

Proyecto de Cierre: Sistema de Gestión y Telemetría de IoT Industrial (ASP.NET Core & C#)

🎯 Objetivo General

Desarrollar una plataforma empresarial moderna en ASP.NET Core que permita registrar dispositivos de IoT (Internet de las Cosas), recibir telemetría en tiempo real, procesar alertas automáticas basadas en reglas de negocio y exponer una API REST segura para un panel de control interactivo.

🔧 Stack Tecnológico

- **Lenguaje:** C# (última versión)
- **Framework Backend:** ASP.NET Core Web API / Minimal APIs
- **Acceso a Datos:** Entity Framework Core (EF Core) con **SQL Server** o **PostgreSQL**
- **Tiempo Real:** SignalR (para notificaciones instantáneas de alertas al panel)
- **Autenticación & Seguridad:** ASP.NET Core Identity con **JWT (JSON Web Tokens)** y políticas basadas en roles (*Admin, Operador, Auditor*).
- **Documentación de API:** Swagger / OpenAPI con anotaciones detalladas.

📋 Requerimientos Funcionales

1. Módulo de Autenticación y Gestión de Usuarios (ASP.NET Core Identity + JWT)

- Registro de usuarios y asignación de roles mediante Claims y Policies.
- Endpoint de autenticación (/api/auth/login) que devuelva un token JWT con tiempo de expiración y refresh tokens.

2. Módulo de Gestión de Dispositivos (CRUD y Estados)

- Registro de dispositivos industriales (Dispositivo: ID, Nombre, Ubicación, Estado: *Activo, Inactivo, Mantenimiento*).
- Relación de uno a muchos: Un dispositivo genera múltiples registros de telemetría.

3. Ingesta de Telemetría (Endpoints de Alto Rendimiento)

- Endpoint protegido para recibir datos de sensores (/api/telemetry):
 - Recibe un JSON con: DeviceId, Temperatura, Humedad, Presion, Timestamp.
- **Lógica de Negocio / Alertas:** Al recibir la telemetría, el sistema debe evaluar reglas predefinidas (ej. si la temperatura supera los 80°C). Si se cumple, debe generar una **Alerta** en la base de datos de forma automática.

4. Notificaciones en Tiempo Real (SignalR)

- Implementar un Hub de SignalR (/hubs/alerts) para que, al generarse una alerta crítica por telemetría, el backend la emita inmediatamente a todos los clientes conectados al panel de control sin necesidad de recargar la página.

5. Reportes y Consultas Avanzadas (LINQ & EF Core)

- Endpoint para consultar el historial de telemetría con **paginación, filtros por rango de fechas y ordenamiento**.

- Uso de consultas LINQ optimizadas con proyecciones (Select) para no sobrecargar la memoria.

Arquitectura Sugerida

Se recomienda aplicar principios de **Clean Architecture** o **Arquitectura en Capas**:

1. **Domain:** Entidades de negocio (Dispositivo, TelemetryData, Alert) y excepciones personalizadas.
2. **Application:** Interfaces de servicios, DTOs, validaciones con **FluentValidation** y mapeos con **AutoMapper** (o mapeo manual).
3. **Infrastructure:** Contexto de EF Core, repositorios o patrón Unit of Work, implementaciones de servicios externos.
4. **API (Presentation):** Controladores o Minimal APIs, configuración de middlewares de manejo global de excepciones y autenticación.

Fases de Desarrollo Sugeridas

1. **Fase 1: Configuración del Core y Base de Datos**
 - Crear la solución en .NET (Clean Architecture o Web API estructurada).
 - Configurar EF Core, las migraciones iniciales y las relaciones de las entidades.
2. **Fase 2: Seguridad y Autenticación**
 - Implementar Identity y la emisión de tokens JWT.
 - Proteger los endpoints usando atributos [Authorize(Policy = "AdminOnly")].
3. **Fase 3: Motor de Telemetría y Alertas**
 - Desarrollar el endpoint de recepción de datos y la lógica interna para disparar alertas automáticas.
4. **Fase 4: Tiempo Real y Extras**
 - Integrar SignalR para el push de alertas.
 - Documentar la API con Swagger y añadir pruebas unitarias básicas utilizando **xUnit** y **Moq**.

