

Basis-Template für Blazor-Apps

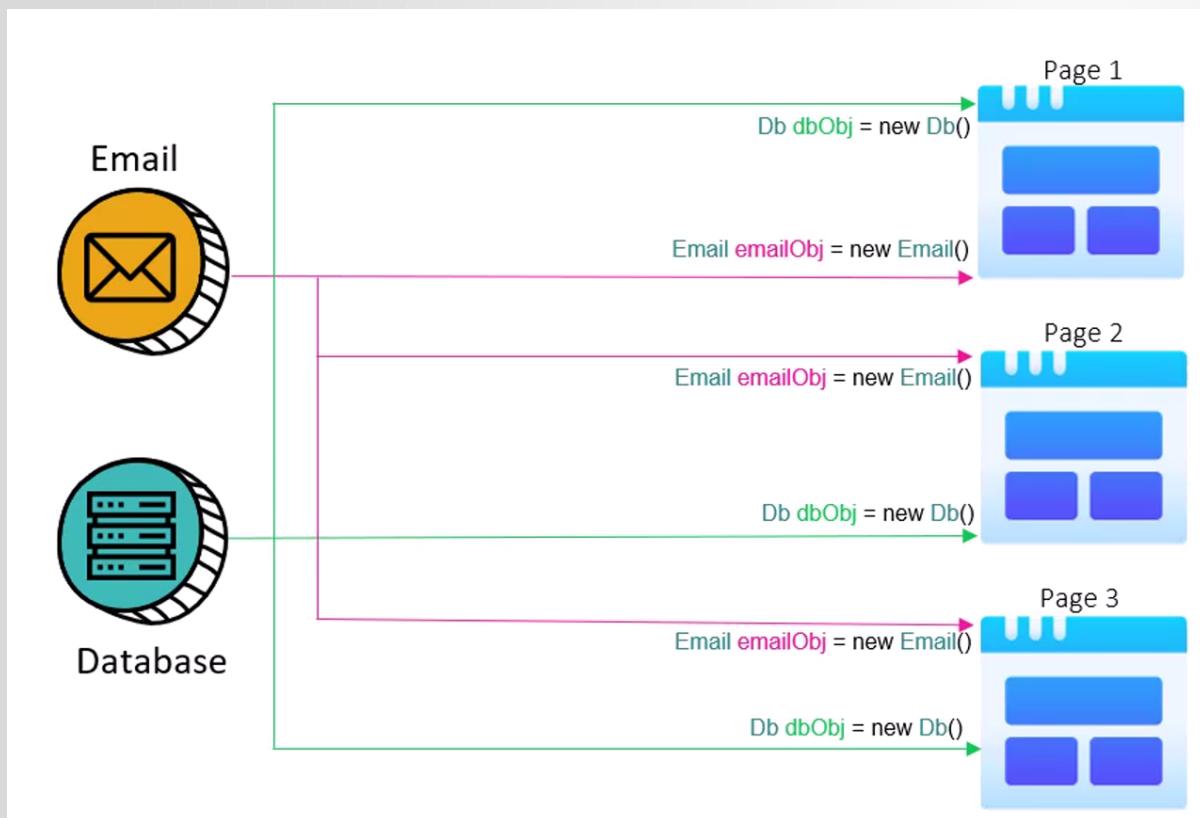
- Basierend auf anerkannten Pattern
 - Clean-Architecture
 - UnitOfWork/Repository-Pattern
 - CQRS und Mediator-Pattern
- OO-Basisprogrammiertechniken
 - SOLID
 - DRY
 - Don't repeat yourself
 - Verwendung Tests zur Qualitätssicherung

SOLID – wartbar und sauber

- S - Single Responsibility Principle (SRP)
 - Eine Klasse sollte nur eine einzige Verantwortung haben
- O - Open/Closed Principle (OCP)
 - Neue Funktionalität soll durch **Erweiterung**, nicht durch Änderung des bestehenden Codes umgesetzt werden.
- L - Liskov Substitution Principle (LSP)
 - Objekte einer abgeleiteten Klasse sollen sich so verhalten, dass sie überall als Objekte der Basisklasse einsetzbar sind.
- I - Interface Segregation Principle (ISP)
 - Viele kleine, spezifische Schnittstellen sind besser als eine große, allgemeine.
- D - Dependency Inversion Principle (DIP)
 - Abhängigkeiten sollen über Abstraktionen (Interfaces) definiert werden, nicht über konkrete Implementierungen.

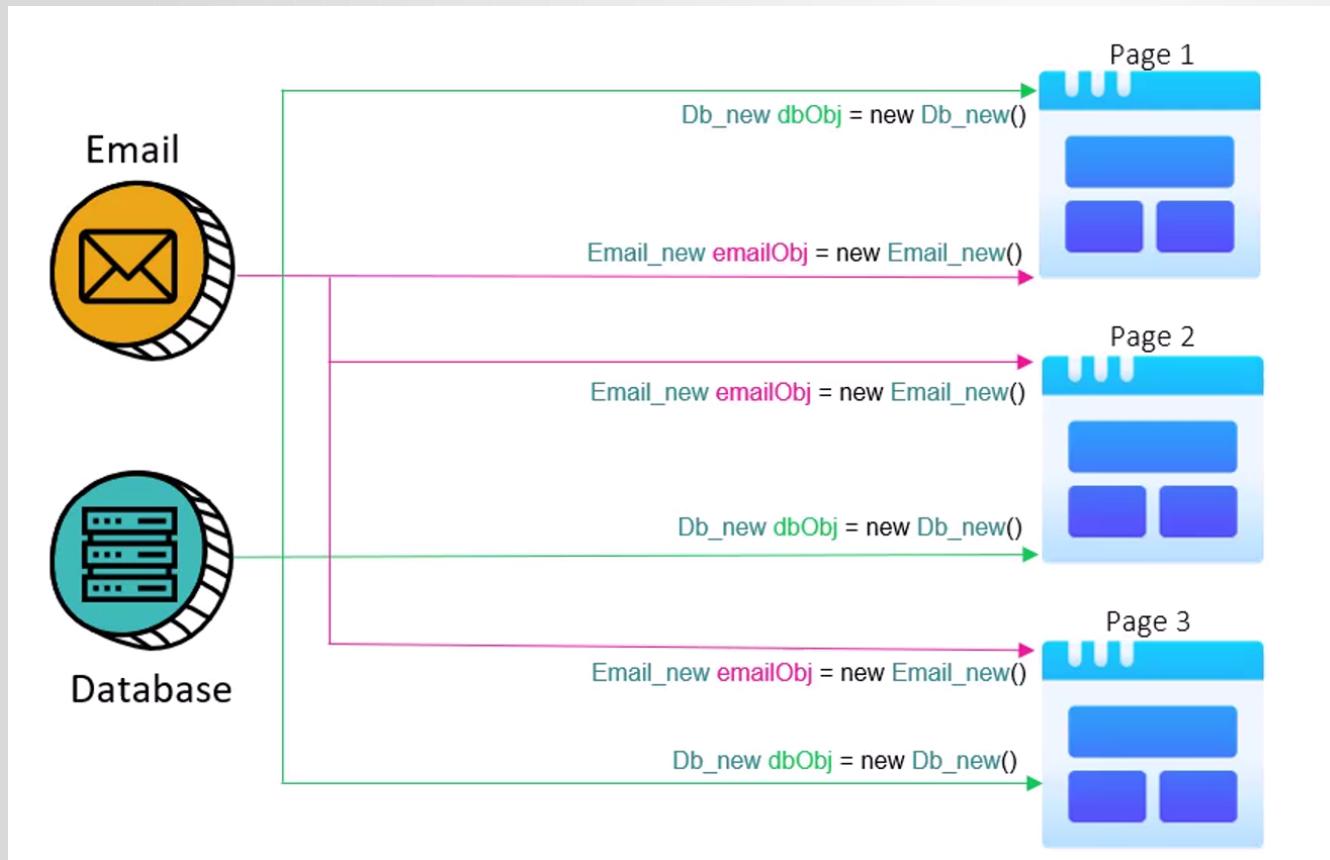
Dependency Injection

- Ohne DI → Services werden bei Verwendung erzeugt
 - Abhängigkeit von konkreter Implementierung



