирования, который разделяет операции на два типа:

- команды (commands) - изменяют состояние бизнес-сущностей;
- запросы (queries) - не изменяют состояние бизнес-сущностей и возвращают какие-то данные.

Применение CQRS позволяет:

- улучшить способность системы к масштабированию;
- получить возможность улучшения производительности операций чтения без изменения кода прикладного уровня.

Достигается это за счет того, что при разделении операций на две понятных категории у нас появляется и возможность разделить источники данных для этих операций, например, следующим образом:

- все команды работают с master-репликой базы данных и имеют доступ к самому последнему состоянию бизнес-сущностей;
- все запросы работают либо с master-репликой, либо со slave-репликами базы данных, которые могут иметь не самое свежее состояние бизнес-сущностей.

Также возможны и другие стратегии для разделения источников данных, но это самый простой. Выбираемая стратегия должна зависеть от решаемой задачи и конкретных требований, предъявляемых к вашему приложению.

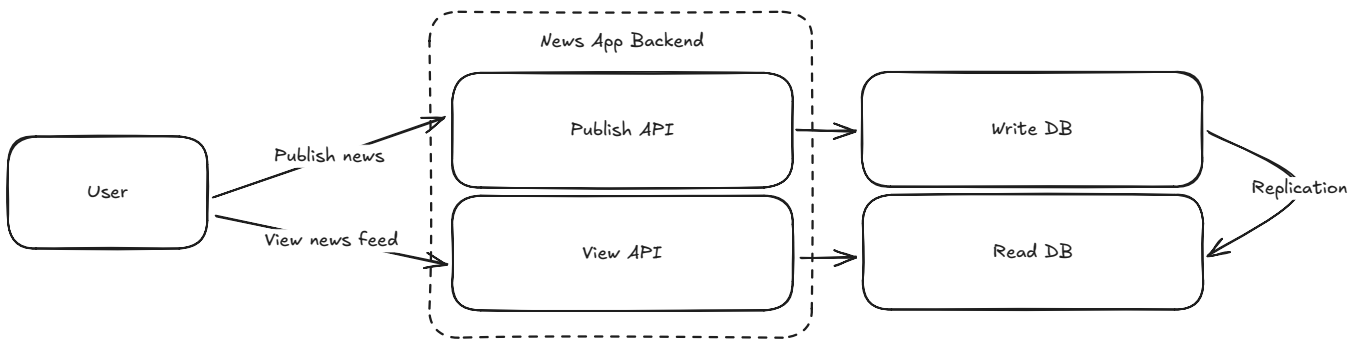
Практика

Для демонстрации паттерна CQRS рассмотрим задачу построения API для новостного портала.

Мы будем поддерживать три операции:

- Создание новости;
- Публикация новости;
- Получение новостной ленты.

Архитектура приложения будет следующей:



Как видно, у нас есть два типа API:

- API для публикации новостей;
- API для получения новостной ленты.

Каждое из API обращается к своей собственной базе данных. При этом данные из БД для записи попадают в БД для чтения путем репликации. Данное решение никак не позволяет нам предобрабатывать новости перед публикацией, но оно простое и позволит нам изучить паттерн CQRS на практике. Далее ничего не помешает нам при необходимости добавить механизм предварительной подготовки новостей перед отображением в ленте.

Этап 1. Подготовка проектов

Начнем с подготовки структуры проектов. Создадим следующие проекты:

- `NewsAppBackend.Domain` - общие сущности и интерфейсы;
- `NewsAppBackend.Application` - прикладной уровень;
- `NewsAppBackend.Infrastructure` - инфраструктурный уровень;
- `NewsAppBackend.WebApi` - уровень представления.

