BUZZWatch

V.0.1 BETA

**СЪДЪРЖАНИЕ**

1. **BUZZWatch v0.1 BETA – Въведение**  
   1.1. Главна цел  
   1.2. Основни задачи на проекта
2. **Архитектурен преглед**  
   2.1. Логически слоеве  
   2.2. Поток на данните  
   2.3. Ползи от архитектурата  
   2.4. Типични сценарии за ползване
3. **Детайлна структура на проектите**  
   3.1. 📗 BuzzWatch.Domain – бизнес-ядро  
     3.1.1. Роля и характеристики  
     3.1.2. Каталог на папките  
     3.1.3. Базови класове  
     3.1.4. Основни Entity-та  
     3.1.5. Value Objects  
     3.1.6. Domain Events  
     3.1.7. Инварианти и валидиране  
     3.1.8. Чести капани

3.2. 📘 BuzzWatch.Application – CQRS слой  
  3.2.1. Място в слоевете  
  3.2.2. Основни принципи  
  3.2.3. Структура на папките  
  3.2.4. Команди и Handler-и  
  3.2.5. Queries  
  3.2.6. Интерфейси към Infrastructure  
  3.2.7. Pipeline behaviors  
  3.2.8. DI регистрация  
  3.2.9. Тестване и чести грешки

3.3. ⚙️ BuzzWatch.Infrastructure – технологична реализация  
  3.3.1. Роля и обхват  
  3.3.2. NuGet зависимости  
  3.3.3. Структура на папките  
  3.3.4. ApplicationDbContext  
  3.3.5. Fluent API конфигурации  
  3.3.6. Репозитории и Unit-of-Work  
  3.3.7. Domain-event dispatcher  
  3.3.8. Hosted services (AlertEngine)  
  3.3.9. Dependency Injection  
  3.3.10. Миграции, логове, telemetry  
  3.3.11. Чести капани

3.4. 🔗 BuzzWatch.Contracts – DTO слой  
  3.4.1. Причина за отделен проект  
  3.4.2. Типове проекти и зависимости  
  3.4.3. Организация на DTO-тата  
  3.4.4. Версиониране и JSON атрибути  
  3.4.5. Mapping и употреба  
  3.4.6. Тестове за сериализация

3.5. 🌐 BuzzWatch.Api – REST gateway  
  3.5.1. Роля и функции  
  3.5.2. Шаблон и зависимости  
  3.5.3. Файлова структура  
  3.5.4. Program.cs – конфигурация  
  3.5.5. Endpoint groups и примери  
  3.5.6. Аутентикация / авторизация  
  3.5.7. Swagger / OpenAPI  
  3.5.8. Грешко-обработка  
  3.5.9. Dev/Prod настройки и чести грешки

3.6. 💻 BuzzWatch.Web – MVC UI  
  3.6.1. Цел и потребителски роли  
  3.6.2. Зависимости и шаблон  
  3.6.3. Папки, контролери и изгледи  
  3.6.4. Аутентикация / авторизация  
  3.6.5. HttpClient и SignalR интеграция  
  3.6.6. Администраторска зона  
  3.6.7. Производителност и локализация  
  3.6.8. Тестове и чести грешки

3.7. ⏳ BuzzWatch.Worker – фонови услуги  
  3.7.1. Мотивация за отделен процес  
  3.7.2. Задачи: Alert Engine, Aggregates, House-keeping  
  3.7.3. Quartz конфигурация  
  3.7.4. Outbox dispatcher  
  3.7.5. Разгръщане и мониторинг  
  3.7.6. Чести грешки

3.8. ✅ BuzzWatch.Tests – тестови пакети  
  3.8.1. Видове тестове (Unit / Integration / E2E)  
  3.8.2. Зависимости и инструменти  
  3.8.3. Папкова структура и примери  
  3.8.4. CI интеграция и покритие  
  3.8.5. Чести капани

1. **Архитектурни предимства**  
   4.1. Чисто разделение на отговорности  
   4.2. Обратно-съвместим публичен слой  
   4.3. Фонов engine, който не товари UI  
   4.4. Пълно автоматизирано тестване
2. **Заключение** – BUZZWatch като мащабируема, надеждна и лесно-разширяема IoT платформа

**📌 Общ преглед на проекта BUZZWatch**

**🌟 Главна цел**

**BUZZWatch** е система за мониторинг на състоянието на кошери в реално време. Системата събира и обработва данни за вътрешна и външна температура, влажност и тегло на кошерите. Данните се визуализират ясно в потребителски интерфейс, управляван през уеб-базиран MVC панел.