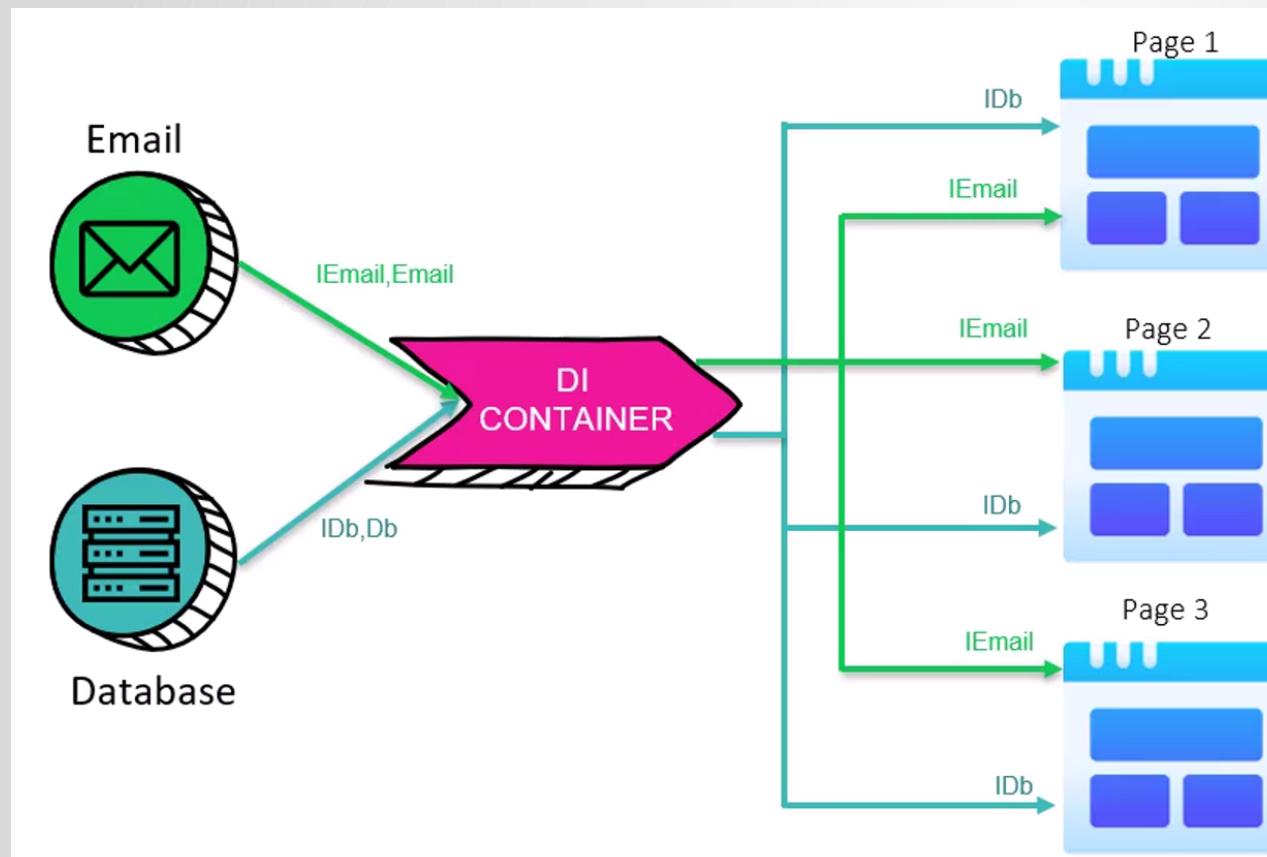
Dependency Injection

- Änderung der Service-Klasse → Änderung des Codes nötig



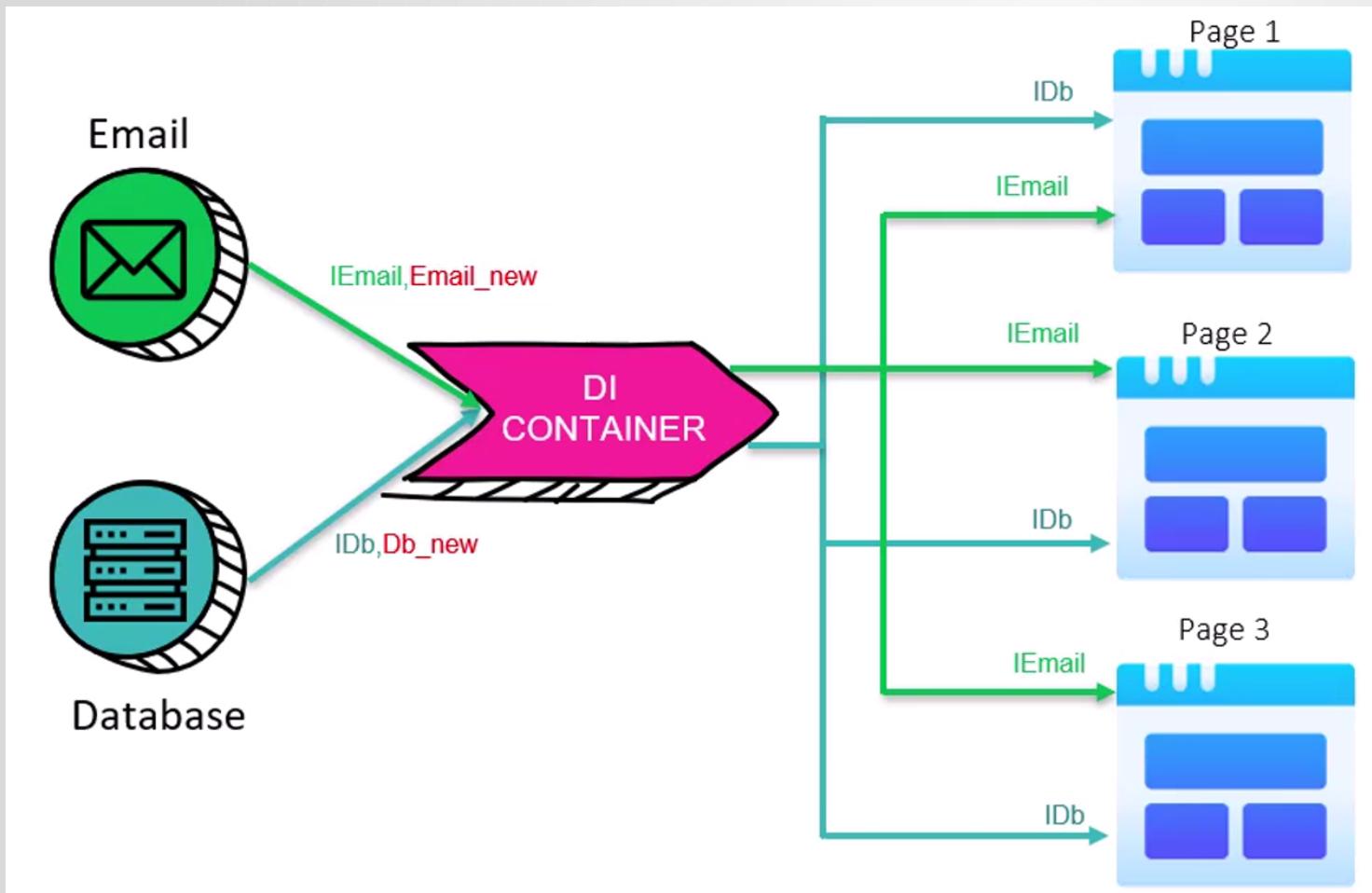
Dependency Injection

- Injektion des erzeugten Services
 - Basis sind meist entsprechende „Verträge“

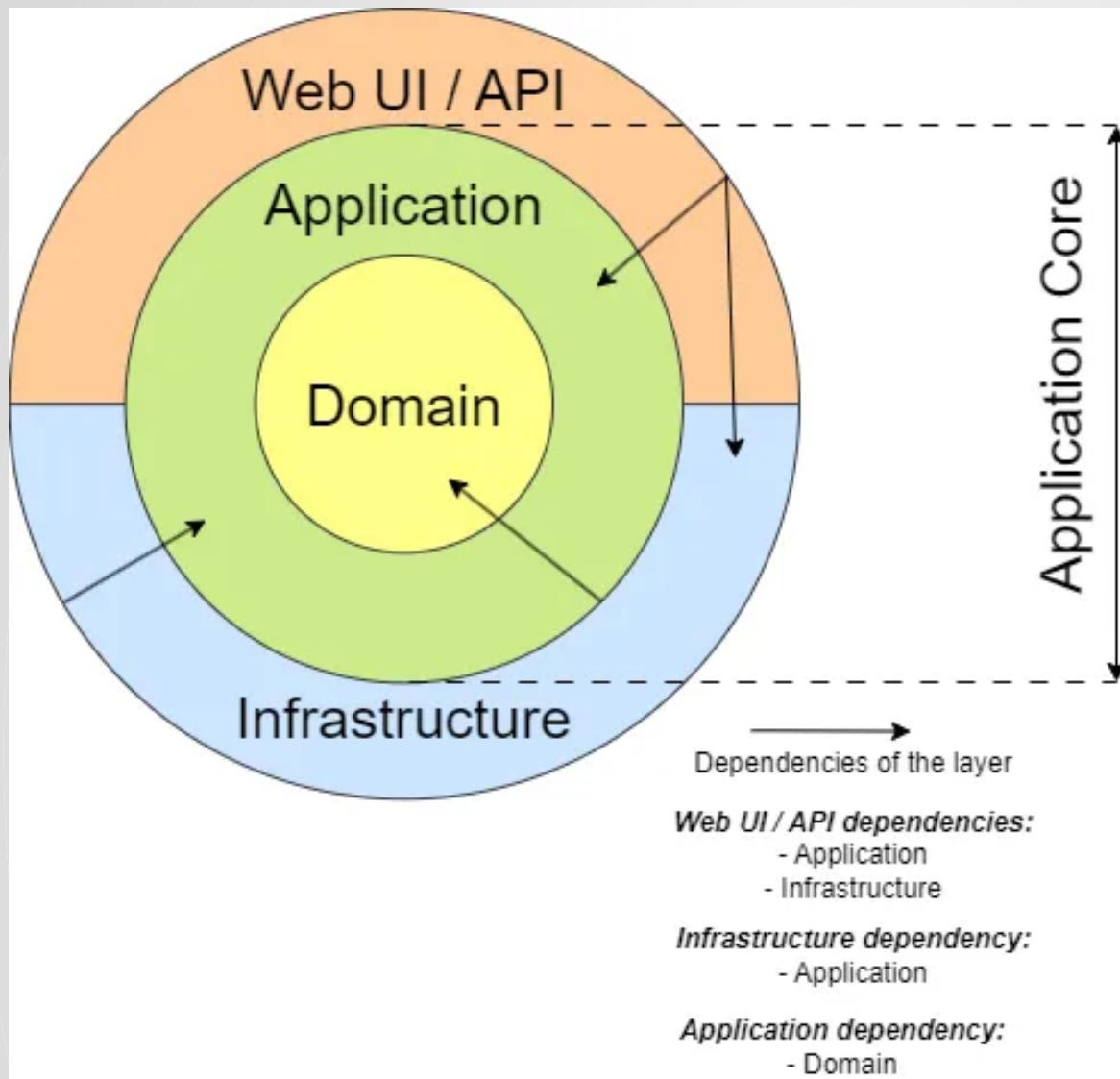


Dependency Injection

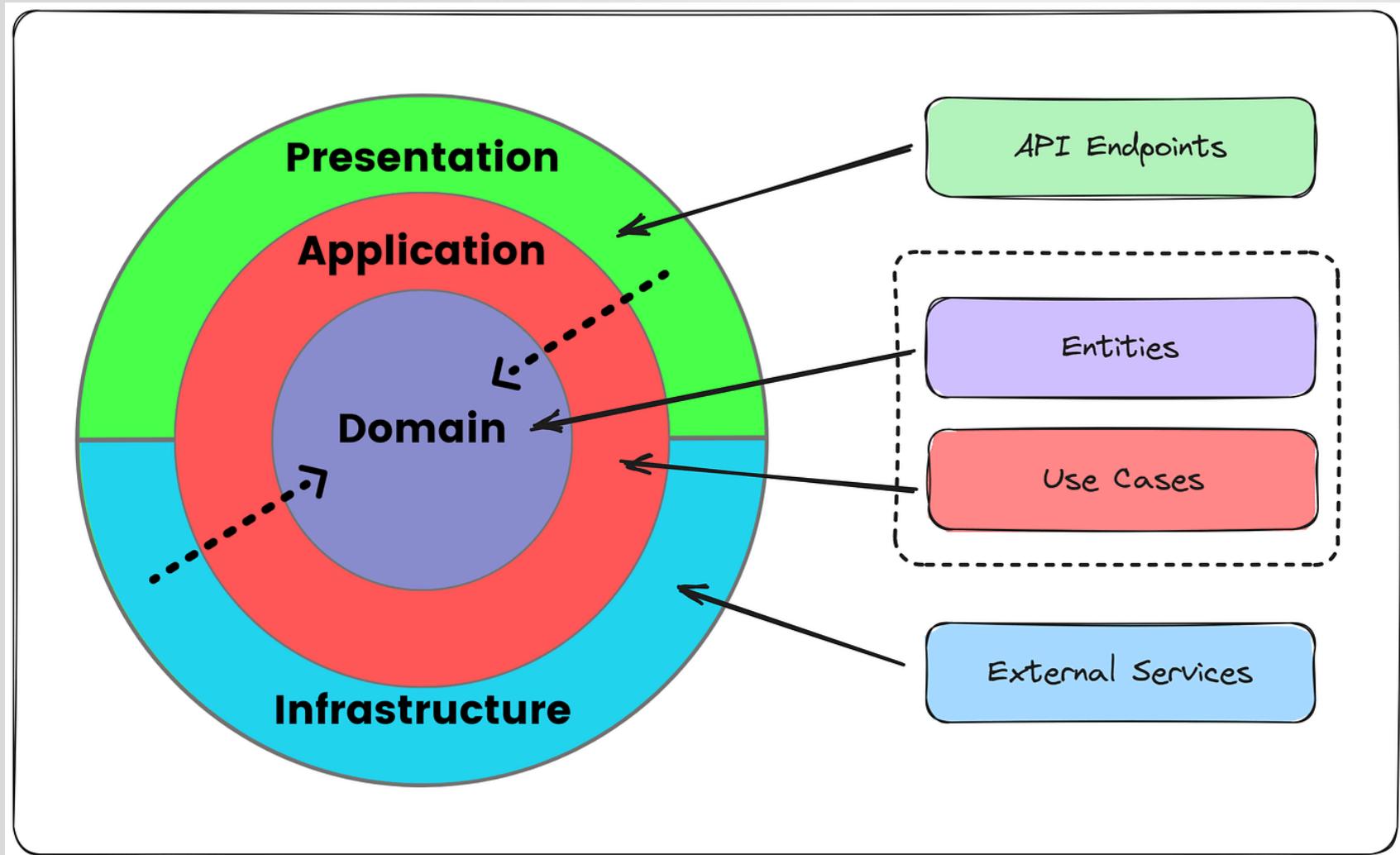
- Keine Codeänderung notwendig
 - Solange Interface kompatibel bleibt



