A. 🧠 Как работает DDD-архитектура в целом?

Представь “пирог”:

```
[ Presentation / WebApi ]  <-- контроллеры, UI
[ Application ]            <-- команды, запросы, бизнес-процессы
[ Domain ]                 <-- правила предметной области
[ Infrastructure ]         <-- EF Core, DB, файлы, внешние сервисы
```

Domain ничего не знает о других слоях. Application знает о Domain. WebApi знает о Application. Infrastructure подключается «снизу» и подставляет реализацию интерфейсов.

🧱 1. DOMAIN (предметная область)

Что здесь?

✔ Entities ✔ Value Objects ✔ Domain Methods (инварианты) ✔ Никаких DTO ✔ Никакого EF Core ✔ Никакой инфраструктуры

Domain отвечает на вопрос:

> "Какие правила существуют в бизнесе?"

✨ Пример Domain Entity

```
public class Material : BaseEntity
{
    private Material() { } // EF protected

    public Material(string name, string unit)
    {
        Rename(name);
        SetUnit(unit);
        IsActive = true;
    }

    public string Name { get; private set; } = string.Empty;
    public string Unit { get; private set; } = string.Empty;
    public bool IsActive { get; private set; }

    public void Rename(string name)
    {
```

```
        if (string.IsNullOrWhiteSpace(name))
            throw new DomainException("Name required.");
        Name = name.Trim();
    }

    public void SetUnit(string unit)
    {
        if (string.IsNullOrWhiteSpace(unit))
            throw new DomainException("Unit required.");
        Unit = unit.Trim();
    }

    public void Deactivate()
    {
        if (!IsActive)
            throw new DomainException("Already inactive.");
        IsActive = false;
    }
}
```

Это — **сущность**, бизнес-правила внутри которой:

- нельзя создать без имени
- нельзя поменять имя на пустое
- нельзя дважды деактивировать

🧩 2. APPLICATION

На этом слое:

✔ Команды и Queries ✔ Handlers ✔ DTOs ✔ Validators ✔ Интерфейсы: `IApplicationDbContext`, `IFileStorage` ✔ Orchestration (взаимодействие между Domain + DB)

Application отвечает на вопрос:

> "Что пользователь хочет сделать?"

И вызывает Domain, чтобы добиться результата.

🎯 Пример DTO (для WebApi)

```
public class MaterialDto
{
    public Guid Id { get; set; }
    public string Name { get; set; } = "";
```

```
    public string Unit { get; set; } = "";
    public bool IsActive { get; set; }
}
```

DTO — это **данные для API/UI**, не доменная модель.

📬 Пример Команды

```
public record CreateMaterialCommand(string Name, string Unit)
    : IRequest<MaterialDto>;
```

🧪 Пример Validator

```
public class CreateMaterialCommandValidator
    : AbstractValidator<CreateMaterialCommand>
{
    public CreateMaterialCommandValidator()
    {
        RuleFor(x => x.Name).NotEmpty();
        RuleFor(x => x.Unit).NotEmpty();
    }
}
```

Application проверяет входные данные. Domain — бизнес-правила.

Они **дополняют** друг друга.

⚙️ Пример Handler

```
public class CreateMaterialCommandHandler
    : IRequestHandler<CreateMaterialCommand, MaterialDto>
{
    private readonly IApplicationDbContext _db;

    public CreateMaterialCommandHandler(IApplicationDbContext db)
    {
        _db = db;
    }

    public async Task<MaterialDto> Handle(
        CreateMaterialCommand cmd,
        CancellationToken ct)
    {
```

```
        // 1) создать доменную сущность
        var material = new Material(cmd.Name, cmd.Unit);

        // 2) сохранить через интерфейс контекста
        _db.Materials.Add(material);
        await _db.SaveChangesAsync(ct);

        // 3) вернуть DTO
        return new MaterialDto
        {
            Id = material.Id,
            Name = material.Name,
            Unit = material.Unit,
            IsActive = material.IsActive
        };
    }
}
```

Что здесь происходит?