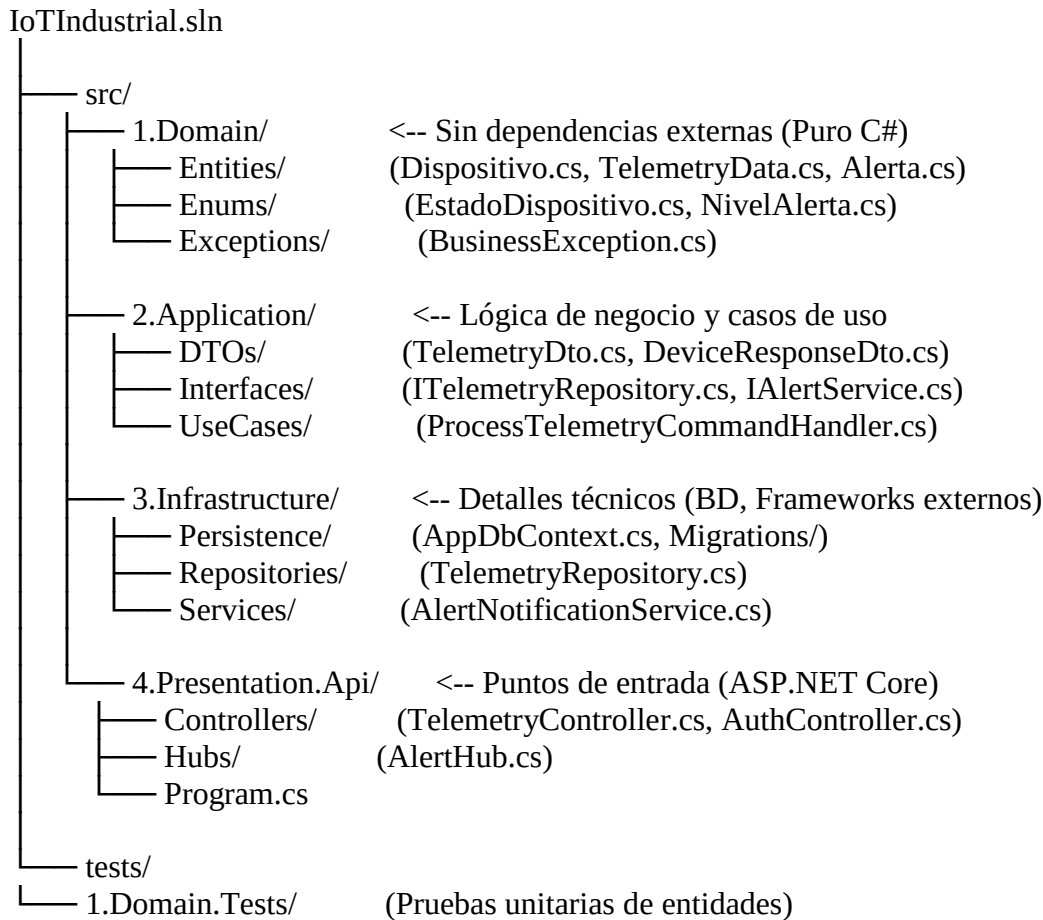
Vamos a mostrar como estructurar los dos pilares más avanzados y críticos de este proyecto: **Clean Architecture** (para mantener el código desacoplado y mantenible) y la **Configuración de SignalR con autenticación JWT** (para el manejo seguro de WebSockets en tiempo real).

Clean Architecture (para mantener el código desacoplado y mantenible) y la **Configuración de SignalR con autenticación JWT** (para el manejo seguro de WebSockets en tiempo real).

Parte 1: Estructurando la Clean Architecture en .NET

Clean Architecture divide tu solución en capas concéntricas, donde **la regla de oro es que las dependencias siempre apuntan hacia adentro**. El Dominio no sabe nada de la Base de Datos ni de la API.

Estructura de Proyectos (Solución en Visual Studio / CLI)



Reglas de Dependencia Clave:

1. **Domain:** Contiene las reglas de negocio puras. No referencia a EF Core ni a ASP.NET Core.
2. **Application:** Define las interfaces (ITelemetryRepository) que usará para comunicarse con el exterior, pero no implementa los detalles. Utiliza patrones como **Mediator** o servicios de aplicación limpios.
3. **Infrastructure:** Implementa las interfaces de Application. Aquí vive Entity Framework Core, SQL Server y las conexiones externas.
4. **Presentation.Api:** Es el punto de arranque. Conecta todo usando el contenedor de Inyección de Dependencias (DI) de .NET (builder.Services.AddScoped...).