Abhängigkeiten



Beispiel WebApi



Domain-Layer

- Basis-Entities mit Id und RowVersion
- Domain-Entities
- Notwendige Contracts (Interfaces) für Business-Logik
- Validierungen, die auf Geschäftslogik beruhen
 - Z.B. Maximaler Rabatt ist abhängig vom bisherigen Umsatz
 - Sichern Konsistenz in Datenbank

BaseEntity

```
public abstract class BaseEntity : IBaseEntity
{
    /// <summary>
    /// Primärschlüssel (Identity). Protected set, damit nur EF/abgeleitete Klassen setzen können.
    /// </summary>
    6 references
    public int Id { get; protected set; }

    /// <summary>
    /// Concurrency-Token (RowVersion). EF setzt diesen Wert bei jeder Änderung.
    /// Dient zur Erkennung konkurrierender Updates.
    /// </summary>
    [Timestamp]
    3 references
    public byte[] RowVersion { get; set; } = [];
}
```

Beispiel-Entity: Sensor

```
public class Sensor : BaseEntity
{
    /// <summary>
    /// Standort/Ort des Sensors, z.B. "Wohnzimmer".
    /// </summary>
    13 references
    public string Location { get; private set; } = string.Empty;

    /// <summary>
    /// Anzeigename des Sensors, z.B. "Temperatur".
    /// </summary>
    13 references
    public string Name { get; private set; } = string.Empty;

    1 reference
    public ICollection<Measurement> Measurements { get; set; } = default!;

    0 references
    private Sensor() { } // Für EF Core notwendig (parameterloser Konstruktor)
```

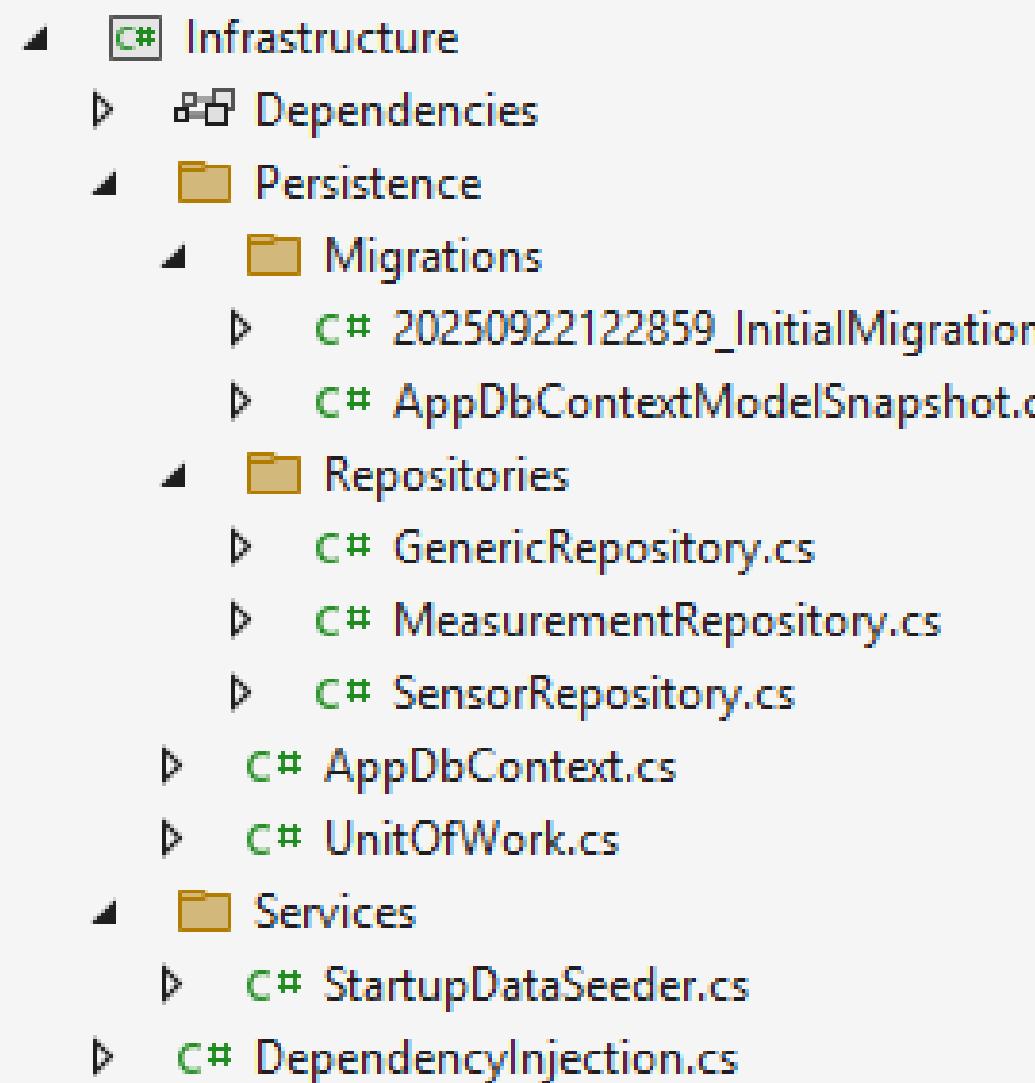
Application-Layer

- Use-Cases der Anwendung
- Im einfachsten Fall CRUD
 - Interfaces für die Persistence → Abstraktion nicht Implementierung
- Ablauf gibt die Validierung vor (nicht Zustand des Entities)
 - Beispiel: Ein Benutzer kann sein Passwort nur ändern, wenn er das alte Passwort korrekt eingegeben hat.
- Häufig: Validierung von DTOs

Infrastructure-Layer

- Implementierungen der Services
 - Persistenz
 - CSV-Reader
 - Seeding
 - ...
- Methode für UI: Registrierung der Services

Infrastructure - Überblick



DI-Registrierung für Infrastruktur

- Wird in Program.cs (UI) aufgerufen

```
public static IServiceCollection AddInfrastructure(this IServiceCollection services, string csvPath,
    string? connectionString = null)
{
    // Falls kein ConnectionString übergeben, lokalen SQL Server Express verwenden
    connectionString ??= "Server=(localdb)\\MSSQLLocalDB;Database=Iot;Trusted_Connection=True;Multip";
    services.AddDbContext<AppDbContext>(opt => opt.UseSqlServer(connectionString));

    // Repositories und UoW (Scoped: pro HTTP-Request eine Instanz)
    services.AddScoped<ISensorRepository, SensorRepository>();
    services.AddScoped<IMeasurementRepository, MeasurementRepository>();
    services.AddScoped<IUnitOfWork, UnitOfWork>();

    // Hosted Service zum initialen Datenimport beim Start der Anwendung
    //services.AddHostedService<StartupDataSeeder>();
    services.AddSingleton<IHostedService>(serviceProvider =>
        new StartupDataSeeder(csvPath, serviceProvider));
    return services;
}
```

StartupDataSeeder

```
public class StartupDataSeeder : IHostedService
{
    private readonly string _csvPath;
    private readonly IServiceProvider _provider;
    private readonly SemaphoreSlim _lock = new(1, 1);

    1 reference
    public StartupDataSeeder(string csvPath, IServiceProvider provider) ...

    0 references
    public async Task StartAsync(CancellationToken cancellationToken)
    {
        using var scope = _provider.CreateScope();
        var ctx = scope.ServiceProvider.GetRequiredService<AppDbContext>();
        await ctx.Database.MigrateAsync(cancellationToken);

        // Nur seeden, wenn noch keine Sensoren existieren (idempotent)
        if (await ctx.Sensors.AnyAsync(cancellationToken)) return;

        var allMeasurements = await ReadMeasurementsFromCsv(cancellationToken);
        ctx.Measurements.AddRange(allMeasurements);
        await ctx.SaveChangesAsync(cancellationToken);
    }
}
```

Migration

- CLI-Tools installieren
 - `dotnet tool install –global dotnet-ef`
- CLI
 - `dotnet ef migrations add InitialMigration --project ./Infrastructure --startup-project ./Api --output-dir ./Persistence/Migrations`
 - `dotnet ef database update --project ./Infrastructure --startup-project ./Api`
- Package-Manager Console
 - Default project: Infrastructure, Startup project: Api
 - `Add-Migration InitialMigration`
 - `Update-Database`

UnitOfWork - IDisposable

```
public class UnitOfWork : IUnitOfWork, IDisposable
{
    private readonly AppDbContext _ctx;
    private bool _disposed;

    /// <summary> Zugriff auf Sensor-Repository.</summary>
    5 references
    public ISensorRepository Sensors { get; }

    /// <summary> Zugriff auf Measurement-Repository.</summary>
    5 references
    public IMeasurementRepository Measurements { get; }

    0 references
    public UnitOfWork(AppDbContext ctx, ISensorRepository sensors, IMeasurementRepository measurements)
    {
        _ctx = ctx;
        Sensors = sensors;
        Measurements = measurements;
    }

    /// <summary> Persistiert alle Änderungen in die DB. Gibt die Anzahl der betroff ...
    2 references
    public Task<int> SaveChangesAsync(CancellationToken ct = default) => _ctx.SaveChangesAsync(ct);
```

Generic Repository

5 references

```
public class GenericRepository<T> : IGenericRepository<T> where T : class, IBaseEntity
{
    protected readonly AppDbContext _ctx;
    protected readonly DbSet<T> _set;
```

2 references

```
public GenericRepository(AppDbContext ctx)
{
    _ctx = ctx;
    _set = ctx.Set<T>();
}
```

Generische Get-Methoden

```
/// <summary> Holt eine Entität per Primärschlüssel.
```