1. Команда приходит в Handler
2. Handler вызывает **доменные методы** (`new Material` , `Rename` , `Deactivate`)
3. Handler сохраняет изменения через `IApplicationDbContext`
4. Handler возвращает DTO для WebApi

🧱 3. INFRASTRUCTURE слой

Тут находятся:

✔ EF Core DbContext ✔ Конфигурации Fluent API ✔ Реализации интерфейсов Application слоя ✔ Миграции ✔ Реальные сервисы (хранение файлов, email, JWT, hashing)

Например:

```
public class ApplicationDbContext : DbContext, IApplicationDbContext
{
    public DbSet<Material> Materials => Set<Material>();

    public async Task<int> SaveChangesAsync(CancellationToken ct)
        => await base.SaveChangesAsync(ct);
}
```

Infrastructure знает о Domain и реализует интерфейсы для Application

Но Domain — ничего не знает ни о чем “снаружи”.

🌐 4. WebApi слой (Presentation)

Задача: получить HTTP-запрос → отправить его в Application → вернуть ответ.

🌎 Пример контроллера

```
[ApiController]
[Route("api/materials")]
public class MaterialsController : ControllerBase
{
    private readonly IMediator _mediator;

    public MaterialsController(IMediator mediator)
    {
        _mediator = mediator;
    }

    [HttpPost]
    public async Task<ActionResult<MaterialDto>> Create(CreateMaterialCommand cmd)
    {
        var result = await _mediator.Send(cmd);
        return Ok(result);
    }
}
```

Контроллер:

- **не знает Domain**
- **не знает EF Core**
- просто пересылает запрос → Application слой

🔗 Как слои взаимодействуют (простая схема)

```
[ WebApi ]
    ↓ отправляет command/query
[ Application ]
    ↓ вызывает domain методы
[ Domain ]
    ↓ проверяет правила
[ Application ]
    ↓ вызывает интерфейсы БД (IApplicationDbContext)
[ Infrastructure ]
    ↓ сохраняет в Postgres (EF Core)
```

А потом назад:

Postgres → Infrastructure → Application → DTO → WebApi → UI

🧩 Мини-диаграмма

```
User → WebApi → Mediator → Handler → Domain Entity
                                        ↓
                                  IApplicationDbContext
                                        ↓
                                   EF Core / DB
```

🎉 Резюме:

Domain

- сущности
- инварианты
- методы вроде `Deactivate()`, `Rename()`
- **никакого EF Core, DTO, HTTP**

Application

- команды
- queries
- handlers
- validators
- DTO
- orchestration

Infrastructure

- реализация EF
- DbContext
- репозитории
- внешние сервисы

Presentation (WebApi, MAUI)

- контроллеры
- ViewModels
- вызывают Application

👣 B. Путь отправки данных в БД и получение ответа

Ниже — **предельно ясный "сквозной" пример**: как идет полный путь запроса от **WebApi → Application → Domain → DB → обратно**.

Возьмём понятный кейс:

> **Создать материал** POST /api/materials

🧩 1. КЛИЕНТ → WEBAPI

MAUI или Postman отправляет:

```
POST /api/materials
Content-Type: application/json

{
  "name": "Steel Pipe",
  "unit": "m"
}
```

🌐 2. WEBAPI (Presentation layer)

Контроллер просто получает JSON и отправляет команду в Application слой через MediatR.

```
[ApiController]
[Route("api/materials")]
public class MaterialsController : ControllerBase
{
    private readonly IMediator _mediator;

    public MaterialsController(IMediator mediator)
    {
        _mediator = mediator;
    }
```

```
    [HttpPost]
    public async Task<ActionResult<MaterialDto>> Create(CreateMaterialCommand command)
    {
        var dto = await _mediator.Send(command);
        return Ok(dto);
    }
}
```

❗ Контроллер **ничего не знает** про БД, доменные правила, EF Core.

Он — просто "почтальон".

🎁 3. APPLICATION получает команду

Команда — обычный **record**:

```
public record CreateMaterialCommand(string Name, string Unit)
    : IRequest<MaterialDto>;
```

Valdiator проверяет вход:

```
public class CreateMaterialCommandValidator
    : AbstractValidator<CreateMaterialCommand>
{
    public CreateMaterialCommandValidator()
    {
        RuleFor(x => x.Name).NotEmpty();
        RuleFor(x => x.Unit).NotEmpty();
    }
}
```

Всё OK → MediatR вызывает Handler.