Системата включва автоматизирана алармена функционалност, която уведомява потребителите при критични промени в условията на кошерите.

**🚀 Основни задачи на проекта**

* Да събира надеждно данни от IoT устройства (кошери)
* Да визуализира събраните данни по ясен и разбираем начин
* Да управлява потребителски акаунти, роли и достъп до ресурси
* Да уведомява при критични събития (аларми) автоматично
* Да осигурява лесна интеграция с други системи и услуги

**📚 Подробно описание на всеки проект (компонент)**

**1. 📗 BuzzWatch.Domain**

**Тип:** Class Library

**Роля:**

Дефинира основните обекти и правила на системата. Представлява бизнес логиката, която не се променя при технически промени на проекта.

**Съдържание:**

* Entity-класове (напр. Device, MeasurementHeader, AlertRule, AlertEvent, ApiKey)
* Value Objects (напр. Temperature, Humidity, Weight)
* Бизнес правила и валидиране на данни (без технически зависимости)

**Зависимости:**

Никакви (независим слой).

**2. 📘 BuzzWatch.Application**

**Тип:** Class Library (MediatR + CQRS)

**Роля:**

Организира взаимодействията с бизнес логиката (Domain). Дефинира и изпълнява бизнес операции.

**Съдържание:**

* Commands и Queries (напр. CreateMeasurementCommand, GetMeasurementsQuery)
* Command и Query Handlers (CQRS)
* Валидация на заявки (FluentValidation)
* DTO мапинги (AutoMapper/Mapster)

**Зависимости:**

* BuzzWatch.Domain

**3. ⚙️ BuzzWatch.Infrastructure**

**Тип:** Class Library

**Роля:**

Реализира конкретните технологични детайли, нужни за работа с данни и външни услуги.

**Съдържание:**

* EF Core: ApplicationDbContext и конфигурации (базата данни MSSQL)
* ASP.NET Identity: Потребители, роли, права
* Реализации на интерфейси от Application слоя (например, изпращане на email/SMS)
* Фонова услуга (алармен работник), която следи и управлява алармите

**Зависимости:**

* BuzzWatch.Domain
* BuzzWatch.Application
* Външни пакети: EF Core, Identity, MailKit и др.

**4. 🔗 BuzzWatch.Contracts *(опционален, но препоръчителен)***

**Тип:** Class Library

**Роля:**

Съдържа общи DTO-та и типове данни, използвани за комуникация между API и UI.

**Съдържание:**

* Data Transfer Objects (DTO-та), например DeviceDto, MeasurementDto, AlertDto

**Зависимости:**

Никакви

**5. 🌐 BuzzWatch.Api**

**Тип:** ASP.NET Core Web API (Minimal API)

**Роля:**

REST API сървър за приемане на телеметрия от IoT устройствата и предоставяне на данни към клиентските приложения.

**Съдържание:**

* Minimal API endpoints
* Middleware за аутентикация (JWT / ApiKey)
* Swagger/OpenAPI за документация на endpoints
* Използва MediatR handlers от Application-слоя

**Зависимости:**

* BuzzWatch.Application
* BuzzWatch.Infrastructure
* BuzzWatch.Contracts

**6. 💻 BuzzWatch.Web**

**Тип:** ASP.NET Core MVC

**Роля:**

Потребителският интерфейс, достъпен през браузър. Потребителите виждат графики, получават известия и управляват ресурсите си.