1 reference

```
public virtual async Task<T?> GetByIdAsync(int id, CancellationToken ct = default)
    => await _set.FindAsync([id], ct);
```

```
/// <summary> Holt alle Entitäten (optional gefiltert und sortiert). NoTracking ...
```

5 references

```
public virtual async Task<IReadOnlyCollection<T>> GetAllAsync(
    CancellationToken ct = default,
    Func<IQueryable<T>, IOrderedQueryable<T>>? orderBy = null,
    Expression<Func<T, bool>>? filter = null)
{
    IQueryable<T> query = _set.AsNoTracking();
    if (filter is not null)
        query = query.Where(filter);
    if (orderBy is not null)
        query = orderBy(query);
    return await query.ToListAsync(ct);
}
```

Generische Änderungsmethoden

```
/// <summary> Fügt eine neue Entität hinzu (noch nicht gespeichert).
```

2 references

```
public virtual async Task AddAsync(T entity, CancellationToken ct = default)
=> await _set.AddAsync(entity, ct);
```

```
/// <summary> Fügt mehrere neue Entitäten hinzu (noch nicht gespeichert).
```

1 reference

```
public virtual async Task AddRangeAsync(IEnumerable<T> entities, CancellationToken ct = default)
=> await _set.AddRangeAsync(entities, ct);
```

```
/// <summary> Markiert eine Entität als geändert.
```

1 reference

```
public virtual void Update(T entity) => _set.Update(entity);
```

```
/// <summary> Entfernt eine Entität.
```

1 reference

```
public virtual void Remove(T entity) => _set.Remove(entity);
```

```
/// <summary> Entfernt mehrere Entitäten.
```

1 reference

```
public virtual void RemoveRange(IEnumerable<T> entities) => _set.RemoveRange(entities);
```

Repository wird schlanker

```
public class SensorRepository(AppDbContext ctx) : GenericRepository<Sensor>(ctx), ISensorRepository
{
    /// <summary>
    /// Findet einen Sensor anhand von Standort und Name.
    /// </summary>
    public async Task<Sensor?> GetByLocationAndNameAsync(string location, string name,
        CancellationToken ct = default)
        => await _ctx.Sensors.FirstOrDefaultAsync(s => s.Location == location && s.Name == name, ct);

    // <summary>
    // Überschreibt GetAllAsync, um die Sensoren konsistent sortiert zurückzugeben.
    // </summary>
    //public override async Task<IReadOnlyCollection<Sensor>> GetAllAsync(
    //    CancellationToken ct = default,
    //    Func<IQueryable<Sensor>, IOrderedQueryable<Sensor>>? orderBy = null,
    //    Expression<Func<Sensor, bool>>? filter = null)
    //    => await (orderBy is null
    //        ? _ctx.Sensors.AsNoTracking().WhereIf(filter).OrderBy(s => s.Location).ThenBy(s => s.Name)
    //        : orderBy(_ctx.Sensors.AsNoTracking().WhereIf(filter)))
    //    .ToListAsync(ct);
}
```

Presentation-Layer

- UI-der Anwendung
 - WPF-View in Windows-Anwendungen
 - HTML-View mit CSS und JS in Webanwendungen
 - Swagger-UI für API
- Validierungen der Benutzereingabe
 - Korrektes Format, ...



Swagger
Supported by SMARTBEAR

Select a definition

Measurement API v1

Measurement API v1 OAS 3.0

</swagger/v1/swagger.json>

API zum Auslesen von Sensoren und Messwerten aus einer CSV

Measurements



GET /api/Measurements



GET /api/Measurements/by-sensor-id/{sensorId}



Sensors



GET /api/Sensors



POST /api/Sensors/addsensor



Swagger direkt starten

<https://json.schemastore.org/launchsettings.json>

```
1  {
2      "$schema": "https://json.schemastore.org/launchsettings.json",
3      "profiles": {
4          "http": {
5              "commandName": "Project",
6              "dotnetRunMessages": true,
7              "launchBrowser": false,
8              "applicationUrl": "http://localhost:5186",
9              "environmentVariables": {
10                  "ASPNETCORE_ENVIRONMENT": "Development"
11              },
12              "launchUrl": "swagger"
13          }
14      }
15 }
```

SensorsController – SRP?

- Aufgabe des Controllers?

```
    /// <summary>
    /// Endpunkte rund um Sensoren.
    /// </summary>
    [ApiController]
    [Route("api/[controller]")]
    0 references
    public class SensorsController(IUnitOfWork uow) : ControllerBase
    {
        /// <summary>
        /// Liefert alle Sensoren sortiert nach Location und Name.
        /// </summary>
        [HttpGet]
        [ProducesResponseType(typeof(IEnumerable<GetSensorDto>), StatusCodes.Status200OK)]
        1 reference
        public async Task<IActionResult> GetAll(CancellationToken ct)
        {
            var sensors = await uow.Sensors.GetAllAsync(ct,
                q => q.OrderBy(s => s.Location).ThenBy(s => s.Name));
            // Nur die Felder an den Client geben, die nötig sind (ohne RowVersion)
            var dtos = sensors.Select(s => new GetSensorDto(s.Id, s.Location, s.Name));
            return Ok(dtos);
        }
    }
```

AddSensor – SingleResponsabilityPrinciple?

```
[HttpPost("addsensor")]
[ProducesResponseType(typeof(GetSensorDto), StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status409Conflict)]
0 references
public async Task<IActionResult> AddSensor([FromBody] CreateSensorRequestDto request, CancellationToken ct)
{
    if (request.Equals(default))
        return BadRequest("Request ist erforderlich.");

    var location = request.Location?.Trim();
    var name = request.Name?.Trim();

    if (string.IsNullOrWhiteSpace(location))
        return BadRequest("Location ist erforderlich.");

    if (string.IsNullOrWhiteSpace(name) || name!.Length < 2)
        return BadRequest("Sensornname muss mindestens zwei Zeichen lang sein.");

    if (string.Equals(location, name, StringComparison.OrdinalIgnoreCase))
        return BadRequest("Sensornname darf nicht mit der Location übereinstimmen.");

    var existing = await uow.Sensors.GetByLocationAndNameAsync(location!, name!, ct);
    if (existing is not null)
        return Conflict($"Sensor '{name}' an Location '{location}' existiert bereits.");
}

    var sensor = new Sensor(location!, name!);
    await uow.Sensors.AddAsync(sensor, ct);
    await uow.SaveChangesAsync(ct);

    var dto = new GetSensorDto(sensor.Id, sensor.Location, sensor.Name);
    return CreatedAtAction(nameof(GetAll), null, dto);
}
```

Übung – Ergänzung IoT-App

- GetSensorByLocationAndName
- GetSensorsWithNumberOfMeasurements
- UpdateSensor
- DeleteSensor
- GetAllMeasurements ordered by TimeStamp desc
- GetMeasurementsBySensorId
 - Paging

GetMeasurementsBySensorId

Request URL

```
https://localhost:7885/api/Measurements/by-sensor-id/7?page=1&pageSize=10
```

Response body

```
[  
  {  
    "id": 41070,  
    "sensorId": 7,  
    "value": 481,  
    "timestamp": "2024-10-24T23:59:02"  
  },  
  {  
    "id": 41060,  
    "sensorId": 7,  
    "value": 491,  
    "timestamp": "2024-10-24T23:55:40"  
  },  
  {  
    "id": 41034,  
    "sensorId": 7,  
    "value": 501,  
    "timestamp": "2024-10-24T23:47:37"  
  },  
  {  
    "id": 41024,  
    "sensorId": 7,  
    "value": 511,  
    "timestamp": "2024-10-24T23:40:10"  
  }]
```

Paging-Info im Header

```
  },
  {
    "id": 40863,
    "sensorId": 7,
    "value": 536,
    "timestamp": "2024-10-24T22:47:36"
  },
  {
    "id": 40854,
```

Response headers

```
content-type: application/json; charset=utf-8
date: Wed, 24 Sep 2025 09:02:30 GMT
server: Kestrel
x-page: 1
x-page-size: 50
x-total-count: 3096
```

Abfrage am SQL-Server

- Query wird tatsächlich am Server ausgeführt

```
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (29ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [s].[Id], [s].[Location], [s].[Name], (
          SELECT COUNT(*)
          FROM [Measurements] AS [m]
          WHERE [s].[Id] = [m].[SensorId])
      FROM [Sensors] AS [s]
      ORDER BY [s].[Location], [s].[Name]
```

Erweiterung der Architektur

- SRP für Controller → verschlanken
 - CQRS-Pattern
- Mapping-Aufgaben über zentralen Mapper (Mapster)
- Validierung über Fluent-Validation
- Wer nachlesen will
 - <https://www.ezzylearning.net/tutorial/building-blazor-web-applications>

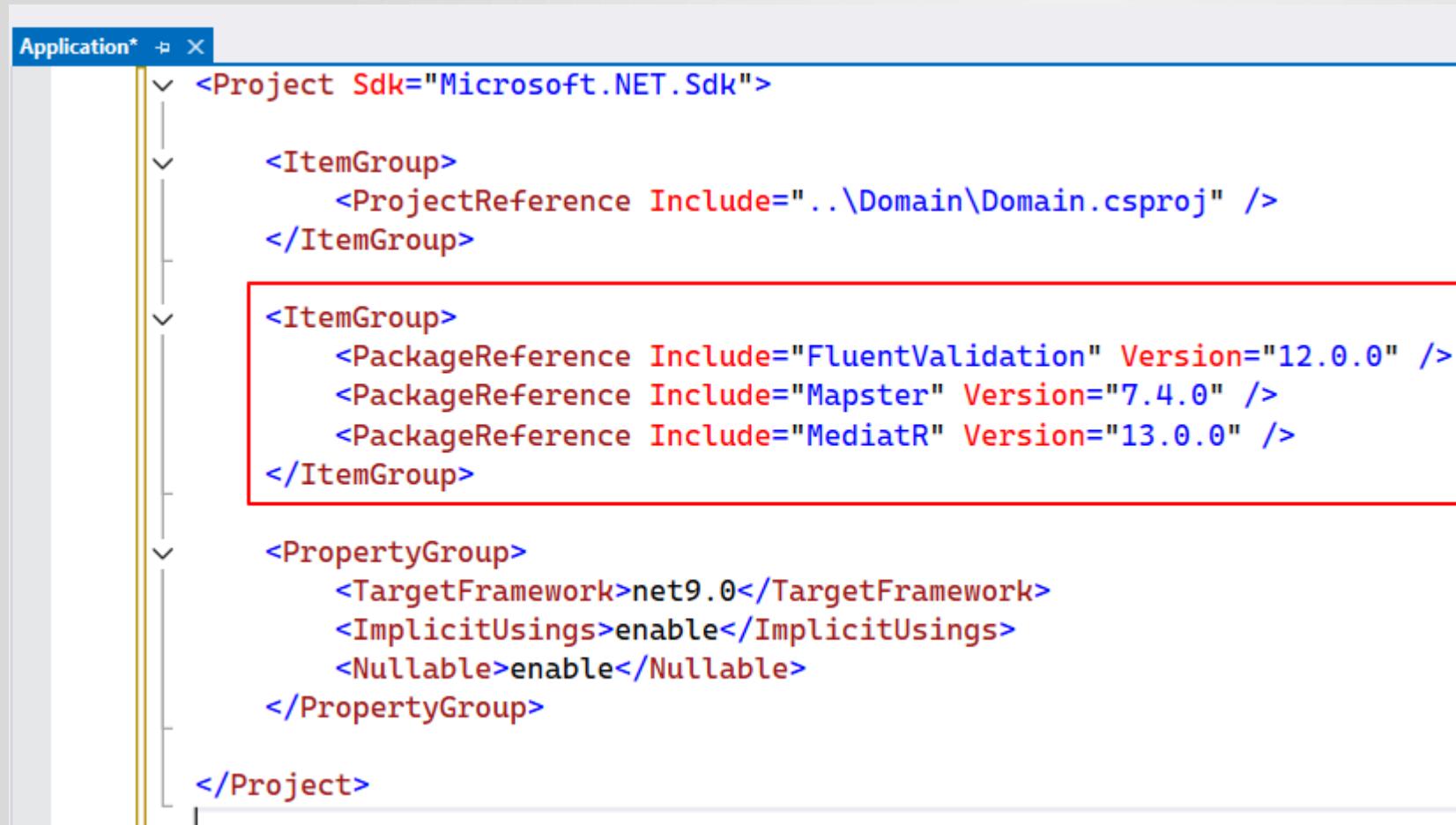
CQRS - Command Query Responsibility Segregation