⚙️ 4. APPLICATION Handler → создаёт доменную сущность

```
public class CreateMaterialCommandHandler
    : IRequestHandler<CreateMaterialCommand, MaterialDto>
{
    private readonly IApplicationDbContext _db;

    public CreateMaterialCommandHandler(IApplicationDbContext db)
    {
```

```
        _db = db;
    }

    public async Task<MaterialDto> Handle(CreateMaterialCommand cmd, CancellationToken ct)
    {
        // 1. Создаём ДОМЕННУЮ сущность
        var material = new Material(cmd.Name, cmd.Unit);

        // 2. Добавляем в контекст
        _db.Materials.Add(material);

        // 3. Сохраняем в БД
        await _db.SaveChangesAsync(ct);

        // 4. Возвращаем DTO клиенту
        return new MaterialDto
        {
            Id = material.Id,
            Name = material.Name,
            Unit = material.Unit,
            IsActive = material.IsActive
        };
    }
}
```

Handler:

- вызывает **доменные методы**
- вызывает **инфраструктуру** через интерфейс `IApplicationDbContext`
- формирует DTO

🧠 5. DOMAIN — правила бизнеса

Доменная сущность проверяет правила:

```
public class Material : BaseEntity
{
    public Material(string name, string unit)
    {
        Rename(name);
        SetUnit(unit);
        IsActive = true;
    }

    public string Name { get; private set; }
    public string Unit { get; private set; }
    public bool IsActive { get; private set; }
```

```
    public void Rename(string name)
    {
        if (string.IsNullOrWhiteSpace(name))
            throw new DomainException("Name required.");

        Name = name.Trim();
    }

    public void SetUnit(string unit)
    {
        if (string.IsNullOrWhiteSpace(unit))
            throw new DomainException("Unit required.");

        Unit = unit.Trim();
    }
}
```

Domain:

- не знает о БД
- не знает о Web
- знает только **правила предметной области**

🗄️ 6. INFRASTRUCTURE → DB (EF Core)

В ApplicationDbContext:

```
public class ApplicationDbContext
    : DbContext, IApplicationDbContext
{
    public DbSet<Material> Materials => Set<Material>();

    public Task<int> SaveChangesAsync(CancellationToken ct)
        => base.SaveChangesAsync(ct);
}
```

Когда Handler вызвал:

```
_db.Materials.Add(material);
await _db.SaveChangesAsync();
```

EF Core:

- формирует INSERT SQL
- отправляет в PostgreSQL

- получает Id
- обновляет сущность

📤 7. Application → WebApi → Клиент

Handler создаёт DTO:

```
return new MaterialDto { ... }
```

Контроллер возвращает JSON клиенту:

```
HTTP 200 OK
{
  "id": "a3f...b12",
  "name": "Steel Pipe",
  "unit": "m",
  "isActive": true
}
```

MAUI отобразит на странице таблицы материалов.

🔗 Короткая схема всего процесса

```
[MAUI / Client]
        |
        ▼
[WebApi Controller]
        |  Send(command)
        ▼
[Application Handler]
        |
        |--> Validator
        |
        |--> Domain Entity (правила)
        |
        |--> IApplicationDbContext
        |       ↓
        |    [Infrastructure] → EF INSERT → PostgreSQL
        |
        ▼
     DTO result
        ▼
[WebApi → JSON]
```

```
        ▼
    [MAUI UI updates]
```

✅ C. Гайд по Infrastructure

Полное объяснение, как Infrastructure работает в DDD-проекте. Простой, но глубокий.

🧱 Что такое слой Infrastructure в DDD?

Infrastructure — это слой, который отвечает за:

Задача	Что внутри
Доступ к БД	EF Core, DbContext, конфигурации, миграции
Хранение файлов	IFileStorage, LocalFileStorage
Аутентификация/авторизация	JWT, Identity
Отправка email/SMS (если есть)	почтовые сервисы
Интеграции	HTTP clients, внешние API
Реализация интерфейсов Application	репозитории, UoW

Он НЕ содержит бизнес-логики.

🗄️ 1. DbContext (EF Core)

```
public class ApplicationDbContext : DbContext, IApplicationDbContext
{
    public DbSet<Material> Materials => Set<Material>();
    public DbSet<Employee> Employees => Set<Employee>();
    ...
}
```

Важное правило:

DbContext = отражение ERD + Domain (1:1). Ни одного лишнего класса.