**Съдържание:**

* MVC Controllers и Razor Views (Dashboard, Devices, Alerts)
* Identity login / регистрация
* Клиентски JavaScript за live ъпдейти (чрез SignalR)
* HttpClient за извикване на API endpoints от **BuzzWatch.Api**

**Зависимости:**

* BuzzWatch.Application
* BuzzWatch.Infrastructure (Identity)
* BuzzWatch.Contracts

**7. ⏳ BuzzWatch.Worker**

**Тип:** Worker Service (.NET)

**Роля:**

Изпълнява периодични задачи (напр. проверка на алармени състояния, обработка на данни на заден фон).

**Съдържание:**

* HostedService (BackgroundWorker) за аларми, агрегация на данни, почистване и архивиране на стари данни

**Зависимости:**

* BuzzWatch.Application
* BuzzWatch.Infrastructure

**8. ✅ BuzzWatch.Tests**

**Тип:** xUnit Test Project

**Роля:**

Осигурява качеството на системата чрез автоматизирани тестове.

**Съдържание:**

* Unit тестове на бизнес логика и CQRS handlers
* Интеграционни тестове за API endpoints и база данни

**Зависимости:**

* BuzzWatch.Domain
* BuzzWatch.Application
* BuzzWatch.Infrastructure (по желание)

**📌 Как комуникират проектите**

BuzzWatch.Web (UI MVC)

│

│ HTTP (REST)

▼

BuzzWatch.Api (REST API)

│

▼ MediatR CQRS

BuzzWatch.Application

│

▼

BuzzWatch.Infrastructure

│

▼

MSSQL Database

▲

│

BuzzWatch.Worker

**🔑 Ползи от тази архитектура:**

* **Чистота и яснота** – Всеки проект има ясна и ограничена роля.
* **Гъвкавост** – Лесно добавяне на нови функции и промяна на съществуващи.
* **Поддръжка** – Лесна локализация на бъгове и модификации.
* **Тестируемост** – Лесно автоматизиране и тестване.

**🎯 Очаквани сценарии за ползване от потребители:**

* Потребител влиза, наблюдава своите кошери на Dashboard.
* Системата автоматично генерира аларми при опасни условия.
* Администратор добавя нови потребители и устройства.
* Устройства изпращат данни чрез Web API до базата.
* Потребителите анализират данни и правят експорти.

Тази архитектура и разделение на проекти създават система, която е лесна за мащабиране, поддръжка и развитие, като същевременно осигурява висока надеждност и сигурност.

## **Проект BuzzWatch.Domain — детайлно ръководство**

*(„какво“, а не „как“)*

**1. Роля в архитектурата — домейн-слоят**

| **Характеристика** | **Обяснение** |
| --- | --- |
| **Бизнес истина** | Описва всички съществителни (Entity-та) и правила, валидни за пчеларския домейн — без да знае за база, HTTP, UI, IoT протоколи. |
| **Независим** | Няма NuGet пакети за EF Core, ASP.NET или Serilog. Позволени са само леки зависимости (напр. System.\*, FluentResults, MediatR за Domain Events). |
| **Тестируем** | Може да се unit-тества без база, защото не съдържа инфраструктура. |
| **Устойчив** | Промени във върховните слоеве (UI, API) не чупят домейн-кода. |

**2. Препоръчителна структура на папките**

BuzzWatch.Domain

├─ Common

│ ├─ BaseEntity.cs

│ ├─ ValueObject.cs

│ └─ DomainEvent.cs

├─ Devices

│ ├─ Device.cs

│ └─ HiveLocation.cs (Value Object)

├─ Measurements

│ ├─ MeasurementHeader.cs

│ ├─ MeasurementTempIn.cs

│ ├─ MeasurementHumIn.cs

│ ├─ MeasurementTempOut.cs

│ ├─ MeasurementHumOut.cs

│ └─ MeasurementWeight.cs

├─ Alerts

│ ├─ AlertRule.cs

│ └─ AlertEvent.cs

├─ Security

│ └─ ApiKey.cs

└─ Enums

└─ ComparisonOperator.cs

**Tip:** Папките отразяват *Ubiquitous Language* — „Devices“, „Measurements“, „Alerts“.

**3. Общи базови класове**

// BaseEntity.cs

public abstract class BaseEntity<TKey>

{

public TKey Id { get; protected set; } = default!;

private readonly List<IDomainEvent> \_domainEvents = new();

public IReadOnlyCollection<IDomainEvent> DomainEvents => \_domainEvents;

protected void AddDomainEvent(IDomainEvent evt) => \_domainEvents.Add(evt);

public void ClearDomainEvents() => \_domainEvents.Clear();