- Trennung der Lese- und Schreiboperationen einer Anwendung in **zwei strikt getrennte Modelle**
 - **Queries**
 - Reine Leseoperationen liefern DTOs
 - **Commands**
 - Datenänderungen
- Implementiert mehrere Entwurfsprinzipien
 - Single Responsibility Principle aus SOLID
 - Separation of Concerns
 - Command-Pattern
 - Jeder Schreibzugriff ist ein eigener Befehl

Mediator-Pattern

- Zentraler Kommunikationsknoten zwischen Objekten
 - Lose Kopplung → Objekte kennen nur Mediator
- Strukturierung der Requests
 - Je Aufgabe ein Handler, der dann vom Mediator aufgerufen wird
- Nachteil: Anwendung wirkt aufgebläht
 - Mehr Verzeichnisse und Dateien
 - Viel Boilerplate-Code
 - Bei kleinen Anwendungen → viel Overhead

Application-Layer verwendet NuGet-Packages



```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <ProjectReference Include=".\\Domain\\Domain.csproj" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="FluentValidation" Version="12.0.0" />
    <PackageReference Include="Mapster" Version="7.4.0" />
    <PackageReference Include="MediatR" Version="13.0.0" />
  </ItemGroup>
  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

Queries – Verwendung im Controller

- SRP → bleibt schlank
- Kernaufgabe: Request empfangen und Response zurücksenden
- Datenbeschaffung und Mapping geschieht woanders
 - Kann dann auch wiederverwendet werden

```
[HttpGet]
[ProducesResponseType(typeof(IEnumerable<GetSensorDto>), StatusCodes.Status200OK)]
0 references
public async Task<IActionResult> GetAll(CancellationToken ct)
{
    var dtos = await mediator.Send(new GetAllSensorsQuery(), ct);
    return Ok(dtos);
}
```

Queries (MediatR) im Applicationlayer

- Daten aus DB laden → UnitOfWork/Repository
- Schema bleibt immer gleich

```
public record GetAllSensorsQuery : IRequest<IReadOnlyCollection<GetSensorDto>>;  
  
public class GetAllSensorsQueryHandler(IUnitOfWork uow) : IRequestHandler<GetAllSensorsQuery,  
    IReadOnlyCollection<GetSensorDto>>  
{  
    0 references  
    public async Task<IReadOnlyCollection<GetSensorDto>> Handle(GetAllSensorsQuery request,  
        CancellationToken cancellationToken)  
    {  
        var sensors = await uow.Sensors.GetAllAsync(cancellationToken,  
            orderBy: q => q.OrderBy(s => s.Location).ThenBy(s => s.Name));  
        return sensors.Adapt<IReadOnlyCollection<GetSensorDto>>();  
    }  
}
```

Command-Pattern im Detail

4 references

```
public readonly record struct CreateSensorCommand(string Location, string Name) : IRequest<GetSensorDto>;
```

- Immutable Wertetyp → keine defensive Kopie nötig
- Parameter des Records entsprechen genau dem DTO
- Rückgabetypr ist GetSensorDto

CommandHandler mit Mapster

```
public class CreateSensorCommandHandler(IUnitOfWork uow) :  
    IRequestHandler<CreateSensorCommand, GetSensorDto>  
{  
    0 references  
    public async Task<GetSensorDto> Handle(CreateSensorCommand request,  
        CancellationToken cancellationToken)  
    {  
        var entity = new Sensor(request.Location, request.Name);  
        await uow.Sensors.AddAsync(entity, cancellationToken);  
        await uow.SaveChangesAsync(cancellationToken);  
        return entity.Adapt<GetSensorDto>();  
    }  
}
```

Domain Driven Development

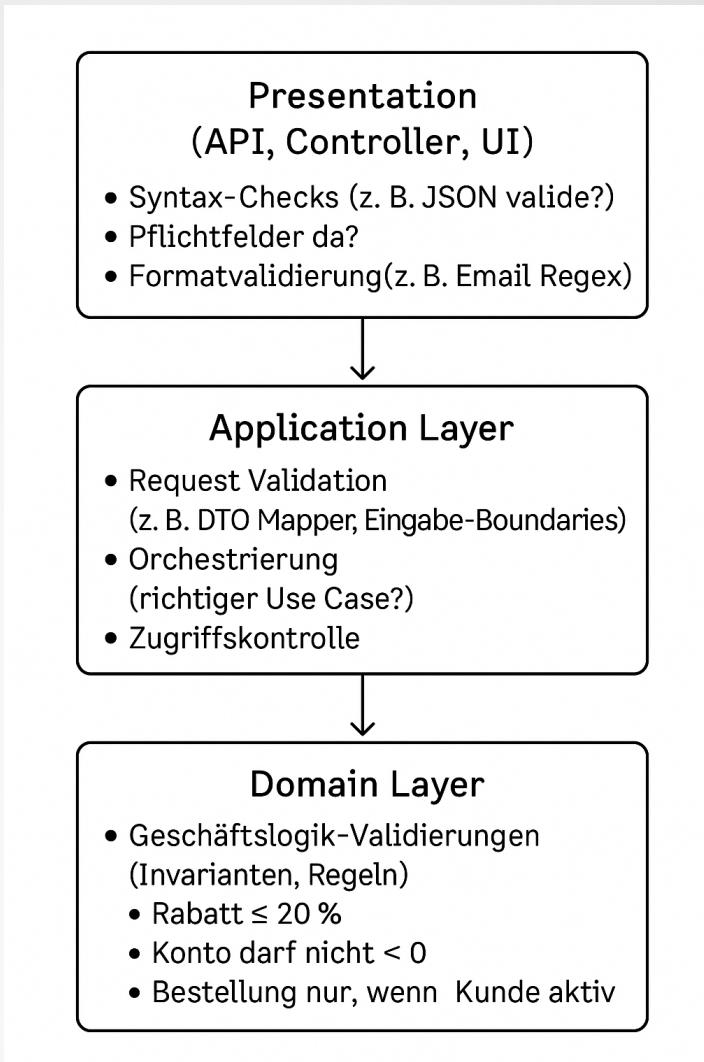
- Orientierung an der Businesslogik
- Domain steht mit ihren fachlichen Regeln im Mittelpunkt
 - Entities als zentrale Elemente entsprechen Busimnessobjekten
 - Value Object ohne eigene Identität aber mit eigenen Regeln
 - Adresse
 - Einhaltung der Regeln wird innerhalb des Entities garantiert
- Domainservices prüfen Regeln die über Entitätsgrenzen hinausgehen
 - Z.B. Keine zwei Sensoren mit selben Namen und Location

Validation

- Prüft auf mehreren Ebenen, ob die Daten (Eingabe, Empfang, ...) korrekt sind
- Ebenen
 - Presentation Layer: Syntax, Eingabefehler, ...
 - Use-Case: korrekt im Sinne des Anwendungsfalls
 - Domain: werden Businessrules eingehalten
 - Infrastructure: Regeln für die Abbildung auf DB
 - Unique-Indizes, String-Längen, Wertebereiche, ...
- Letztlich ist wichtig, dass keine fehlerhaften Daten persistiert werden

Validation auf mehreren Ebenen

- Domain-Regeln werden auch in höheren Ebenen verwendet → DRY
 - Application, UI
- Unterscheidung
 - Entitäts-intern
 - Simple Unitests
 - Über Entitätsgrenzen
 - Komplexer
 - Z.B. keine Sensoren mit gleicher Location und gleichem Namen
 - Benötigen UOW-Zugriff
 - In UI (Browser) nicht sinnvoll wiederverwendbar



Eigener Typ ValidationResult

- Mit Factory-Methods
 - Wozu?
 - Alternativen?

The screenshot shows a code editor window with the following code:

```
namespace Domain.Common;

/// <summary>
/// Lightweight domain validation result with factory helpers.
/// </summary>
15 references
public sealed record ValidationResult(bool IsValid, string? ErrorMessage)
{
    3 references
    public static ValidationResult Success() => new(true, null);
    4 references
    public static ValidationResult Failure(string message) => new(false, message);
}
```

A red box highlights the namespace declaration. A red arrow points from the word "Success" in the first static constructor to the "Success" method in the ValidationResult class definition.

Domain-Rules zentral verwalten

- Code syntaktisch und semantisch analysieren

```
0 references
public static class SensorSpecifications
{
    public const int NameMinLength = 2;

    1 reference
    public static ValidationResult CheckLocation(string location) ...

    1 reference
    public static ValidationResult CheckName(string name) =>
        string.IsNullOrWhiteSpace(name)
            ? ValidationResult.Failure("Name darf nicht leer sein.")
        : name.Trim().Length < NameMinLength
            ? ValidationResult.Failure($"Name muss mindestens {NameMinLength} Zeichen haben.")
            : ValidationResult.Success();

    1 reference
    public static ValidationResult CheckNameNotEqualLocation(string name, string location) =>
        string.Equals(name.Trim(), location.Trim(), StringComparison.OrdinalIgnoreCase)
            ? ValidationResult.Failure("Name darf nicht der Location entsprechen.")
            : ValidationResult.Success();
```

Ohne Bedingungsoperator (?:)

1 reference

```
public static ValidationResult CheckName(string name)
{
    if (string.IsNullOrWhiteSpace(name))
    {
        return ValidationResult.Failure("Name darf nicht leer sein.");
    }
    if (name.Trim().Length < NameMinLength)
    {
        return ValidationResult.Failure($"Name muss mindestens {NameMinLength} Zeichen enthalten");
    }
    return ValidationResult.Success();
}
```

Alle Prüfungen aufrufen

- Was hat es mit dem yield return auf sich

```
public static IEnumerable<ValidationResult> CheckAll(string location, string name)
{
    yield return CheckLocation(location);
    yield return CheckName(name);
    yield return CheckNameNotEqualLocation(name, location);
}
```

Alternative zu yield return

- Unterschiede?