🔒 Parte 2: Configurando SignalR con Autenticación JWT

El gran reto con SignalR y WebSockets es que los navegadores o clientes IoT muchas veces **no pueden enviar cabeceras HTTP personalizadas** (Authorization: Bearer <token>) durante el protocolo de conexión inicial (Handshake).

La solución estándar en ASP.NET Core es capturar el token desde la **Query String** de la URL y pasárselo al middleware de JWT.

Paso 1: Configurar el JWT y SignalR en Program.cs

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;

var builder = WebApplication.CreateBuilder(args);

// 1. Configurar Autenticación JWT
var jwtSettings = builder.Configuration.GetSection("JwtSettings");
var secretKey = Encoding.UTF8.GetBytes(jwtSettings["Secret"]!);

builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = jwtSettings["Issuer"],
        ValidAudience = jwtSettings["Audience"],
        IssuerSigningKey = new SymmetricSecurityKey(secretKey)
    }
});

// 🕒 TRUCO CLAVE PARA SIGNALR: Leer el token de la Query String
options.Events = new JwtBearerEvents
{
    OnMessageReceived = context =>
    {
        var accessToken = context.Request.Query["access_token"];
        var path = context.HttpContext.Request.Path;

        // Verifica si la petición va dirigida al Hub de SignalR
        if (!string.IsNullOrEmpty(accessToken) && path.StartsWithSegments("/hubs/alerts"))
        {
            context.Token = accessToken;
        }
        return Task.CompletedTask;
    }
};

builder.Services.AddSignalR();
builder.Services.AddControllers();
```

```

var app = builder.Build();

app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();
app.MapHub<AlertHub>("/hubs/alerts"); // Mapear el Hub

app.Run();

```

Paso 2: Crear el Hub Protegido

Para asegurarte de que solo los usuarios autenticados (o dispositivos con permisos) se conecten al canal de tiempo real, simplemente añades el atributo [Authorize]:

```

C#
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.SignalR;

[Authorize] // Requiere token JWT válido
public class AlertHub : Hub
{
    public override async Task OnConnectedAsync()
    {
        // Puedes acceder al usuario autenticado mediante Context.User
        var username = Context.User?.Identity?.Name;
        await Groups.AddToGroupAsync(Context.ConnectionId, "IndustrialOperators");
        await base.OnConnectedAsync();
    }

    public override async Task OnDisconnectedAsync(Exception? exception)
    {
        await Groups.RemoveFromGroupAsync(Context.ConnectionId, "IndustrialOperators");
        await base.OnDisconnectedAsync(exception);
    }
}

```

Paso 3: Emitir Alertas desde el Backend (Servicio o Controller)

Cuando llegue una telemetría que dispare una alerta crítica, puedes inyectar `IHubContext<AlertHub>` en tu servicio de aplicación para enviarla a todos los conectados:

```

C#
public class AlertNotificationService
{
    private readonly IHubContext<AlertHub> _hubContext;

    public AlertNotificationService(IHubContext<AlertHub> hubContext)
    {
    }
}

```

```

{
    _hubContext = hubContext;
}

public async Task NotificarAlertaCriticaAsync(string dispositivoId, string mensaje)
{
    // Envía el mensaje a todos los clientes en el grupo o conectados al hub
    await _hubContext.Clients.Group("IndustrialOperators")
        .SendAsync("RecibirAlerta", new { DispositivoId = dispositivoId, Mensaje = mensaje,
Timestamp = DateTime.UtcNow });
}
}

```

FluentValidation dentro de la capa de **Application**.

Colocar las validaciones aquí es una práctica clave en Clean Architecture porque **las reglas de validación forman parte de las reglas de negocio y de los casos de uso**, no de la interfaz web ni de la base de datos. De esta manera, si en el futuro decides cambiar tu API por una aplicación de consola o gRPC, las reglas de validación siguen intactas y protegidas.

✂ Paso 1: Instalación de Paquetes

En tu proyecto 2.Application, asegúrate de instalar el paquete NuGet:

```

Bash
dotnet add package FluentValidation.DependencyInjectionExtensions

```

📁 Paso 2: Crear el DTO y su Validador

Imagina que queremos validar los datos de telemetría que llegan de un sensor industrial. Primero definimos el DTO y luego creamos su regla de validación heredando de `AbstractValidator<T>`.

El DTO (CreateTelemetryDto.cs)

```

C#
namespace Application.DTOs;

public record CreateTelemetryDto(
    string DeviceId,
    decimal Temperature,
    decimal Humidity,
    decimal Pressure,
    DateTime Timestamp
);

```

El Validador (CreateTelemetryDtoValidator.cs)

```

C#
using FluentValidation;

```

```

using Application.DTOs;

namespace Application.Validators;

public class CreateTelemetryDtoValidator : AbstractValidator<CreateTelemetryDto>
{
    public CreateTelemetryDtoValidator()
    {
        RuleFor(x => x.DeviceId)
            .NotEmpty().WithMessage("El identificador del dispositivo es obligatorio.")
            .MaxLength(50).WithMessage("El ID del dispositivo no puede superar los 50
caracteres.");

        RuleFor(x => x.Temperature)
            .InclusiveBetween(-50m, 150m).WithMessage("La temperatura debe estar en un rango
industrial seguro entre -50°C y 150°C.");

        RuleFor(x => x.Humidity)
            .InclusiveBetween(0m, 100m).WithMessage("La humedad debe ser un porcentaje válido
entre 0% y 100%.");

        RuleFor(x => x.Timestamp)
            .LessThanOrEqualTo(DateTime.UtcNow).WithMessage("El timestamp no puede ser una
fecha futura.")
            .GreaterThan(DateTime.UtcNow.AddDays(-1)).WithMessage("El registro es demasiado
antiguo para ser procesado.");
    }
}

```

⚙ Paso 3: Integración Elegante con MediatR (Pipeline Behavior)

En una Clean Architecture avanzada, los casos de uso se manejan comúnmente con el patrón **CQRS** (usando la librería **MediatR**). Para evitar validar manualmente en cada controlador o manejador, creamos un **Behavior (Filtro) global** que intercepta cualquier comando y lo valida automáticamente antes de ejecutar la lógica de negocio.

El Behavior de Validación (ValidationBehavior.cs)

```

C#
using FluentValidation;
using MediatR;

namespace Application.Common.Behaviors;

public class ValidationBehavior<TRequest, TResponse> : IPipelineBehavior<TRequest,
TResponse>
    where TRequest : IRequest<TResponse>
{
    private readonly IEnumerable<IValidator<TRequest>> _validators;

```

```

public ValidationBehavior(IEnumerable<IValidator<TRequest>> validators)
{
    _validators = validators;
}

public async Task<TResponse> Handle(TRequest request,
RequestHandlerDelegate<TResponse> next, CancellationToken cancellationToken)
{
    if (_validators.Any())
    {
        var context = new ValidationContext<TRequest>(request);

        // Ejecuta todas las validaciones asociadas al comando
        var validationResults = await Task.WhenAll(
            _validators.Select(v => v.ValidateAsync(context, cancellationToken))
        );

        var failures = validationResults
            .SelectMany(r => r.Errors)
            .Where(f => f != null)
            .ToList();

        if (failures.Count != 0)
        {
            // Lanza una excepción personalizada de negocio si hay errores
            throw new ValidationException(failures);
        }
    }

    return await next();
}
}

```

Paso 4: Registro en el Contenedor de Dependencias

Para que la capa de Application descubra automáticamente todos los validadores y el comportamiento de MediatR, creamos un método de extensión dentro de Application (DependencyInjection.cs):

```

C#
using FluentValidation;
using Microsoft.Extensions.DependencyInjection;
using System.Reflection;
using MediatR;
using Application.Common.Behaviors;

namespace Application;

public static class DependencyInjection
{

```

```

public static IServiceCollection AddApplication(this IServiceCollection services)
{
    var assembly = Assembly.GetExecutingAssembly();

    // Registra todos los validadores que heredan de AbstractValidator en este ensamblado
    services.AddValidatorsFromAssembly(assembly);

    // Registra MediatR y el Pipeline Behavior de validación
    services.AddMediatR(cfg => cfg.RegisterServicesFromAssembly(assembly));
    services.AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehavior<,>));

    return services;
}
}

```

Luego, en el Program.cs de tu API (capa Presentation), simplemente llamas a:

```

C#
builder.Services.AddApplication();

```

Para asegurar que las reglas de negocio y validación funcionen correctamente sin necesidad de levantar toda la API o la base de datos, podemos escribir **pruebas unitarias puras** utilizando **xUnit** y el paquete oficial de ayuda que incluye **FluentValidation**.

✂ Paso 1: Paquetes Necesarios para el Proyecto de Tests

En tu proyecto de pruebas (por ejemplo, tests/Application.Tests), asegúrate de tener instalados los siguientes paquetes NuGet:

```

Bash
dotnet add package xunit
dotnet add package xunit.runner.visualstudio
dotnet add package FluentAssertions
dotnet add package FluentValidation.TestsExtensions

```

□ Paso 2: Implementando las Pruebas Unitarias para el Validador

Gracias a la extensión TestValidate proporcionada por FluentValidation, probar los validadores es extremadamente limpio y expresivo.

Crea una clase llamada CreateTelemetryDtoValidatorTests.cs:

```

C#
using FluentValidation.TestHelper;
using Xunit;
using Application.DTOs;
using Application.Validators;

namespace Application.Tests.Validators;

```

```

public class CreateTelemetryDtoValidatorTests
{
    private readonly CreateTelemetryDtoValidator _validator;

    public CreateTelemetryDtoValidatorTests()
    {
        _validator = new CreateTelemetryDtoValidator();
    }

    [Fact]
    public void Should_Not_Have_Errors_When_Telemetry_Is_Valid()
    {
        // Arrange: Creamos un DTO con datos totalmente válidos
        var validDto = new CreateTelemetryDto(
            DeviceId: "SENSOR-LAB-01",
            Temperature: 25.5m,
            Humidity: 60.0m,
            Pressure: 1.02m,
            Timestamp: DateTime.UtcNow.AddMinutes(-2) // Reciente pero no en el futuro
        );

        // Act
        var result = _validator.TestValidate(validDto);

        // Assert
        result.ShouldNotHaveAnyValidationErrors();
    }

    [Theory]
    [InlineData("")]
    [InlineData(null)]
    public void Should_Have_Error_When_DeviceId_Is_Null_Or_Empty(string invalidDeviceId)
    {
        // Arrange
        var dto = new CreateTelemetryDto(invalidDeviceId, 25.0m, 50.0m, 1.0m, DateTime.UtcNow);

        // Act
        var result = _validator.TestValidate(dto);

        // Assert
        result.ShouldHaveValidationErrorFor(x => x.DeviceId)
            .WithErrorMessage("El identificador del dispositivo es obligatorio.");
    }

    [Theory]
    [InlineData(-51.0)] // Por debajo del límite industrial seguro
    [InlineData(150.1)] // Por encima del límite industrial seguro
    public void Should_Have_Error_When_Temperature_Is_Out_Of_Range(decimal invalidTemp)
    {

```

```

    // Arrange
    var dto = new CreateTelemetryDto("SENSOR-01", invalidTemp, 50.0m, 1.0m,
DateTime.UtcNow);

    // Act
    var result = _validator.TestValidate(dto);

    // Assert
    result.ShouldHaveValidationErrorFor(x => x.Temperature)
        .WithErrorMessage("La temperatura debe estar en un rango industrial seguro entre -50°C y
150°C.");
}

[Theory]
[InlineData(-1.0)]
[InlineData(100.1)]
public void Should_Have_Error_When_Humidity_Is_Out_Of_Percentage_Range(decimal
invalidHumidity)
{
    // Arrange
    var dto = new CreateTelemetryDto("SENSOR-01", 25.0m, invalidHumidity, 1.0m,
DateTime.UtcNow);

    // Act
    var result = _validator.TestValidate(dto);

    // Assert
    result.ShouldHaveValidationErrorFor(x => x.Humidity)
        .WithErrorMessage("La humedad debe ser un porcentaje válido entre 0% y 100%.");
}

[Fact]
public void Should_Have_Error_When_Timestamp_Is_In_The_Future()
{
    // Arrange: Un timestamp adelantado 2 horas
    var dto = new CreateTelemetryDto("SENSOR-01", 25.0m, 50.0m, 1.0m,
DateTime.UtcNow.AddHours(2));