Выполним создание при помощи команд:

```
dotnet new sln -n NewsAppBackend -o NewsAppBackend
cd NewsAppBackend

dotnet new classlib -n NewsAppBackend.Domain
dotnet new classlib -n NewsAppBackend.Application
dotnet new classlib -n NewsAppBackend.Infrastructure
dotnet new webapi -n NewsAppBackend.WebApi

dotnet sln NewsAppBackend.sln add
NewsAppBackend.Domain/NewsAppBackend.Domain.csproj
dotnet sln NewsAppBackend.sln add
NewsAppBackend.Application/NewsAppBackend.Application.csproj
dotnet sln NewsAppBackend.sln add
NewsAppBackend.Infrastructure/NewsAppBackend.Infrastructure.csproj
dotnet sln NewsAppBackend.sln add
NewsAppBackend.WebApi/NewsAppBackend.WebApi.csproj

dotnet add NewsAppBackend.WebApi/NewsAppBackend.WebApi.csproj reference
NewsAppBackend.Application/NewsAppBackend.Application.csproj
dotnet add NewsAppBackend.WebApi/NewsAppBackend.WebApi.csproj reference
NewsAppBackend.Infrastructure/NewsAppBackend.Infrastructure.csproj

dotnet add NewsAppBackend.Application/NewsAppBackend.Application.csproj reference
NewsAppBackend.Domain/NewsAppBackend.Domain.csproj
dotnet add NewsAppBackend.Application/NewsAppBackend.Application.csproj package
Microsoft.Extensions.DependencyInjection.Abstractions --version 8

dotnet add NewsAppBackend.Infrastructure/NewsAppBackend.Infrastructure.csproj
reference NewsAppBackend.Domain/NewsAppBackend.Domain.csproj
dotnet add NewsAppBackend.Infrastructure/NewsAppBackend.Infrastructure.csproj
```

```
reference NewsAppBackend.Application/NewsAppBackend.Application.csproj
dotnet add NewsAppBackend.Infrastructure/NewsAppBackend.Infrastructure.csproj
package Npgsql.EntityFrameworkCore.PostgreSQL --version 8
dotnet add NewsAppBackend.Infrastructure/NewsAppBackend.Infrastructure.csproj
package Microsoft.EntityFrameworkCore.Design --version 8
dotnet add NewsAppBackend.Infrastructure/NewsAppBackend.Infrastructure.csproj
package Microsoft.Extensions.Hosting.Abstractions --version 8
```

Этап 2. Добавление сущностей

Начнем работу с проекта `NewsAppBackend.Domain`.

Создадим в нем каталог `Entities` и добавим в него класс `FeedItem`.

```
namespace NewsAppBackend.Domain.Entities;

public sealed class FeedItem
{
    public Guid Id { get; }
    public string Title { get; }
    public string Content { get; }
    public DateTimeOffset CreatedAt { get; }

    public FeedItem(Guid id, string title, string content, DateTimeOffset
createdAt)
    {
        Id = id;
        Title = title;
        Content = content;
        CreatedAt = createdAt;

        Validate();
    }

    private void Validate()
    {
        if (string.IsNullOrWhiteSpace(Title))
        {
            throw new ArgumentException("Title is required");
        }

        if (string.IsNullOrWhiteSpace(Content))
        {
            throw new ArgumentException("Content is required");
        }
    }
}
```

После этого в том же каталоге создадим класс `Draft`.

```
namespace NewsAppBackend.Domain.Entities;

public sealed class Draft
{
    public Guid Id { get; }
    public string Title { get; }
    public string Content { get; }
    public DateTimeOffset CreatedAt { get; }
    public FeedItem? FeedItem { get; }

    public Draft(Guid id, string title, string content, DateTimeOffset createdAt,
FeedItem? feedItem)
    {
        Id = id;
        Title = title;
        Content = content;
        CreatedAt = createdAt;
        FeedItem = feedItem;

        Validate();
    }

    public Draft Publish()
    {
        if (FeedItem is not null)
        {
            throw new InvalidOperationException("Draft is already published");
        }

        var feedItem = new FeedItem(Id, Title, Content, CreatedAt);

        return new Draft(Id, Title, Content, CreatedAt, feedItem);
    }

    private void Validate()
    {
        if (string.IsNullOrWhiteSpace(Title))
        {
            throw new ArgumentException("Title is required");
        }

        if (string.IsNullOrWhiteSpace(Content))
        {
            throw new ArgumentException("Content is required");
        }
    }
}
```

Как видим, у нас есть две сущности:

- `Draft` - не опубликованная новость;
- `FeedItem` - опубликованная новость, которая создается на основе неопубликованной новости.

Этап 3. Общие интерфейсы

После добавления сущностей перейдем к прикладному уровню и проекту `NewsAppBackend.Application`.

Обычно реализация CQRS в .NET предполагает наличие следующих интерфейсов:

- `ICommand` - интерфейс для команд;
- `ICommandHandler` - интерфейс для обработчиков команд;
- `IQuery` - интерфейс для запросов;
- `IQueryHandler` - интерфейс для обработчиков запросов.

Существуют библиотеки, которые предоставляют реализацию этих интерфейсов, но мы не будем использовать их в данном примере, а напишем свои собственные интерфейсы.

Начнем с добавления каталога `Common/Abstractions` и в нем интерфейса `ICommand`.

```
namespace NewsAppBackend.Application.Common.Abstractions;

public interface ICommand
{
}

public interface ICommand<out TResult> : ICommand
{
}
```

Как можно видеть, мы объявили два интерфейса - с возвращаемым результатом и без него, так как на практике встречаются и те, и другие.