```
0 references
public static IEnumerable<ValidationResult> CheckAll(string location, string name)
{
    var results = new List<ValidationResult>
    {
        CheckLocation(location),
        CheckName(name),
        CheckNameNotEqualLocation(name, location)
    };

    return results;
}
```

Im Fehlerfall → DomainValidationException

```
namespace Domain.Exceptions;

/// <summary>
/// Ausnahme für Verletzungen von Domain-Invarianten / Validierungsregeln innerhalb des Domain-Modells.
/// </summary>
0 references
public class DomainValidationException(string message) : Exception(message)
{}
```

Entity sichert seinen validen Zustand ab

```
/// <summary>
/// Erzeugt einen neuen Sensor.
/// Im Fehlerfall wird beim ersten Fehler eine DomainValidationException geworfen.
/// </summary>
2 references
public Sensor(string location, string name)
{
    foreach (var result in SensorSpecifications.CheckAll(location, name))
    {
        if (!result.IsValid)
            throw new DomainValidationException(result.ErrorMessage);
    }
    Location = location;
    Name = name;
}
```

Validation greift, aber ...

- Wir wollen einen BadRequest ohne Stacktrace

Curl

```
curl -X 'POST' \
  'https://localhost:7085/api/Sensors' \
  -H 'accept: text/plain' \
  -H 'Content-Type: application/json' \
  -d '{
    "location": "asd",
    "name": "asd"
}'
```

Request URL

```
https://localhost:7085/api/Sensors
```

Server response

Code Details

500 Error: response status is 500
Undocumented

Response body

```
Domain.Exceptions.DomainValidationException: Name darf nicht der Location entsprechen.
  at Domain.Entities.Sensor..ctor(String location, String name) in D:\work\CSharp\Blazor_2025_26\Übungen\01_CleanArchitecture\01_CleanArchitecture\ValidationTemplate\Application\Features\Sensors\Commands\CreateSensor\CommandHandler.cs:line 23
  at Application.Features.Sensors.Commands.CreateSensor.CreateCommandHandler.Handle(CreateSensorCommand request, CancellationToken cancellationToken) in D:\work\CSharp\Blazor_2025_26\Übungen\01_CleanArchitecture\ValidationTemplate\Application\Features\Sensors\Commands\CreateSensor\CommandHandler.cs:line 23
  at Api.Controllers.SensorsController.Create(CreateSensorCommand command, CancellationToken ct) in D:\work\CSharp\Blazor_2025_26\Übungen\01_CleanArchitecture\ValidationTemplate\Infrastructure\Controllers\SensorsController.cs:line 73
  at Microsoft.AspNetCore.Mvc.Infrastructure.ActionMethodExecutor.TaskOfIActionResultExecutor.Execute(ActionContext actionContext, Object controller, Object[] arguments)
  at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.<InvokeActionMethodAsync>g__Logged|12_1(ControllerActionInvoker controllerActionInvoker, Object controller, Object[] arguments)
  at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.<InvokeNextActionFilterAsync>g__Awaited|10_0(ControllerActionInvoker controllerActionInvoker, ValueTask result)
```

Exception → Responsecode 400

```
0 references
public class CreateSensorCommandHandler(IUnitOfWork uow) :
    IRequestHandler<CreateSensorCommand, GetSensorDto>
{
    0 references
    public async Task<GetSensorDto> Handle(CreateSensorCommand request,
        CancellationToken cancellationToken)
    {
        var entity = new Sensor(request.Location, request.Name);
        await uow.Sensors.AddAsync(entity, cancellationToken);
        await uow.SaveChangesAsync(cancellationToken);
        return entity.Adapt<GetSensorDto>();
    }
}
```

Eine Möglichkeit - CommandHandler

```
public async Task<CreateSensorResultDto> Handle(CreateSensorCommand request,  
    CancellationToken cancellationToken)  
{  
    try  
    {  
        var entity = new Sensor(request.Location, request.Name);  
        await uow.Sensors.AddAsync(entity, cancellationToken);  
        await uow.SaveChangesAsync(cancellationToken);  
        return CreateSensorResultDto.Ok(entity.Adapt<GetSensorDto>());  
    }  
    catch (Exception ex)  
    {  
        return CreateSensorResultDto.Fail(ex.Message);  
    }  
}
```

Sieht doch schon besser aus

Curl

```
curl -X 'POST' \  
  'https://localhost:7085/api/Sensors' \  
  -H 'accept: text/plain' \  
  -H 'Content-Type: application/json' \  
  -d '{  
    "location": "string",  
    "name": "string"  
}'
```

Request URL

```
https://localhost:7085/api/Sensors
```

Server response

Code Details

400 Error: response status is 400

Response body

```
{  
  "error": "Name darf nicht der Location entsprechen."  
}
```

Use-Case Validation

- AddMeasurementCommand(location, name, timestamp, value)
- Regeln
 - Sensor mit location/name wird angelegt, falls nicht vorhanden
 - Regeln für Sensor müssen natürlich eingehalten werden
 - Timestamp muss innerhalb der letzten Stunde liegen
 - Nur im UseCase, sonst könnten keine historischen Daten importiert werden
- Validation im Domain-Layer und im Application-Layer

- erzeuge mir ein command AddMeasurementCommand(location, name, timestamp, value) mit einen entsprechenden handler, der für den sensor mit location/name (wenn sensor nicht existiert, wird er angelegt) ein measurement und biete den usecase über einen endpoint im measurementscontroller an.
- wenn ich bei addmeasurement einen fehlerhaften sensornamen eingebe, kommt die exception bis in den response durch. fange mir die domainvalidationexceptions ab und wandle sie in einen badrequest um

AddMeasurementCommandHandler

- Ohne Validation

```
public async Task<GetMeasurementDto> Handle(AddMeasurementCommand request, CancellationToken cancellationToken)
{
    // Sensor anhand Location/Name holen oder anlegen
    var sensor = await uow.Sensors.GetByLocationAndNameAsync(request.Location, request.Name, cancellationToken);
    if (sensor is null)
    {
        sensor = new Sensor(request.Location, request.Name);
        await uow.Sensors.AddAsync(sensor, cancellationToken);
        // Save to get sensor.Id for FK
        await uow.SaveChangesAsync(cancellationToken);
    }

    var measurement = new Measurement(sensor, request.Value, request.Timestamp);
    await uow.Measurements.AddAsync(measurement, cancellationToken);
    await uow.SaveChangesAsync(cancellationToken);

    return measurement.Adapt<GetMeasurementDto>();
}
```

Zumindest die Domain-Validation greift

Curl

```
curl -X 'POST' \  
'https://localhost:7085/api/Measurements' \  
-H 'accept: text/plain' \  
-H 'Content-Type: application/json' \  
-d '{  
  "location": "a",  
  "name": "a",  
  "timestamp": "2025-10-04T08:33:12.197Z",  
  "value": 0  
}'
```

Request URL

<https://localhost:7085/api/Measurements>

Server response

Code Details

500 Error: response status is 500

Undocumented

Response body

```
Domain.Exceptions.DomainValidationException: Name muss mindestens 2 Zeichen haben.  
  at Domain.Entities.Sensor..ctor(String location, String name) in D:\work\CSharp\Blazor_2025_26\Übungen\01_CleanArchitecture\...  
  at Application.Features.Measurements.Commands.AddMeasurement.AddMeasurementCommandHandler.Handle(AddMeasurementCommand...  
_2025_26\Übungen\01_CleanArchitecture\Validation\2_Application\Application\Features\Measurements\Commands\AddMeasurement...  
  at Api.Controllers.MeasurementsController.Add(AddMeasurementCommand command, CancellationToken ct) in D:\work\CSharp\B...  
Controllers\MeasurementsController.cs:line 52
```

Timestamp-Einschränkung greift nicht

```
{  
    "location": "asd",  
    "name": "fnh",  
    "timestamp": "2026-10-04T08:33:12.197Z",  
    "value": 99  
}
```

201

Response body

```
{  
    "id": 41072,  
    "sensorId": 16,  
    "value": 99,  
    "timestamp": "2026-10-04T08:33:12.197Z"  
}
```

Validation im CommandHandler

- Sensor wie gehabt
- Funktionaler Code und Validationcode ist vermischt

```
public async Task<CreateMeasurementResultDto> Handle(CreateMeasurementCommand request,  
    CancellationToken cancellationToken)  
{  
    // Sensor anhand Location/Name holen oder anlegen  
    var sensor = await uow.Sensors.GetByLocationAndNameAsync(request.Location, request.Name, cancellationToken);  
    if (sensor is null)  
    {  
        try...  
    }  
  
    if (request.Timestamp > DateTime.UtcNow || request.Timestamp < DateTime.UtcNow.AddHours(-1))  
    {  
        return CreateMeasurementResultDto.Fail("Invalid timestamp.");  
    }  
  
    var measurement = new Measurement(sensor, request.Value, request.Timestamp);  
    await uow.Measurements.AddAsync(measurement, cancellationToken);  
    await uow.SaveChangesAsync(cancellationToken);  
  