}

// ValueObject.cs (simplified)

public abstract record ValueObject;

// DomainEvent.cs

public interface IDomainEvent { DateTimeOffset OccurredOn { get; } }

public record MeasurementCreatedEvent(long MeasurementId, Guid DeviceId) : IDomainEvent

{

public DateTimeOffset OccurredOn => DateTimeOffset.UtcNow;

}

**4. Основни Entity-та**

**4.1 Device**

public class Device : BaseEntity<Guid>

{

private Device() { } // EF / сериализация

public string Name { get; private set; } = default!;

public HiveLocation? Location { get; private set; }

public DateTimeOffset CreatedAt { get; private set; }

private readonly List<MeasurementHeader> \_measurements = new();

public IReadOnlyCollection<MeasurementHeader> Measurements => \_measurements;

public static Device Create(string name, HiveLocation? location = null)

{

if (string.IsNullOrWhiteSpace(name))

throw new ArgumentException("Device name required", nameof(name));

return new Device

{

Id = Guid.NewGuid(),

Name = name.Trim(),

Location = location,

CreatedAt = DateTimeOffset.UtcNow

};

}

}

*Инварианти:* име не може да е празно; идентификатор се генерира веднъж.

**4.2 MeasurementHeader + тесни метрик-класове**

public class MeasurementHeader : BaseEntity<long>

{

private MeasurementHeader() { }

public Guid DeviceId { get; private set; }

public DateTimeOffset RecordedAt { get; private set; }

public MeasurementTempIn? TempIn { get; private set; }

public MeasurementHumIn? HumIn { get; private set; }

public MeasurementTempOut? TempOut { get; private set; }

public MeasurementHumOut? HumOut { get; private set; }

public MeasurementWeight? Weight { get; private set; }

public static MeasurementHeader Create(Guid deviceId, DateTimeOffset at) =>

new()

{

DeviceId = deviceId,

RecordedAt = at

};

public void AttachTempInside(decimal valueC)

{

TempIn = new MeasurementTempIn(Id, valueC);

}

}

public sealed record MeasurementTempIn(long Id, decimal ValueC)

{

public const decimal Min = -40m;

public const decimal Max = 60m;

public MeasurementTempIn : this

{

if (ValueC is < Min or > Max)

throw new ArgumentOutOfRangeException(nameof(ValueC), "Out of range");

}

}

*Всеки тесен клас съдържа свои* ***граници*** *и валидиране — това е бизнес логика, не инфраструктура.*

**4.3 AlertRule**

public class AlertRule : BaseEntity<Guid>

{

private AlertRule() { }

public Guid DeviceId { get; private set; }

public string Metric { get; private set; } = default!; // e.g. "TempInsideC"

public ComparisonOperator Operator { get; private set; }

public decimal Threshold { get; private set; }

public int DurationSeconds { get; private set; }

public bool Active { get; private set; } = true;

public DateTimeOffset CreatedAt { get; private set; }

public static AlertRule Create(Guid deviceId,

string metric,

ComparisonOperator op,

decimal threshold,

int durationS)

{

if (durationS < 0) throw new ArgumentOutOfRangeException(nameof(durationS));

return new AlertRule

{

Id = Guid.NewGuid(),

DeviceId = deviceId,

Metric = metric,

Operator = op,

Threshold = threshold,

DurationSeconds = durationS,

CreatedAt = DateTimeOffset.UtcNow

};

}

public void Disable() => Active = false;

}

**5. Value Objects (пример)**

public record HiveLocation(string Address, double? Latitude, double? Longitude)

{

public override string ToString() => Address;

}

*Value Object-ите са* ***immutable****; сравнение по стойност.*

**6. Domain Events — защо?**

* Отделят **страничните ефекти** (изпращане на e-mail, запис в лог) от чистия модел.
* Излъчват се в Application слоя чрез MediatR Pipeline.