🧬 2. Конфигурации сущностей (Fluent API)

Каждая Domain сущность имеет **отдельный Configuration-класс**:

```
Persistence/
    Configurations/
        MaterialsConfiguration.cs
        EmployeesConfiguration.cs
        …
```

Пример:

```
public class MaterialConfiguration : IEntityTypeConfiguration<Material>
{
    public void Configure(EntityTypeBuilder<Material> builder)
    {
        builder.ToTable("Materials");

        builder.HasKey(x => x.Id);

        builder.Property(x => x.Name)
            .IsRequired()
            .HasMaxLength(Material.NameMaxLength);

        builder.HasOne(x => x.Type)
            .WithMany()
            .HasForeignKey(x => x.TypeId);
    }
}
```

📦 3. Репозитории (Domain-Driven Persistence)

Слой Application определяет интерфейс:

```
public interface IRepository<T>
{
    Task<T?> GetByIdAsync(Guid id);
    Task AddAsync(T entity);
    Task<List<T>> ListAsync(ISpecification<T> spec);
    void Remove(T entity);
}
```

Infrastructure даёт реализацию:

```
public class EfRepository<T> : IRepository<T> where T : class
{
    private readonly ApplicationDbContext _db;

    public async Task<T?> GetByIdAsync(Guid id)
        => await _db.Set<T>().FindAsync(id);

    public async Task AddAsync(T entity)
        => await _db.Set<T>().AddAsync(entity);

    public void Remove(T entity)
        => _db.Set<T>().Remove(entity);
}
```

🔍 4. Specification Pattern

Используется для сложных запросов:

```
public class MaterialsByTypeSpec : Specification<Material>
{
    public MaterialsByTypeSpec(Guid typeId)
    {
        Query
            .Where(x => x.TypeId == typeId)
            .OrderBy(x => x.Name);
    }
}
```

🔄 5. Unit Of Work

Работает поверх DbContext:

```
public class UnitOfWork : IUnitOfWork
{
    private readonly ApplicationDbContext _db;

    public async Task<int> SaveChangesAsync()
        => await _db.SaveChangesAsync();
}
```

🪣 6. File Storage (пример)

Интерфейс:

```
public interface IFileStorage
{
    Task<FileResource> SaveAsync(Stream file, string fileName);
    Task<Stream> GetAsync(Guid id);
    Task DeleteAsync(Guid id);
}
```

Реализация:

```
public class LocalFileStorage : IFileStorage
{
    private readonly string _root = "storage";

    public async Task<FileResource> SaveAsync(Stream file, string fileName)
    {
        var id = Guid.NewGuid();
        var path = Path.Combine(_root, id.ToString());

        using var fs = File.Create(path);
        await file.CopyToAsync(fs);

        return new FileResource(id, fileName, path);
    }
}
```

🧪 7. Тестирование Infrastructure

Тесты должны включать:

- репозитории (CRUD)
- спецификации (фильтры / include)
- soft-delete
- аудитирование
- конфигурации EF Core
- работу DbContext с InMemory provider
- работу InitialDataSeeder

Пример теста:

```
[Fact]
public async Task Add_Material_Works()
{
    var db = TestDbFactory.Create();
    var repo = new EfRepository<Material>(db);

    var material = new Material(...);

    await repo.AddAsync(material);
    await db.SaveChangesAsync();

    var loaded = await repo.GetByIdAsync(material.Id);

    Assert.NotNull(loaded);
}
```

🧱 8. Как Infrastructure связывается с Application

Application определяет интерфейсы:

✔ `IApplicationDbContext` ✔ `IRepository<T>` ✔ `IUnitOfWork` ✔ `IFileStorage` ✔ сервисы домена (через интерфейсы)

Infrastructure даёт **реализацию**, регистрирует в DI:

```
services.AddDbContext<ApplicationDbContext>();
services.AddScoped<IApplicationDbContext>(provider => provider.GetRequiredService<ApplicationD
services.AddScoped(typeof(IRepository<>), typeof(EfRepository<>));
services.AddScoped<IUnitOfWork, UnitOfWork>();
services.AddSingleton<IFileStorage, LocalFileStorage>();
```

🧬 9. Резюме: Из чего состоит Infrastructure

Компонент	Назначение
DbContext	Работа с БД
Конфигурации EF	Схема + связи
Миграции	Синхронизация БД
Репозитории	Работа с агрегатами

Компонент	Назначение
Спецификации	Query logic
Unit of Work	Транзакции
File storage	Хранение файлов
Seeders	Начальные данные
DI extensions	Подключение к WebApi
Интеграции	Email, HTTP, etc.
Тесты Infrastructure	Проверка корректности