После этого в том же каталоге создадим интерфейс `ICommandHandler`.

```
namespace NewsAppBackend.Application.Common.Abstractions;

public interface ICommandHandler<in TCommand> where TCommand : ICommand
{
    Task HandleAsync(TCommand command, CancellationToken cancellationToken);
}

public interface ICommandHandler<in TCommand, TResult> where TCommand :
ICommand<TResult>
{
    Task<TResult> HandleAsync(TCommand command, CancellationToken
cancellationToken);
}
```

По аналогии с интерфейсом `ICommand` мы добавили интерфейсы для двух типов обработчиков команд.

С интерфейсом `IQuery` все немного проще, так как он подразумевает возвращение результата всегда:

```
namespace NewsAppBackend.Application.Common.Abstractions;

public interface IQuery<out TResult>
{
}
```

После `IQuery` добавим интерфейс `IQueryHandler`.

```
namespace NewsAppBackend.Application.Common.Abstractions;

public interface IQueryHandler<in TQuery, TResult> where TQuery : IQuery<TResult>
{
    Task<TResult> HandleAsync(TQuery query, CancellationToken cancellationToken);
}
```

Этап 4. Бизнес-логика - Создание новости

После добавления необходимых абстракций перейдем к реализации бизнес-логики. Начнем с создания новости. Для этого нам понадобятся следующие типы:

- `CreateDraftCommand` - команда для создания новости;
- `CreatedDraftDto` - DTO, описывающая добавленную новость;
- `CreateDraftCommandHandler` - обработчик команды для создания новости;
- `ICreateDraftRepository` - интерфейс репозитория для создания новости, так как нам нужно каким-то образом сохранить наши данные.

Начнем с создания каталога для нашей бизнес-операции. В корне проекта `NewsAppBackend.Application` создадим каталог `UseCases` и в нем каталог `CreateDraft`.

Далее в этом каталоге добавляем нашу DTO.

```
namespace NewsAppBackend.Application.UseCases.CreateDraft;

public sealed record CreatedDraftDto(Guid Id, string Title, string Content,
DateTimeOffset CreatedAt);
```

После этого можно определить класс команды.

```
namespace NewsAppBackend.Application.UseCases.CreateDraft;

public sealed record CreateDraftCommand(string Title, string Content) :
ICommand<CreatedDraftDto>;
```

Когда у нас есть команда - настало время ее обработать. Нам понадобятся репозиторий и обработчик команды.

Репозиторий:

```
namespace NewsAppBackend.Application.UseCases.CreateDraft;

public interface ICreateDraftRepository
{
    Task<Draft> CreateAsync(Draft draft, CancellationToken cancellationToken);
}
```

Обработчик команды:

```
namespace NewsAppBackend.Application.UseCases.CreateDraft;

internal sealed class CreateDraftCommandHandler(
    ICreateDraftRepository repository
) : ICommandHandler<CreateDraftCommand, CreatedDraftDto>
{
    public async Task<CreatedDraftDto> HandleAsync(CreateDraftCommand command,
CancellationToken cancellationToken)
    {
        var draft = new Draft(
            Guid.NewGuid(),
            command.Title,
            command.Content,
            createdAt: DateTimeOffset.UtcNow,
            feedItem: null
        );

        var createdDraft = await repository.CreateAsync(draft, cancellationToken);

        return new CreatedDraftDto(createdDraft.Id, createdDraft.Title,
createdDraft.Content, createdDraft.CreatedAt);
    }
}
```

Далее добавим код регистрации нашего обработчика в DI-контейнере. Для этого в корне проекта `NewsAppBackend.Application` создадим класс `ServiceCollectionExtensions` и добавим в него следующий код:

```
namespace NewsAppBackend.Application;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddApplication(this IServiceCollection
services)
```

```
    {
        services.AddScoped<ICommandHandler<CreateDraftCommand, CreatedDraftDto>,
CreateDraftCommandHandler>();

        return services;
    }
}
```

Этап 5. Бизнес-логика - Публикация новости

По аналогии с созданием новости, для реализации операции публикации новости нам понадобятся следующие типы:

- `PublishDraftCommand` - команда для публикации новости;
- `PublishedDraftDto` - DTO, описывающая опубликованную новость;
- `PublishedFeedItemDto` - DTO, описывающая опубликованный элемент новостной ленты;
- `PublishDraftCommandHandler` - обработчик команды для публикации новости;
- `IPublishDraftRepository` - интерфейс репозитория для публикации новости.

Создадим в проекте `NewsAppBackend.Application` каталог `UseCases/PublishDraft` и добавим в него наши DTO.