Пример: MeasurementCreatedEvent се излъчва в момент, в който ново измерване е валидно и добавено.

**7. Валидация на инварианти**

* **Хвърляй изключение** в конструктор/фабрика, ако данните нарушават правилата.
* Поставяй **const Min/Max** стойности до всяка метрика, за да улесниш четимостта.

**8. Без EF Core атрибути!**

* Мапингът към SQL се прави във **BuzzWatch.Infrastructure** чрез Fluent API (OnModelCreating).
* Домейн слой остава чист.

**9. Unit тестове (в BuzzWatch.Tests)**

[Fact]

public void CreateDevice\_ShouldThrow\_WhenNameEmpty()

{

Assert.Throws<ArgumentException>(() => Device.Create(""));

}

Тестваш модели и бизнес-правила без база.

**10. Често срещани капани**

| **Грешка** | **Корекция** |
| --- | --- |
| Писане на DbContext тук | Премести в Infrastructure |
| Добавяне на [Key] / [Table] атрибути | Използвай Fluent API във външния слой |
| Сериализация логика (JSON) | Вмъкни я в API/Web, не в Domain |

**TL;DR**

* **BuzzWatch.Domain** = граматиката и правилата на пчеларския мониторинг.
* Съдържа *само* бизнес-обекти, техните инварианти и (по желание) Domain Events.
* Няма EF, няма HTTP, няма UI.
* Той е фундаментът — останалите слоеве (Application, Infrastructure, Api, Web) се изграждат около него, без да го променят.

**Проект BuzzWatch.Application — пълно ръководство**

*(“как” се изпълняват бизнес-сценариите, стъпка между домейна и външния свят)*

**1. Мястото му в слоя**

| **Слой** | **Отговорност** | **Зависимости** |
| --- | --- | --- |
| **Domain** | *Какво е* устройство, измерване, аларма | – |
| **Application** | *Какво правим* с тях: **use-case-и** (създай измерване, вземи графика, отвори аларма) | Domain, (леки) външни пакети |
| **Infrastructure / Hosts** | *Как го правим* с конкретни технологии – EF Core, API, MVC, Worker | Application |

**2. Основни принципи**

* **CQRS + MediatR**
  + *Command* → модифицира състояние.
  + *Query* → връща данни, но не пише.
  + Всеки Command/Query има **Handler** (един файл, един отговор).
* **Slim Application** – само координатор, няма EF контекст; работи с *интерфейси*.
* **Pipeline Behaviors** – AOP за валидация, логове, performance metering.
* **FluentValidation** – правило “навън”; не дублира инварианти от Domain.
* **Mapping** – Domain ↔ DTO (Mapster / AutoMapper) в Handler-а или във външен профил.

**3. Папки и файлове (пример)**

BuzzWatch.Application

├─ Abstractions

│ ├─ IDeviceRepository.cs

│ ├─ IMeasurementRepository.cs

│ ├─ IDateTimeProvider.cs

│ └─ IEmailSender.cs

├─ Common

│ ├─ Result.cs (FluentResults или собствен)

│ ├─ PaginatedList.cs

│ └─ ConstMetrics.cs

├─ Devices

│ ├─ Commands

│ │ ├─ CreateDeviceCommand.cs

│ │ └─ Handlers/CreateDeviceHandler.cs

│ ├─ Queries

│ │ ├─ GetDeviceListQuery.cs

│ │ └─ Handlers/GetDeviceListHandler.cs

│ └─ Validators

│ └─ CreateDeviceCommandValidator.cs

├─ Measurements

│ ├─ Commands

│ │ └─ CreateMeasurementCommand.cs

│ ├─ Queries

│ │ ├─ GetLatestMeasurementQuery.cs

│ │ └─ GetRangeMeasurementsQuery.cs

│ └─ Mapping

│ └─ MeasurementMappings.cs

├─ Alerts

│ ├─ Commands

│ │ └─ CreateAlertRuleCommand.cs

│ └─ Queries

│ └─ GetOpenAlertsQuery.cs

└─ Pipeline

├─ ValidationBehavior.cs

├─ LoggingBehavior.cs

└─ PerformanceBehavior.cs

Файлове < 300 LOC, лесни за четене и тестове.