    return CreateMeasurementResultDto.Ok(measurement.Adapt<GetMeasurementDto>());  
}
```

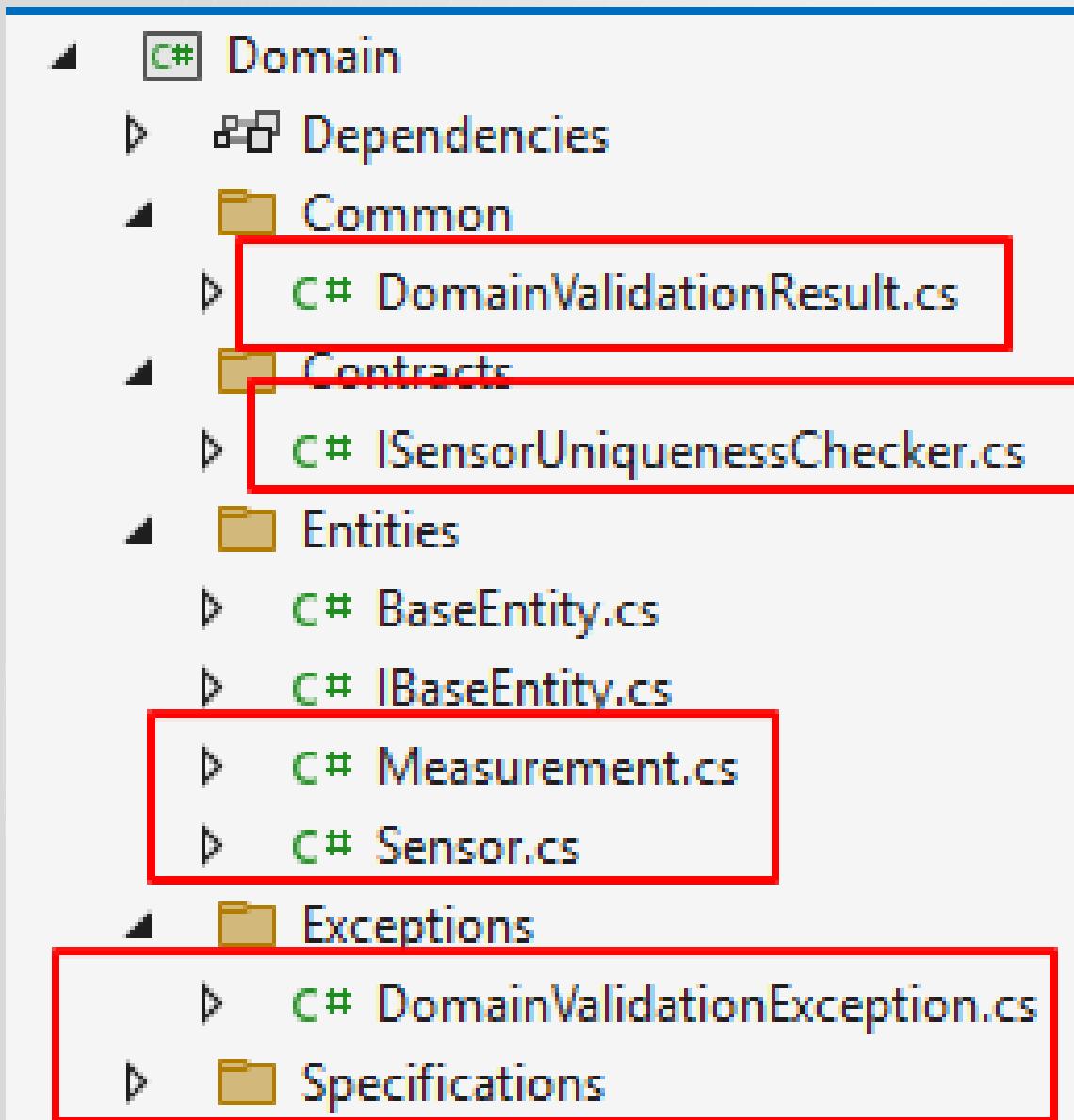
Unique Index

```
modelBuilder.Entity<Sensor>(sensor =>
{
    sensor.Property(s => s.Location).HasMaxLength(100).IsRequired();
    sensor.Property(s => s.Name).HasMaxLength(100).IsRequired();
    sensor.Property(s => s.RowVersion).IsRowVersion();
    // Uniqueness constraint (Location + Name)
    sensor.HasIndex(s => new { s.Location, s.Name })
        .IsUnique()
        .HasDatabaseName("UX_Sensors_Location_Name");
});
```


Validation und Kommunikation

- Domain-Layer (DomainValidationResult)
 - Intern
 - Extern übergreifend mittels DI von Services (Contracts)
 - Lösen im Fehlerfall DomainValidationExceptions aus
- Application-Layer
 - Mediator hängt Validation in Request-Pipeline (ValidationBehavior)
 - Führt alle Fluent-Validations aus
 - Erzeugt ValidationExceptions (FluentValidation)
- API-Layer
 - ValidationException-Middleware fängt Exceptions ab
 - Erzeugt die passenden Response-Statuscodes

DomainValidation



DomainValidation → Specifications

```
public static class SensorSpecifications
{
    public const int NameMinLength = 2;

    1 reference
    public static ValidationResult CheckLocation(string location) ...

    // public static ValidationResult CheckName(string name) => ...

    1 reference
    public static ValidationResult CheckName(string name)
    {
        if (string.IsNullOrWhiteSpace(name))
        {
            return ValidationResult.Failure("Name darf nicht leer sein.");
        }
        if (name.Trim().Length < NameMinLength)
        {
            return ValidationResult.Failure($"Name muss mindestens {NameMinLength} Zeichen haben.");
        }
        return ValidationResult.Success();
    }
}
```

Internal und external

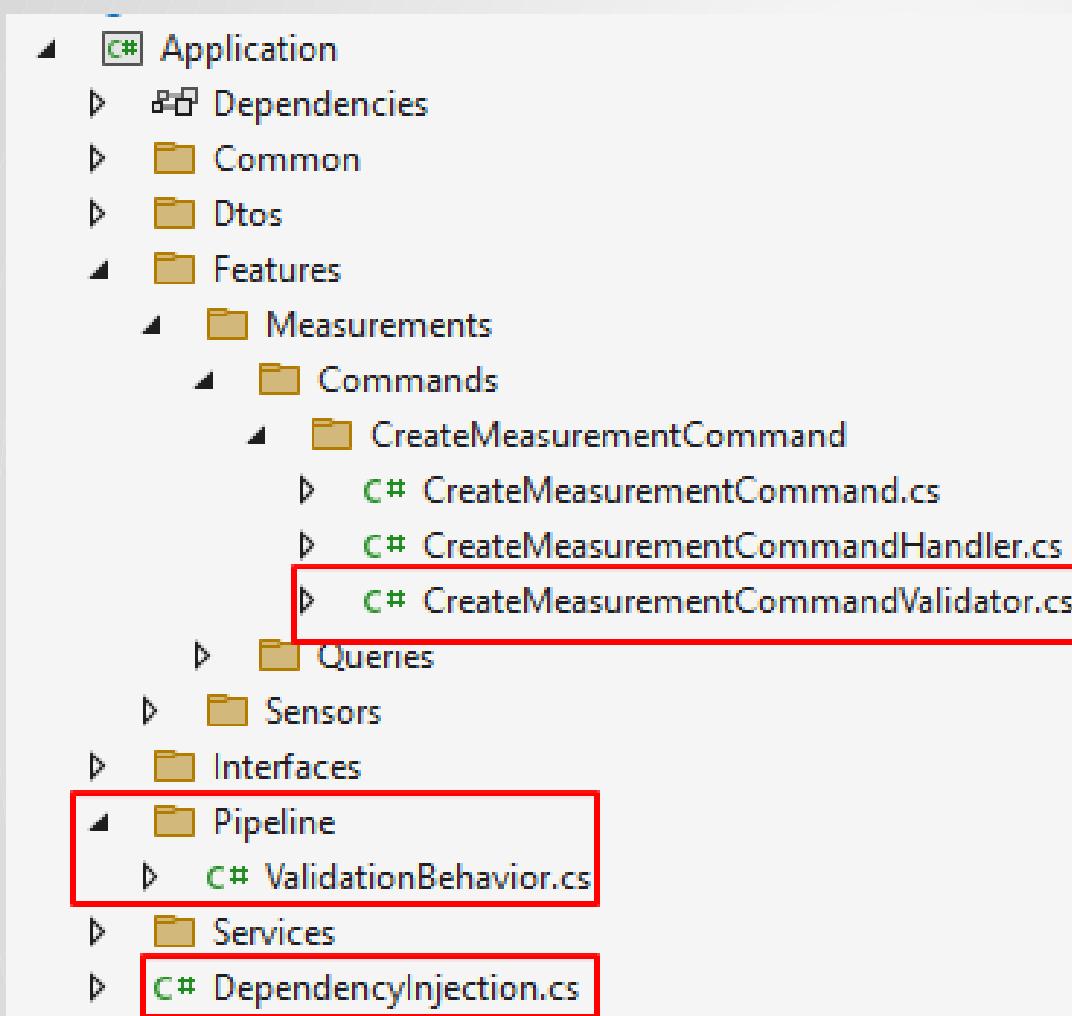
```
public static void ValidateEntityInternal(string location, string name)
{
    var validationResults = new List<ValidationResult>
    {
        CheckLocation(location),
        CheckName(name),
        CheckNameNotEqualLocation(name, location)
    };
    foreach (var result in validationResults)
    {
        if (!result.IsValid)
        {
            throw new DomainValidationException(result.ErrorMessage!);
        }
    }
}
```

2 references

```
public static async Task ValidateEntityExternal(int id, string location, string name,
    ISensorUniquenessChecker uniquenessChecker, CancellationToken ct = default)
{
    if (!await uniquenessChecker.IsUniqueAsync(id, location, name, ct))
        throw new DomainValidationException("Ein Sensor mit der gleichen Location und dem gleichen
```

ApplicationValidation

- Prüft auf UseCase-Ebene



Application-Validation mit FluentValidation

- Validieren auf Use-Case-Ebene
 - Z.B. Bei Passwort-Änderung muss das alte Passwort stimmen
 - User-Entity hat kein altes Passwort
 - Anpassen der Fehlermeldungstexte an Use-Case
- Jeder Request kann eine eigene Request-Pipeline haben
 - Logging, Caching, Validation, ...
- Logik kann damit vor und nach jedem Handler eingeschleust werden
- MediatR bietet dafür Pipeline Behaviors an

Validator → FluentValidation

- Codeverdopplung ist nicht notwendig

```
using FluentValidation;

namespace Application.Features.Measurements.Commands.CreateMeasurementCommand;

1 reference
public class CreateMeasurementCommandValidator : AbstractValidator<CreateMeasurementCommand>
{
    0 references
    public CreateMeasurementCommandValidator()
    {
        RuleFor(x => x.Timestamp)
            .Must(ts => ts <= DateTime.UtcNow && ts >= DateTime.UtcNow.AddHours(-1))
            .WithMessage("Invalid timestamp.");

        //RuleFor(x => x.Location)
        //    .NotEmpty();

        //RuleFor(x => x.Name)
        //    .NotEmpty()
        //    .MinimumLength(2)
        //    .Must((cmd, name) => !string.Equals(name, cmd.Location, StringComparison.OrdinalIgnoreCase))
        //    .WithMessage("Name must not equal Location.");
    }
}
```

Mediator fügt Validation in Pipeline ein

- Trennung Kernfunktion und Fehlerbehandlung
- Wird vor dem eigentlichen CommandHandler aufgerufen
- Löst im Fehlerfall Exception aus oder ruft in der Pipeline next() auf
→ CommandHandler

```
public sealed class ValidationBehavior<TRequest, TResponse>(IEnumerable<IValidator<TRequest>> validators)
    : IPipelineBehavior<TRequest, TResponse>
{
    where TRequest : notnull

    0 references
    public async Task<TResponse> Handle(TRequest request, RequestHandlerDelegate<TResponse> next,
        CancellationToken cancellationToken)
    {
        if (validators.Any())
        {
            var context = new ValidationContext<TRequest>(request);
            var validationResults = await Task.WhenAll(validators.Select(v => v.ValidateAsync(context,
                cancellationToken)));
            var failures = validationResults.SelectMany(r => r.Errors).Where(f => f is not null).ToList();
            if (failures.Count != 0)
            {
                throw new ValidationException(failures);
            }
        }

        return await next(cancellationToken);
    }
}
```

Application – Services registrieren