PublishedFeedItemDto:

```
namespace NewsAppBackend.Application.UseCases.PublishDraft;

public sealed record PublishedFeedItemDto(Guid Id, string Title, string Content,
DateTimeOffset CreatedAt);
```

PublishedDraftDto:

```
namespace NewsAppBackend.Application.UseCases.PublishDraft;

public sealed record PublishedDraftDto(Guid Id, string Title, string Content,
DateTimeOffset CreatedAt, PublishedFeedItemDto FeedItem);
```

После этого определим класс команды.

```
namespace NewsAppBackend.Application.UseCases.PublishDraft;

public sealed record PublishDraftCommand(Guid DraftId) :
ICommand<PublishedDraftDto>;
```

Теперь добавим репозиторий и обработчик команды.

Репозиторий:

```
namespace NewsAppBackend.Application.UseCases.PublishDraft;

public interface IPublishDraftRepository
{
    Task<Draft> GetByIdAsync(Guid id, CancellationToken cancellationToken);
    Task<Draft> UpdateAsync(Draft draft, CancellationToken cancellationToken);
}
```

Обработчик команды:

```
namespace NewsAppBackend.Application.UseCases.PublishDraft;

internal sealed class PublishDraftCommandHandler(
    IPublishDraftRepository repository
) : ICommandHandler<PublishDraftCommand, PublishedDraftDto>
{
    public async Task<PublishedDraftDto> HandleAsync(PublishDraftCommand command,
CancellationToken cancellationToken)
    {
        var draft = await repository.GetByIdAsync(command.DraftId,
cancellationToken);

        var publishedDraft = draft.Publish();

        Debug.Assert(publishedDraft.FeedItem is not null, "Draft is not published
for some reason");

        await repository.UpdateAsync(publishedDraft, cancellationToken);

        var feedItemDto = new PublishedFeedItemDto(
            publishedDraft.FeedItem.Id,
            publishedDraft.FeedItem.Title,
            publishedDraft.FeedItem.Content,
            publishedDraft.FeedItem.CreatedAt
        );

        return new PublishedDraftDto(
            publishedDraft.Id,
            publishedDraft.Title,
            publishedDraft.Content,
            publishedDraft.CreatedAt,
            feedItemDto
        );
    }
}
```

Добавим регистрацию обработчика в DI-контейнер. Для этого дополним метод `AddApplication` в классе `ServiceCollectionExtensions`:

```
public static IServiceCollection AddApplication(this IServiceCollection services)
{
    // ... существующие регистрации ...
    services.AddScoped<ICommandHandler<PublishDraftCommand, PublishedDraftDto>,
PublishDraftCommandHandler>();

    return services;
}
```

Этап 6. Бизнес-логика - Получение новостной ленты

Для реализации операции получения новостной ленты нам понадобятся следующие типы:

- `GetFeedQuery` - запрос для получения новостной ленты;
- `FeedItemDto` - DTO, описывающая элемент новостной ленты;
- `FeedDto` - DTO, описывающая новостную ленту;
- `GetFeedQueryHandler` - обработчик запроса для получения новостной ленты;
- `IGetFeedRepository` - интерфейс репозитория для получения новостной ленты.

Создадим в проекте `NewsAppBackend.Application` каталог `UseCases/GetFeed` и добавим в него наши DTO.

FeedItemDto:

```
namespace NewsAppBackend.Application.UseCases.GetFeed;

public sealed record FeedItemDto(Guid Id, string Title, string Content,
DateTimeOffset CreatedAt);
```

FeedDto:

```
namespace NewsAppBackend.Application.UseCases.GetFeed;

public sealed record FeedDto(IReadOnlyList<FeedItemDto> Items);
```

После этого определим класс запроса.

```
namespace NewsAppBackend.Application.UseCases.GetFeed;

public sealed record GetFeedQuery(int Page, int PageSize) : IQuery<FeedDto>;
```

Теперь добавим репозиторий и обработчик запроса.

Репозиторий:

```
namespace NewsAppBackend.Application.UseCases.GetFeed;

public interface IGetFeedRepository
{
    Task<IEnumerable<FeedItem>> GetFeedItemsAsync(int page, int pageSize,
CancellationToken cancellationToken);
}
```

Обработчик запроса:

```
namespace NewsAppBackend.Application.UseCases.GetFeed;

internal sealed class GetFeedQueryHandler(
    IGetFeedRepository repository
) : IQueryHandler<GetFeedQuery, FeedDto>
{
    public async Task<FeedDto> HandleAsync(GetFeedQuery query, CancellationToken
cancellationToken)
    {
        var feedItems = await repository.GetFeedItemsAsync(query.Page,
query.PageSize, cancellationToken);

        var items = feedItems.Select(item => new FeedItemDto(
            item.Id,
            item.Title,
            item.Content,
            item.CreatedAt
        )).ToList();

        return new FeedDto(items);
    }
}
```