**4. Команди (пример)**

public record CreateMeasurementCommand(

Guid DeviceId,

DateTimeOffset RecordedAt,

decimal? TempIn,

decimal? HumIn,

decimal? TempOut,

decimal? HumOut,

decimal? Weight) : IRequest<Result<long>>;

* **IRequest<T>** от MediatR – връща Result<long> (Id на header-а).
* Никакъв EF, никакъв JSON.

**Handler (съкращен)**

public class CreateMeasurementHandler :

IRequestHandler<CreateMeasurementCommand, Result<long>>

{

private readonly IMeasurementRepository \_repo;

private readonly IUnitOfWork \_uow;

private readonly IDateTimeProvider \_clock;

public CreateMeasurementHandler(IMeasurementRepository repo,

IUnitOfWork uow,

IDateTimeProvider clock)

{

\_repo = repo;

\_uow = uow;

\_clock = clock;

}

public async Task<Result<long>> Handle(CreateMeasurementCommand cmd,

CancellationToken ct)

{

// 1. създаваме header

var header = MeasurementHeader.Create(cmd.DeviceId, cmd.RecordedAt);

// 2. добавяме метрики

if (cmd.TempIn is not null)

header.AttachTempInside(cmd.TempIn.Value);

// ... за останалите

// 3. репо + Unit-of-Work

await \_repo.AddAsync(header, ct);

await \_uow.SaveChangesAsync(ct);

// 4. излъчване на Domain Event автоматично през UoW

return Result.Ok(header.Id);

}

}

**5. Queries (read-only)**

public record GetLatestMeasurementQuery(Guid DeviceId)

: IRequest<MeasurementDto>;

Handler взема DTO директно чрез проекция (Select new MeasurementDto { … }) от репозитория, без да мапва цели ентитети.

**6. Интерфейсите към Infrastructure**

| **Интерфейс** | **Защо го има** |
| --- | --- |
| **IRepository** (Device, Measurement…) | Скрива EF/SQL; позволява мокове в тестове |
| **IUnitOfWork** | Гарантира \* transactional consistency\* за няколко репо- операции |
| **IDateTimeProvider** | Тестове с фиксиран часовник |
| **IEmailSender / INotificationService** | Алерти без твърда зависимост към SMTP/Push |
| **IAlertEvaluator** | Логика за сверяване на правила – може да живее в Worker-а |

Infrastructure дава реалните имплементации; в тестовете – In-Memory/Stub.

**7. Pipeline Behaviors**

public sealed class ValidationBehavior<TReq,TRes>

: IPipelineBehavior<TReq,TRes>

where TReq : notnull

{

private readonly IEnumerable<IValidator<TReq>> \_validators;

public ValidationBehavior(IEnumerable<IValidator<TReq>> validators) =>

\_validators = validators;

public async Task<TRes> Handle(TReq request,

RequestHandlerDelegate<TRes> next,

CancellationToken ct)

{

var ctx = new ValidationContext<TReq>(request);

var failures = \_validators

.Select(v => v.Validate(ctx))

.SelectMany(r => r.Errors)

.Where(f => f != null)

.ToList();

if (failures.Count != 0)

throw new ValidationException(failures);

return await next();

}

}

Регистрира се веднъж в DI и работи за всички заявки.

**8. Dependency Injection (регистрация)**

public static class ServiceCollectionExtensions

{

public static IServiceCollection AddApplication(this IServiceCollection s)

{

// MediatR

s.AddMediatR(cfg => cfg.RegisterServicesFromAssemblyContaining<CreateMeasurementHandler>());

// FluentValidation

s.AddValidatorsFromAssembly(typeof(CreateDeviceCommandValidator).Assembly);

// Pipeline behaviors

s.AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehavior<,>));

s.AddTransient(typeof(IPipelineBehavior<,>), typeof(LoggingBehavior<,>));

s.AddTransient(typeof(IPipelineBehavior<,>), typeof(PerformanceBehavior<,>));

// Mapster (или AutoMapper)