    // Act
    var result = _validator.TestValidate(dto);

    // Assert
    result.ShouldHaveValidationErrorFor(x => x.Timestamp)
        .WithErrorMessage("El timestamp no puede ser una fecha futura.");
}

[Fact]
public void Should_Have_Error_When_Timestamp_Is_Too_Old()
{
    // Arrange: Un registro de hace más de 24 horas

```

```

var dto = new CreateTelemetryDto("SENSOR-01", 25.0m, 50.0m, 1.0m,
DateTime.UtcNow.AddDays(-2));

// Act
var result = _validator.TestValidate(dto);

// Assert
result.ShouldHaveValidationErrorFor(x => x.Timestamp)
    .WithErrorMessage("El registro es demasiado antiguo para ser procesado.");
}
}

```

💡 Ventajas de este enfoque de pruebas:

1. **Velocidad:** Al ser pruebas unitarias puras que no levantan bases de datos ni servidores web, corren en milisegundos.
2. **Uso de [Theory] y [InlineData]:** Te permite probar múltiples escenarios de fallo (como los límites de temperatura o humedad) escribiendo una sola prueba parametrizada, evitando código duplicado.
3. **Independencia:** Si modificas las reglas de negocio en el futuro, estas pruebas te avisarán inmediatamente si rompiste alguna restricción crítica del sistema IoT.

En una arquitectura limpia, la capa de infraestructura es la encargada de implementar las interfaces definidas en la capa de Aplicación. Aquí es donde conectamos el modelo de dominio con el motor de base de datos (por ejemplo, SQL Server o PostgreSQL) utilizando Code First.

✂ Paso 1: Paquetes NuGet Necesarios

En tu proyecto 3.Infraestructure, asegúrate de instalar los siguientes paquetes (asegúrate de usar la versión correspondiente a .NET 8 o .NET 9):

```

Bash
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Design

```

🏠 Paso 2: Configurando las Entidades y el DbContext

Supongamos que en tu capa de Dominio tienes las entidades Dispositivo y TelemetryData. En la infraestructura implementamos el contexto de base de datos (AppDbContext).

El DbContext (AppDbContext.cs)

```

C#
using Microsoft.EntityFrameworkCore;
using Domain.Entities; // Suponiendo que tus entidades están en el Dominio

namespace Infrastructure.Persistence;

public class AppDbContext : DbContext

```

```

{
    public DbSet<Dispositivo> Dispositivos => Set<Dispositivo>();
    public DbSet<TelemetryData> TelemetryDatas => Set<TelemetryData>();
    public DbSet<Alerta> Alertas => Set<Alerta>();

    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // Aplicar configuraciones de Fluent API de manera ordenada
        modelBuilder.ApplyConfigurationsFromAssembly(typeof(AppDbContext).Assembly);
    }
}

```

▀ Paso 3: Configuración con Fluent API (Mapeo de Base de Datos)

Para mantener limpias las entidades de dominio, es una excelente práctica usar configuraciones separadas para EF Core usando `IEntityTypeConfiguration<T>`.

Configuración de Telemetría (TelemetryDataConfiguration.cs)

```

C#
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using Domain.Entities;

namespace Infrastructure.Persistence.Configurations;

public class TelemetryDataConfiguration : IEntityTypeConfiguration<TelemetryData>
{
    public void Configure(EntityTypeBuilder<TelemetryData> builder)
    {
        builder.HasKey(t => t.Id);

        builder.Property(t => t.DeviceId)
            .IsRequired()
            .HasMaxLength(50);

        builder.Property(t => t.Temperature)
            .HasColumnType("decimal(18,2)")
            .IsRequired();

        builder.Property(t => t.Humidity)
            .HasColumnType("decimal(18,2)")
            .IsRequired();
    }
}

```

```

builder.Property(t => t.Pressure)
    .HasColumnType("decimal(18,2)")
    .IsRequired();

builder.Property(t => t.Timestamp)
    .IsRequired();

// Índices para optimizar consultas por dispositivo y fecha
builder.HasIndex(t => new { t.DeviceId, t.Timestamp });
}
}

```

Paso 4: Implementando los Repositorios (Patrón Repositorio / Unit of Work)

La capa de Aplicación define una interfaz (ej. `ITelemetryRepository`), y la capa de Infraestructura la implementa usando EF Core.

Interfaz en Aplicación (`ITelemetryRepository.cs`):