Добавим регистрацию обработчика в DI-контейнер. Для этого дополним метод `AddApplication` в классе `ServiceCollectionExtensions`:

```
public static IServiceCollection AddApplication(this IServiceCollection services)
{
    // ... существующие регистрации ...
    services.AddScoped<IQueryHandler<GetFeedQuery, FeedDto>, GetFeedQueryHandler>
();

    return services;
}
```

Этап 7. Инфраструктура - Настройка ORM

Для реализации CQRS нам нужно настроить два контекста базы данных:

- `ReadWriteDbContext` - для операций записи (команд);
- `ReadOnlyDbContext` - для операций чтения (запросов).

Начнем с создания базовых сущностей для ORM. Создадим в проекте `NewsAppBackend.Infrastructure` каталог `Database/Entities` и добавим в него следующие классы.

FeedItemEntity:

```
namespace NewsAppBackend.Infrastructure.Database.Entities;

internal sealed class FeedItemEntity
{
    public Guid Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public DateTimeOffset CreatedAt { get; set; }
}
```

DraftEntity:

```
namespace NewsAppBackend.Infrastructure.Database.Entities;

internal sealed class DraftEntity
{
    public Guid Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public DateTimeOffset CreatedAt { get; set; }
    public FeedItemEntity? FeedItem { get; set; }
}
```

Теперь создадим базовый класс контекста базы данных. Для этого в каталоге `Database` создадим класс `BaseDbContext`:

```
namespace NewsAppBackend.Infrastructure.Database;

internal abstract class BaseDbContext : DbContext
{
    protected BaseDbContext(DbContextOptions options) : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
```

```
    {
        modelBuilder.Entity<DraftEntity>(entity =>
        {
            entity.ToTable("drafts");

            entity.HasKey(e => e.Id);

            entity.Property(e => e.Id)
                .HasColumnName("id")
                .IsRequired();

            entity.Property(e => e.Title)
                .HasColumnName("title")
                .IsRequired();

            entity.Property(e => e.Content)
                .HasColumnName("content")
                .IsRequired();

            entity.Property(e => e.CreatedAt)
                .HasColumnName("created_at")
                .IsRequired();

            entity.HasOne(e => e.FeedItem)
                .WithOne()
                .HasForeignKey<DraftEntity>("feed_item_id");
        });

        modelBuilder.Entity<FeedItemEntity>(entity =>
        {
            entity.ToTable("feed_items");

            entity.HasKey(e => e.Id);

            entity.Property(e => e.Id)
                .HasColumnName("id")
                .IsRequired();

            entity.Property(e => e.Title)
                .HasColumnName("title")
                .IsRequired();

            entity.Property(e => e.Content)
                .HasColumnName("content")
                .IsRequired();

            entity.Property(e => e.CreatedAt)
                .HasColumnName("created_at")
                .IsRequired();
        });
    }
}
```

После этого создадим контекст для операций записи:

```
namespace NewsAppBackend.Infrastructure.Database;

internal sealed class ReadWriteDbContext : BaseDbContext
{
    public DbSet<DraftEntity> Drafts { get; set; }
    public DbSet<FeedItemEntity> FeedItems { get; set; }

    public ReadWriteDbContext(DbContextOptions<ReadWriteDbContext> options) :
base(options)
    {
    }
}
```

И контекст для операций чтения:

```
namespace NewsAppBackend.Infrastructure.Database;

internal sealed class ReadOnlyDbContext : BaseDbContext
{
    public IQueryable<FeedItemEntity> FeedItems => Set<FeedItemEntity>();

    public ReadOnlyDbContext(DbContextOptions<ReadOnlyDbContext> options) :
base(options)
    {
    }
}
```

Теперь создадим фабрику для контекста записи, чтобы можно было сгенерировать миграции:

```
namespace NewsAppBackend.Infrastructure.Database;

internal sealed class ReadWriteDbContextFactory :
IDesignTimeDbContextFactory<ReadWriteDbContext>
{
    public ReadWriteDbContext CreateDbContext(string[] args)
    {
        var optionsBuilder = new DbContextOptionsBuilder<ReadWriteDbContext>();

optionsBuilder.UseNpgsql("Host=localhost;Database=NewsAppBackend;Username=postgres
;Password=postgres");

        return new ReadWriteDbContext(optionsBuilder.Options);
    }
}
```