TypeAdapterConfig.GlobalSettings.Scan(typeof(MeasurementMappings).Assembly);

return s;

}

}

Хостовете (Api, Web, Worker) просто извикват builder.Services.AddApplication();

**9. Тестване**

public class CreateMeasurementTests

{

[Fact]

public async Task Should\_Save\_Header\_And\_Metrics()

{

var repo = new InMemoryMeasurementRepo();

var uow = new FakeUow(repo);

var now = new FakeClock();

var handler = new CreateMeasurementHandler(repo, uow, now);

var cmd = new CreateMeasurementCommand(deviceId, now.UtcNow,

35.2m, null, null, null, 42.5m);

var result = await handler.Handle(cmd, CancellationToken.None);

result.IsSuccess.Should().BeTrue();

repo.Headers.Count.Should().Be(1);

repo.Weight.Count.Should().Be(1);

}

}

*Не ти трябва SQL или API – тестът покрива бизнес-пътя.*

**10. Често срещани грешки**

| **Грешка** | **Как да я избегнеш** |
| --- | --- |
| Инжектираш DbContext директно в Handler | Използвай Repository / Unit-of-Work интерфейси |
| Валидация в контролера | Премести я във FluentValidation + Pipeline |
| Мапинг логика в домейн | Дръж Mapster/AutoMapper по-навън (Application) |
| Запис на лог/e-mail в Handler | Пусни Domain Event и го улови в Infrastructure |

**TL;DR**

**BuzzWatch.Application** е *оркестраторът* на всички бизнес-случаи:

1. Приема Command/Query DTO-та.
2. Валидира ги чрез Pipeline.
3. Извиква домейн-логика и добавя/чете чрез *абстрактни* репо-интерфейси.
4. Връща резултат / хвърля контролирани грешки.

Така UI-то и API-то разполагат с **единен, чист и тестируем слой**, а всички технически детайли остават в *Infrastructure*.

**Проект BuzzWatch.Infrastructure — ˮкабинетът с кабелитеˮ**

*(технологичната реализация на всички портове, които Domain + Application изискват)*

**1. Роля в архитектурата**

| **Какво** | **Каква е работата му** |
| --- | --- |
| **Persist** | Записва и чете данни през **EF Core** (SQL Server). |
| **Identity** | Хоства ASP.NET Identity таблиците и логиката за акаунти/роли. |
| **Adapters** | Дава *конкретни* реализации за интерфейсите от Application – репозитории, Unit-of-Work, clock, e-mail/SMS, файлове. |
| **Background** | Пуска Hosted Services (напр. AlertEngine) за периодични задачи. |
| **Infra-cross‐cutting** | Конфигурира логове (Serilog), кеширане, outbox, транзакционни интерсептори. |

⚠️ НИКОГА не съдържа Razor, контролери или UI код.

**2. NuGet пакети (минимум)**

Microsoft.EntityFrameworkCore.SqlServer

Microsoft.EntityFrameworkCore.Design

Microsoft.AspNetCore.Identity.EntityFrameworkCore

MediatR // за домейн-събития (notification handler)

Serilog / Serilog.Sinks.Console

MailKit (ако ще пращаме e-mail)

**3. Папки и файлове — препоръка**

BuzzWatch.Infrastructure

├─ Data

│ ├─ ApplicationDbContext.cs

│ ├─ Configurations (EF Fluent API)

│ │ ├─ DeviceConfig.cs

│ │ ├─ MeasurementHeaderConfig.cs

│ │ ├─ MeasurementTempInConfig.cs

│ │ └─ ...

│ └─ Extensions

│ └─ ModelBuilderExtensions.cs (seed roles, etc.)