```

C#
namespace Application.Interfaces;

public interface ITelemetryRepository
{
    Task AddAsync(TelemetryData telemetryData, CancellationToken cancellationToken);
    Task<IEnumerable<TelemetryData>> GetByDeviceAsync(string deviceId, DateTime from,
    DateTime to, CancellationToken cancellationToken);
}

```

Implementación en Infraestructura (`TelemetryRepository.cs`):

```

C#
using Microsoft.EntityFrameworkCore;
using Domain.Entities;
using Application.Interfaces;
using Infrastructure.Persistence;

namespace Infrastructure.Repositories;

public class TelemetryRepository : ITelemetryRepository
{
    private readonly AppDbContext _context;

    public TelemetryRepository(AppDbContext context)
    {
        _context = context;
    }

    public async Task AddAsync(TelemetryData telemetryData, CancellationToken
    cancellationToken)

```

```

{
    await _context.TelemetryDatas.AddAsync(telemetryData, cancellationToken);
    await _context.SaveChangesAsync(cancellationToken);
}

public async Task<IEnumerable<TelemetryData>> GetByDeviceAsync(string deviceId,
DateTime from, DateTime to, CancellationToken cancellationToken)
{
    return await _context.TelemetryDatas
        .Where(t => t.DeviceId == deviceId && t.Timestamp >= from && t.Timestamp <= to)
        .OrderByDescending(t => t.Timestamp)
        .ToListAsync(cancellationToken);
}
}

```

🔨 Paso 5: Registro de Infraestructura en Inyección de Dependencias

Creamos un método de extensión similar al de la capa de Aplicación para mantener limpio el Program.cs de la API:

```

C#
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Infrastructure.Persistence;
using Infrastructure.Repositories;
using Application.Interfaces;

namespace Infrastructure;

public static class DependencyInjection
{
    public static IServiceCollection AddInfrastructure(this IServiceCollection services,
    IConfiguration configuration)
    {
        // Configurar la conexión a SQL Server
        services.AddDbContext<AppDbContext>(options =>
            options.UseSqlServer(
                configuration.GetConnectionString("DefaultConnection"),
                b => b.MigrationsAssembly(typeof(AppDbContext).Assembly.FullName)
            ));

        // Registrar los repositorios
        services.AddScoped<ITelemetryRepository, TelemetryRepository>();

        return services;
    }
}

```

¿Cómo se conecta todo en el Program.cs de la API?

Gracias a esta separación modular, tu archivo principal Program.cs queda ultra limpio, limitándose a encender los motores de cada capa:

```
C#
var builder = WebApplication.CreateBuilder(args);

// Inyectar las capas de Clean Architecture
builder.Services.AddApplication();
builder.Services.AddInfrastructure(builder.Configuration);

builder.Services.AddControllers();
var app = builder.Build();

app.MapControllers();
app.Run();
```

El **Manejo Global de Excepciones** es la pieza que corona una arquitectura backend robusta. En lugar de llenar tus controladores de bloques try-catch repetitivos, centralizamos los errores para que cualquier fallo (ya sea una validación fallida de FluentValidation, un error de base de datos o un fallo inesperado) se convierta automáticamente en una respuesta HTTP limpia, estructurada y estandarizada bajo el estándar **RFC 7807 (ProblemDetails)**.

A partir de .NET 8, ASP.NET Core incluye soporte nativo y optimizado para esto mediante la interfaz `ExceptionHandler`.

✂ Paso 1: Crear el Manejador Global (GlobalExceptionHandler.cs)

Crea esta clase en tu capa de presentación (Presentation.Api/Middlewares/GlobalExceptionHandler.cs). Se encargará de interceptar las excepciones, registrar el error en los logs y formatear la salida JSON exacta que espera el cliente.