Сгенерируем миграции:

```
dotnet ef migrations add InitialCreate --project
NewsAppBackend.Infrastructure/NewsAppBackend.Infrastructure.csproj --context
ReadWriteDbContext
```

Добавим код для применения миграций:

```
namespace NewsAppBackend.Infrastructure.Database;

internal sealed class MigrationRunner : IHostedService
{
    private readonly IServiceScopeFactory _serviceScopeFactory;

    public MigrationRunner(IServiceScopeFactory serviceScopeFactory)
    {
        _serviceScopeFactory = serviceScopeFactory;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        using var scope = _serviceScopeFactory.CreateScope();
        using var context =
scope.ServiceProvider.GetRequiredService<ReadWriteDbContext>();

        context.Database.Migrate();

        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken) =>
Task.CompletedTask;
}
```

Наконец, добавим регистрацию сервисов в DI-контейнер. Для этого в проекте `NewsAppBackend.Infrastructure` создадим класс `ServiceCollectionExtensions`:

```
namespace NewsAppBackend.Infrastructure;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddInfrastructure(this IServiceCollection
services)
    {
        services.AddDbContext<ReadWriteDbContext>((serviceProvider, options) =>
            options.UseNpgsql(serviceProvider.GetRequiredService<IConfiguration>
().GetConnectionString("ReadWrite")));
```

```
        services.AddDbContext<ReadOnlyDbContext>((serviceProvider, options) =>
            options
                .UseNpgsql(serviceProvider.GetRequiredService<IConfiguration>
().GetConnectionString("ReadOnly"))
                .UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking));

        services.AddHostedService<MigrationRunner>();

        return services;
    }
}
```

Этап 8. Инфраструктура - Реализация репозиториев

Перейдем к реализации репозиториев. Реализации будем размещать в каталоге `UseCases` в корне проекта инфраструктурного уровня в соответствующих подкаталогах.

Добавление новости

Добавим каталог `UseCases/CreateDraft` и в нем класс `CreateDraftRepository`:

```
namespace NewsAppBackend.Infrastructure.UseCases.CreateDraft;

internal sealed class CreateDraftRepository : ICreateDraftRepository
{
    private readonly ReadWriteDbContext _context;

    public CreateDraftRepository(ReadWriteDbContext context)
    {
        _context = context;
    }

    public async Task<Draft> CreateAsync(Draft draft, CancellationToken
cancellationToken)
    {
        var entity = new DraftEntity
        {
            Id = draft.Id,
            Title = draft.Title,
            Content = draft.Content,
            CreatedAt = draft.CreatedAt
        };

        _context.Drafts.Add(entity);
        await _context.SaveChangesAsync(cancellationToken);

        return draft;
    }
}
```

Публикация новости

Добавим каталог `UseCases/PublishDraft` и в нем класс `PublishDraftRepository`:

```
namespace NewsAppBackend.Infrastructure.UseCases.PublishDraft;

internal sealed class PublishDraftRepository : IPublishDraftRepository
{
    private readonly ReadWriteDbContext _context;

    public PublishDraftRepository(ReadWriteDbContext context)
    {
        _context = context;
    }

    public async Task<Draft> GetByIdAsync(Guid id, CancellationToken
cancellationToken)
    {
        var entity = await _context.Drafts
            .Include(d => d.FeedItem)
            .SingleOrDefaultAsync(d => d.Id == id, cancellationToken);

        if (entity is null)
        {
            throw new InvalidOperationException($"Draft with id {id} not found");
        }

        return new Draft(
            entity.Id,
            entity.Title,
            entity.Content,
            entity.CreatedAt,
            entity.FeedItem is null ? null : new FeedItem(
                entity.FeedItem.Id,
                entity.FeedItem.Title,
                entity.FeedItem.Content,
                entity.FeedItem.CreatedAt
            )
        );
    }

    public async Task<Draft> UpdateAsync(Draft draft, CancellationToken
cancellationToken)
    {
        var entity = await _context.Drafts
            .Include(d => d.FeedItem)
            .SingleOrDefaultAsync(d => d.Id == draft.Id, cancellationToken);

        if (entity is null)
        {
            throw new InvalidOperationException($"Draft with id {draft.Id} not
found");
```

```
        }

        if (draft.FeedItem is not null)
        {
            var feedItemEntity = new FeedItemEntity
            {
                Id = draft.FeedItem.Id,
                Title = draft.FeedItem.Title,
                Content = draft.FeedItem.Content,
                CreatedAt = draft.FeedItem.CreatedAt
            };

            _context.FeedItems.Add(feedItemEntity);
            entity.FeedItem = feedItemEntity;
        }

        await _context.SaveChangesAsync(cancellationToken);

        return draft;
    }
}
```