```
public static class DependencyInjection
{
    /// <summary>
    /// Registriert DbContext, Repositories, UnitOfWork, CSV-Reader und Seeder.
    /// </summary>
    1 reference
    public static IServiceCollection AddApplication(this IServiceCollection services)
    {
        // CQRS + MediatR + FluentValidation
        services.AddMediatR(cfg =>
        {
            cfg.RegisterServicesFromAssembly(typeof(IUnitOfWork).Assembly); // Application-Assembly
        });
        services.AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehavior<,>));
        services.AddValidatorsFromAssembly(typeof(IUnitOfWork).Assembly);

        // Domain Services
        services.AddScoped<ISensorUniquenessChecker, SensorUniquenessChecker>();

        return services;
    }
}
```

Registrierung ist der Kleber

```
// CQRS + MediatR + FluentValidation
services.AddMediatR(cfg =>
{
    cfg.RegisterServicesFromAssembly(typeof(IUnitOfWork).Assembly); // Application-Assembly
});
services.AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehavior<,>));
services.AddValidatorsFromAssembly(typeof(IUnitOfWork).Assembly);
```

- Registriert alle MediatR-Komponenten (Requests, Handler, ...) des Application-Assemblies
- Fügt ein Pipeline-Behavior (Validation) hinzu, das vor dem Handler aufgerufen wird
- Registriert alle Validators aus dem Assembly

Result und Result<T>

- Wie DomainValidationResult, aber auch mit mehreren positiven Ergebnissen
- Wieder mit Factory-Methoden
- Rückgabetyp primär bei Commands
 - Queries nur in komplexen Fällen mit Exceptions

```
public enum ResultType
{
    Success,
    Created,
    NoContent,
    NotFound,
    ValidationException,
    Conflict,
    Error
}
```

Result-Klasse

```
16 references
public class Result
{
    2 references
    public bool IsSuccess { get; }
    0 references
    public bool IsFailure => !.IsSuccess;
    9 references
    public string? Message { get; }
    3 references
    public ResultType Type { get; }

    7 references
    protected Result(bool isSuccess, string? message, ResultType type)
    {
        IsSuccess = isSuccess;
        Message = message;
        Type = type;
    }

    0 references
    public static Result Success(string? message = null) => new(true, message, ResultType.Success);
    0 references
    public static Result NotFound(string? message = null) => new(false, message, ResultType.NotFound);
    0 references
    public static Result NoContent(string? message = null) => new(true, message, ResultType.NoContent);
    0 references
    public static Result ValidationError(string? message = null) => new(false, message, ResultType.ValidationError);
    0 references
    public static Result Conflict(string? message = null) => new(false, message, ResultType.Conflict);
    0 references
    public static Result Error(string? message = null) => new(false, message, ResultType.Error);
}
```

Result<T>

```
public class Result<T> : Result
{
    6 references
    public T? Value { get; }

    7 references
    private Result(bool isSuccess, T? value, string? message, ResultType type)
        : base(isSuccess, message, type)
    {
        Value = value;
    }

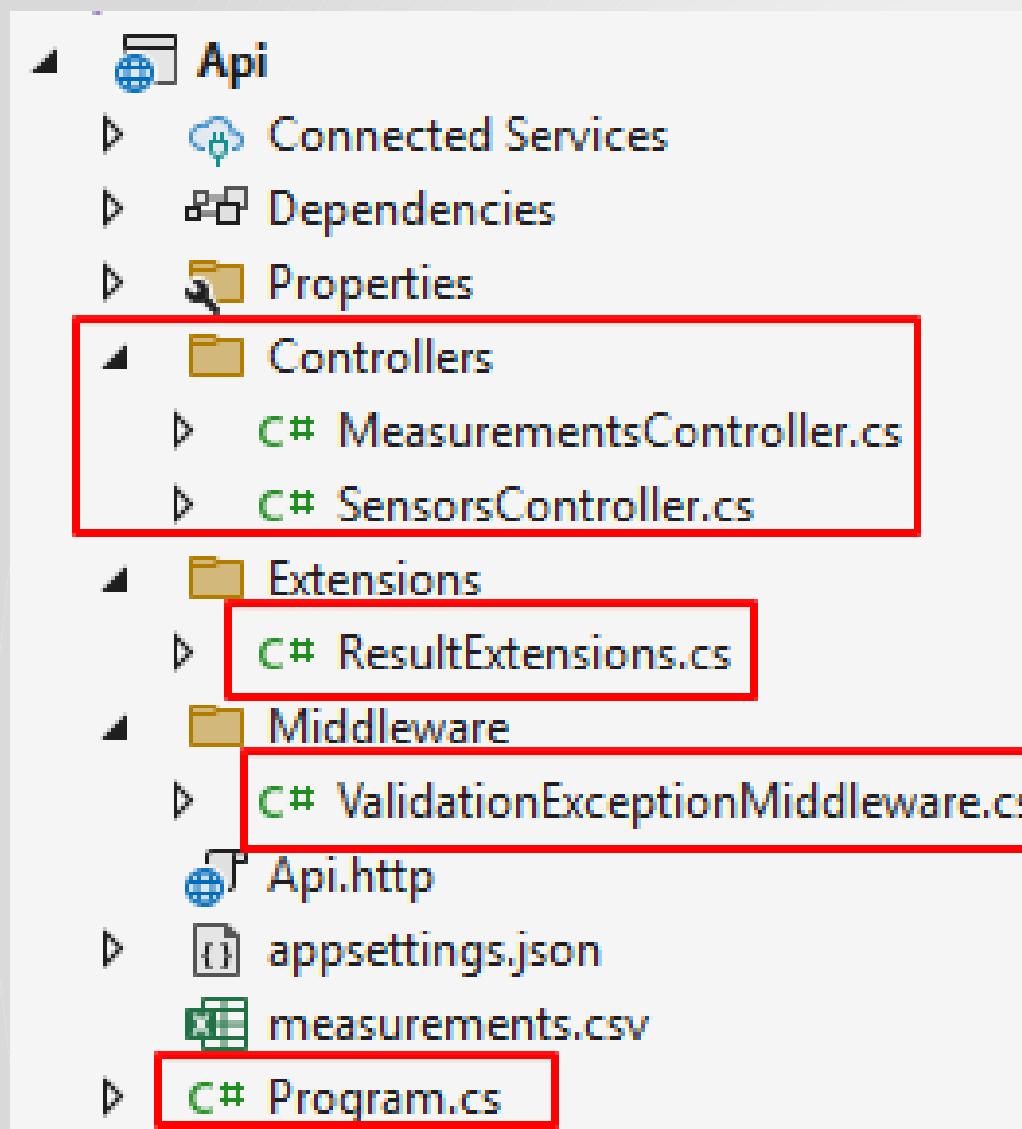
    1 reference
    public static Result<T> Success(T value, string? message = null) ...
    2 references
    public static Result<T> Created(T value, string? message = null)
        => new(true, value, message, ResultType.Created);
    1 reference
    public static Result<T> NoContent(T value, string? message = null) ...
    0 references
    public new static Result<T> NotFound(string? message = null) ...
    1 reference
    public new static Result<T> ValidationError(string? message = null) ...
    0 references
    public new static Result<T> Conflict(string? message = null) ...
    0 references
    public new static Result<T> Error(string? message = null) ...
}
```

Verwendung in CommandHandler

0 references

```
public sealed class CreateSensorCommandHandler(IUnitOfWork uow,
    ISensorUniquenessChecker uniquenessChecker)
: IRequestHandler<CreateSensorCommand, Result<GetSensorDto>>
{
    0 references
    public async Task<Result<GetSensorDto>> Handle(CreateSensorCommand request,
        CancellationToken cancellationToken)
    {
        // Sensor über Domänenlogik erstellen und persistieren
        var entity = await Sensor.CreateAsync(request.Location, request.Name,
            uniquenessChecker, cancellationToken);
        await uow.Sensors.AddAsync(entity, cancellationToken);
        await uow.SaveChangesAsync(cancellationToken);
        return Result<GetSensorDto>.Created(entity.Adapt<GetSensorDto>());
    }
}
```

API → Responses



ResultExtensions

- Bindeglied zwischen deiner Anwendungsschicht (Application) und der API-Schicht (Presentation)
- Verarbeitet fehlerfreie Requests

Vorteil	Beschreibung
 Schichtentrennung	Application kennt kein ASP.NET, Controller kein Domänen-Result-Mapping
 Konsistente HTTP-Antworten	Alle Endpunkte reagieren gleich auf ResultType
 Weniger Boilerplate	Controller-Code bleibt minimal
 Zentral wartbar & testbar	Änderungen an Responses nur an einer Stelle nötig
 REST-Konformität	Unterstützt <code>CreatedAtAction</code> inkl. <code>Location</code> -Header

Schlanke Controller

- Keine Codeverdopplung
 - Result kapselt
- Exceptionhandling über Requestpipeline
- Mapping in ActionResult über ExtensionMethode

```
//> /<-- Controller
[HttpPost]
[ProducesResponseType(typeof(GetSensorDto), StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status409Conflict)]
0 references
public async Task<IActionResult> Create([FromBody] CreateSensorCommand command, CancellationToken ct)
{
    // Erstellung anstoßen; Result in passenden HTTP-Status umwandeln (Created etc.)
    var result = await mediator.Send(command, ct);
    return result.ToActionResult(this, createdAtAction: nameof(GetById),
        routeValues: new { id = result.Value.Id });
    // try ...
}
```

Einheitliche Response-Erzeugung

4 references

```
public static IActionResult ToActionResult<T>(this Result<T> result,
    ControllerBase controller,
    string? createdAtAction = null,
    object? routeValues = null)
{
    return result.Type switch
    {
        ResultType.Success => controller.Ok(result.Value),
        // Für Created nach Möglichkeit immer CreatedAtAction verwenden, damit der Location
        ResultType.Created => createdAtAction is not null
            ? controller.CreatedAtAction(createdAtAction, routeValues, result.Value)
            : controller.StatusCode(StatusCodes.Status201Created, result.Value),
        ResultType.NoContent => controller.NoContent(),
        ResultType.NotFound => controller.NotFound(result.Message),
        ResultType.ValidationError => controller.BadRequest(result.Message),
        ResultType.Conflict => controller.Conflict(result.Message),
        _ => controller.Problem(
            detail: result.Message ?? "An unexpected error occurred.",
            statusCode: 500
        )
    };
}
```

ValidationException-Middleware

- Verarbeitet aufgetretene Exceptions zentral
 - Keine try/catch in den Controllern (DRY)
 - Einheitliches Format
- Passende HTTP-Codes für unterschiedliche Fehler
- Merke:
 - Erfolgreiche Verarbeitung → Result<T>
 - Fehlerbehandlung → Exception → Middleware

Unique Index → DB

```
modelBuilder.Entity<Sensor>(sensor =>
{
    sensor.Property(s => s.Location).HasMaxLength(100).IsRequired();
    sensor.Property(s => s.Name).HasMaxLength(100).IsRequired();
    sensor.Property(s => s.RowVersion).IsRowVersion();
    // Uniqueness constraint (Location + Name)
    sensor.HasIndex(s => new { s.Location, s.Name })
        .IsUnique()
        .HasDatabaseName("UX_Sensors_Location_Name");
});
```