```
C#
using Microsoft.AspNetCore.Diagnostics;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using FluentValidation;

namespace Presentation.Api.Middlewares;

public class GlobalExceptionHandler : IExceptionHandler
{
    private readonly ILogger<GlobalExceptionHandler> _logger;

    public GlobalExceptionHandler(ILogger<GlobalExceptionHandler> logger)
    {
        _logger = logger;
    }
}
```

```

public async ValueTask<bool> TryHandleAsync(
    HttpContext httpContext,
    Exception exception,
    CancellationToken cancellationToken)
{
    // 1. Registramos el error completo en el sistema de logs (Console, Serilog, etc.)
    _logger.LogError(exception, "Ocurrió un error no controlado durante la petición: {Message}",
exception.Message);

    var problemDetails = new ProblemDetails
    {
        Instance = httpContext.Request.Path
    };

    // 2. Evaluamos el tipo de excepción para personalizar la respuesta RFC 7807
    switch (exception)
    {
        case ValidationException validationException:
            httpContext.Response.StatusCode = StatusCodes.Status400BadRequest;
            problemDetails.Status = StatusCodes.Status400BadRequest;
            problemDetails.Title = "Error de Validación";
            problemDetails.Detail = "Se encontraron uno o más errores en los datos enviados.";
            problemDetails.Type = "https://tools.ietf.org/html/rfc7231#section-6.5.1";

            // Agrupamos los errores por propiedad para enviarlos ordenados al cliente
            var errors = validationException.Errors
                .GroupBy(e => e.PropertyName)
                .ToDictionary(g => g.Key, g => g.Select(e => e.ErrorMessage).ToArray());

            problemDetails.Extensions["errors"] = errors;
            break;

        case KeyNotFoundException:
            httpContext.Response.StatusCode = StatusCodes.Status404NotFound;
            problemDetails.Status = StatusCodes.Status404NotFound;
            problemDetails.Title = "Recurso No Encontrado";
            problemDetails.Detail = exception.Message;
            problemDetails.Type = "https://tools.ietf.org/html/rfc7231#section-6.5.4";
            break;

        default:
            // Para errores de base de datos no controlados u otros fallos críticos del sistema
            httpContext.Response.StatusCode = StatusCodes.Status500InternalServerError;
            problemDetails.Status = StatusCodes.Status500InternalServerError;
            problemDetails.Title = "Error Interno del Servidor";
            problemDetails.Detail = "Ocurrió un error inesperado al procesar la solicitud. Intente más
tarde.";
            problemDetails.Type = "https://tools.ietf.org/html/rfc7231#section-6.6.1";
            break;
    }
}

```

```

// 3. Escribimos la respuesta utilizando el formato estándar application/problem+json
httpContext.Response.ContentType = "application/problem+json";
await httpContext.Response.WriteAsJsonAsync(problemDetails, cancellationToken);

// Retornamos 'true' para indicar que la excepción fue completamente manejada
return true;
}
}

```

🔧 Paso 2: Registrar el Manejador en el Program.cs

Integra el manejador en el contenedor de dependencias y activa el middleware en la tubería HTTP de tu API:

```

C#
var builder = WebApplication.CreateBuilder(args);

// Inyectar capas de la arquitectura
builder.Services.AddApplication();
builder.Services.AddInfrastructure(builder.Configuration);

// 📌 REGISTRAR EL MANEJADOR GLOBAL DE EXCEPCIONES Y PROBLEM DETAILS
builder.Services.AddExceptionHandler<GlobalExceptionHandler>();
builder.Services.AddProblemDetails();

builder.Services.AddControllers();
var app = builder.Build();

// 📌 ACTIVAR EL MIDDLEWARE DE EXCEPCIONES AL INICIO DE LA TUBERÍA
app.UseExceptionHandler();

app.UseAuthorization();
app.MapControllers();

app.Run();

```

📦 ¿Cómo luce la respuesta JSON generada (RFC 7807)?

Gracias a este diseño, si un cliente envía una telemetría con una temperatura inválida (provocando el fallo en FluentValidation que vimos antes), el sistema responderá automáticamente con un código **400 Bad Request** y un JSON estructurado como este:

```

JSON
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "Error de Validación",
  "status": 400,
  "detail": "Se encontraron uno o más errores en los datos enviados.",
  "instance": "/api/telemetry",
  "errors": {

```

```
"Temperature": [  
  "La temperatura debe estar en un rango industrial seguro entre -50°C y 150°C."  
]  
}  
}
```

Este formato estandarizado permite que cualquier cliente frontend (React, Angular, una aplicación móvil o un dispositivo IoT) interprete el error de forma predecible y muestre los mensajes de validación correspondientes en la interfaz de usuario.