Получение новостной ленты

Добавим каталог `UseCases/GetFeed` и в нем класс `GetFeedRepository`:

```
namespace NewsAppBackend.Infrastructure.UseCases.GetFeed;

internal sealed class GetFeedRepository : IGetFeedRepository
{
    private readonly ReadOnlyDbContext _context;

    public GetFeedRepository(ReadOnlyDbContext context)
    {
        _context = context;
    }

    public async Task<IEnumerable<FeedItem>> GetFeedItemsAsync(int page, int
pageSize, CancellationToken cancellationToken)
    {
        var entities = await _context.FeedItems
            .OrderByDescending(f => f.CreatedAt)
            .Skip((page - 1) * pageSize)
            .Take(pageSize)
            .ToListAsync(cancellationToken);

        return entities.Select(entity => new FeedItem(
            entity.Id,
            entity.Title,
            entity.Content,
            entity.CreatedAt
```

```
        ));
    }
}
```

Наконец, добавим регистрацию репозиториев в DI-контейнер. Для этого дополним метод `AddInfrastructure` в классе `ServiceCollectionExtensions`:

```
public static IServiceCollection AddInfrastructure(this IServiceCollection
services, IConfiguration configuration)
{
    // ... существующие регистрации ...

    services.AddScoped<ICreateDraftRepository, CreateDraftRepository>();
    services.AddScoped<IPublishDraftRepository, PublishDraftRepository>();
    services.AddScoped<IGetFeedRepository, GetFeedRepository>();

    return services;
}
```

Этап 9. Слой представления - эндпоинты

Для организации эндпоинтов с использованием Minimal API создадим каталог `Endpoints` в проекте `NewsAppBackend.WebApi` и добавим в него следующие файлы.

CreateDraft.cs:

```
namespace NewsAppBackend.WebApi.Endpoints;

public static class CreateDraft
{
    public static void MapCreateDraft(this IEndpointRouteBuilder app)
    {
        app.MapPost("/api/v1/drafts", async (
            CreateDraftCommand command,
            ICommandHandler<CreateDraftCommand, CreatedDraftDto> handler,
            CancellationToken cancellationToken) =>
        {
            var result = await handler.HandleAsync(command, cancellationToken);
            return Results.Ok(result);
        })
        .WithName("CreateDraft")
        .WithOpenApi();
    }
}
```

PublishDraft.cs:

```
namespace NewsAppBackend.WebApi.Endpoints;

public static class PublishDraft
{
    public static void MapPublishDraft(this IEndpointRouteBuilder app)
    {
        app.MapPost("/api/v1/drafts/{draftId}/publish", async (
            Guid draftId,
            ICommandHandler<PublishDraftCommand, PublishedDraftDto> handler,
            CancellationToken cancellationToken) =>
        {
            var command = new PublishDraftCommand(draftId);
            var result = await handler.HandleAsync(command, cancellationToken);
            return Results.Ok(result);
        })
        .WithName("PublishDraft")
        .WithOpenApi();
    }
}
```

GetFeed.cs:

```
namespace NewsAppBackend.WebApi.Endpoints;

public static class GetFeed
{
    public static void MapGetFeed(this IEndpointRouteBuilder app)
    {
        app.MapGet("/api/v1/feed", async (
            int page,
            int pageSize,
            IQueryHandler<GetFeedQuery, FeedDto> handler,
            CancellationToken cancellationToken) =>
        {
            var query = new GetFeedQuery(page, pageSize);
            var result = await handler.HandleAsync(query, cancellationToken);
            return Results.Ok(result);
        })
        .WithName("GetFeed")
        .WithOpenApi();
    }
}
```

После создания эндпоинтов нужно зарегистрировать их в `Program.cs`. Для этого добавим следующий код перед `app.Run();`:

```
app.MapCreateDraft();
app.MapPublishDraft();
```

```
app.MapGetFeed();
```

Теперь у нас есть три эндпоинта:

1. `POST /api/v1/drafts` - создание черновика новости;
2. `POST /api/v1/drafts/{draftId}/publish` - публикация черновика новости;
3. `GET /api/v1/feed` - получение новостной ленты.

Каждый эндпоинт использует соответствующий обработчик команды или запроса, который мы реализовали ранее.

Этап 10. Регистрация сервисов

Наконец, добавим регистрацию всех наших слоев в `Program.cs`. Для этого перед `var app = builder.Build();` добавим следующий код:

```
builder.Services.AddApplication();
builder.Services.AddInfrastructure();
```

Этап 11. Создание Dockerfile

Теперь, когда код проекта дописан, можно создать Dockerfile, который будет использоваться для сборки и запуска нашего приложения.

Создадим файл `Dockerfile` в корне решения и добавим в него следующий код:

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src

COPY ["NewsAppBackend.Domain/NewsAppBackend.Domain.csproj",
"NewsAppBackend.Domain/"]
COPY ["NewsAppBackend.Application/NewsAppBackend.Application.csproj",
"NewsAppBackend.Application/"]
COPY ["NewsAppBackend.Infrastructure/NewsAppBackend.Infrastructure.csproj",
"NewsAppBackend.Infrastructure/"]
COPY ["NewsAppBackend.WebApi/NewsAppBackend.WebApi.csproj",
"NewsAppBackend.WebApi/"]

RUN dotnet restore "NewsAppBackend.WebApi/NewsAppBackend.WebApi.csproj"

COPY . .

WORKDIR "/src/NewsAppBackend.WebApi"

RUN dotnet publish -c Release -o /out

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS publish
WORKDIR /app
```

```
COPY --from=build /out .
ENTRYPOINT ["dotnet", "NewsAppBackend.WebApi.dll"]
```

Также добавим файл `.dockerignore` в корне решения и добавим в него следующий код:

```
bin/
obj/
*.user
*.suo
*.sln.ide
*.db
*.log
*.pdb
*.mdb
*.cache
*.vs/
.vscode/
.DS_Store
Thumbs.db
```

Этап 12. Создание docker-compose.yml

Теперь, когда у нас есть Dockerfile, можно создать docker-compose.yml файл для запуска нашего приложения.

Создадим файл `docker-compose.yml` в корне решения и добавим в него следующий код:

```
x-postgres-common:
  &postgres-common
  image: postgres:15
  restart: always
  healthcheck:
    test: 'pg_isready -U postgres --dbname=postgres'
    interval: 10s
    timeout: 5s
    retries: 5

services:
    news-app-backend:
        build:
            context: .
            dockerfile: Dockerfile
        ports:
            - "8080:8080"
        environment:
            - ASPNETCORE_ENVIRONMENT=Development
            - CONNECTIONSTRINGS__READWRITE=Host=postgres-
master;Database=NewsAppBackend;Username=postgres;Password=postgres
            - CONNECTIONSTRINGS__READONLY=Host=postgres-
```

```
readonly;Database=NewsAppBackend;Username=postgres;Password=postgres;
        depends_on:
            postgres-master:
                condition: service_healthy
            postgres-readonly:
                condition: service_started
    postgres-master:
        <<: *postgres-common
        ports:
            - "5432:5432"
        environment:
            POSTGRES_USER: postgres
            POSTGRES_PASSWORD: postgres
            POSTGRES_DB: NewsAppBackend
            POSTGRES_HOST_AUTH_METHOD: "scram-sha-256\nhost replication all
0.0.0.0/0 md5"
            POSTGRES_INITDB_ARGS: "--auth-host=scram-sha-256"
        command: |
            postgres
            -c wal_level=replica
            -c hot_standby=on
            -c max_wal_senders=10
            -c max_replication_slots=10
            -c hot_standby_feedback=on
        volumes:
            - postgres-master-data:/var/lib/postgresql/data
            - ./init-db.sql:/docker-entrypoint-initdb.d/00-init-db.sql
    postgres-readonly:
        <<: *postgres-common
        ports:
            - "5433:5432"
        environment:
            PGUSER: replicator
            PGPASSWORD: replicator_password
        user: postgres
        command: |
            bash -c "
            if [ ! -f /var/lib/postgresql/data/postgresql.conf ]; then
                until pg_basebackup --pgdata=/var/lib/postgresql/data -R --
slot=replication_slot --host=postgres-master --port=5432
                do
                    echo 'Waiting for primary to connect...'
                    sleep 1s
                done
            fi
            echo 'Backup done, starting replica...'
            chmod 0700 /var/lib/postgresql/data
            postgres
            "
        depends_on:
            postgres-master:
                condition: service_healthy
volumes:
    postgres-master-data:
```

Здесь мы настраиваем запуск нашего приложения, а также двух экземпляров PostgreSQL для демонстрации репликации данных.

Также нам понадобится файл `init-db.sql` для дополнительной инициализации базы данных. Создадим его в корне решения и добавим в него следующий код:

```
CREATE USER replicator WITH REPLICATION ENCRYPTED PASSWORD 'replicator_password';
SELECT pg_create_physical_replication_slot('replication_slot');
```

Теперь можно запускать наше приложение с помощью docker-compose:

```
docker-compose up -d
```

Когда приложение будет запущено, можно перейти в браузере по адресу `http://localhost:8080/swagger/` и проверить работоспособность нашего приложения.