├─ Identity

│ └─ AppUser.cs (extends IdentityUser<Guid>)

├─ Repositories

│ ├─ DeviceRepository.cs

│ ├─ MeasurementRepository.cs

│ └─ UnitOfWork.cs

├─ Services

│ ├─ DateTimeProvider.cs

│ ├─ EmailSender.cs

│ └─ AlertEngineHostedService.cs

├─ Events

│ └─ DomainEventDispatcher.cs

└─ DependencyInjection

└─ InfrastructureServiceCollectionExtensions.cs

**4. ApplicationDbContext (съкратен)**

public class ApplicationDbContext :

IdentityDbContext<AppUser, IdentityRole<Guid>, Guid>

{

public DbSet<Device> Devices => Set<Device>();

public DbSet<MeasurementHeader> Measurements => Set<MeasurementHeader>();

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> opt)

: base(opt) { }

protected override void OnModelCreating(ModelBuilder b)

{

base.OnModelCreating(b); // Identity

b.ApplyConfigurationsFromAssembly(GetType().Assembly);

}

// Hook за Domain Events ➜ MediatR (outbox)

public override async Task<int> SaveChangesAsync(CancellationToken ct = default)

{

var events = ChangeTracker

.Entries<BaseEntity<Guid>>()

.SelectMany(e => e.Entity.DomainEvents)

.ToList();

var result = await base.SaveChangesAsync(ct);

await \_dispatcher.DispatchAsync(events, ct);

return result;

}

private readonly IDomainEventDispatcher \_dispatcher;

}

**5. Fluent API конфигурация (пример)**

public class MeasurementTempInConfig : IEntityTypeConfiguration<MeasurementTempIn>

{

public void Configure(EntityTypeBuilder<MeasurementTempIn> e)

{

e.ToTable("MeasurementTempIn");

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

e.Property(t => t.ValueC)

.HasPrecision(4,1)

.IsRequired();

e.HasOne<MeasurementHeader>()

.WithOne(h => h.TempIn)

.HasForeignKey<MeasurementTempIn>(t => t.Id)

.OnDelete(DeleteBehavior.Cascade);

}

}

*Тук е мястото за HasIndex, default-и, партициониране (SQL script), FK-delete правила.*

**6. Репозитории и UoW (пример)**

public class DeviceRepository : IDeviceRepository

{

private readonly ApplicationDbContext \_db;

public DeviceRepository(ApplicationDbContext db) => \_db = db;

public Task<Device?> GetAsync(Guid id, CancellationToken ct) =>

\_db.Devices.FirstOrDefaultAsync(d => d.Id == id, ct);

public void Add(Device d) => \_db.Devices.Add(d);

}

public class UnitOfWork : IUnitOfWork

{

private readonly ApplicationDbContext \_db;

public UnitOfWork(ApplicationDbContext db) => \_db = db;

public Task<int> SaveChangesAsync(CancellationToken ct) =>

\_db.SaveChangesAsync(ct);

}

**7. Domain Event Dispatcher**

public class DomainEventDispatcher : IDomainEventDispatcher

{

private readonly IMediator \_mediator;

public DomainEventDispatcher(IMediator m) => \_mediator = m;

public Task DispatchAsync(IEnumerable<IDomainEvent> events, CancellationToken ct)

{

var tasks = events.Select(e => \_mediator.Publish(e, ct));

return Task.WhenAll(tasks);

}

}

Позволява Application/Worker да слушат за MeasurementCreatedEvent и да стартират алармената проверка **без** да бъркат в DbContext.

**8. Hosted Service — AlertEngine**

public class AlertEngineHostedService : BackgroundService

{

private readonly IServiceProvider \_sp;

private readonly ILogger<AlertEngineHostedService> \_log;

public AlertEngineHostedService(IServiceProvider sp, ILogger<AlertEngineHostedService> log) =>

(\_sp, \_log) = (sp, log);

protected override async Task ExecuteAsync(CancellationToken ct)

{

while (!ct.IsCancellationRequested)

{

using var scope = \_sp.CreateScope();

var evaluator = scope.ServiceProvider.GetRequiredService<IAlertEvaluator>();

await evaluator.ProcessAsync(ct);

await Task.Delay(TimeSpan.FromSeconds(30), ct);

}

}

}

*Регистрираш го в AddHostedService<AlertEngineHostedService>() — може да живее и в отделния* ***Worker*** *проект, ако предпочиташ.*

**9. Регистрация в DI**

public static class InfrastructureServiceCollectionExtensions

{

public static IServiceCollection AddInfrastructure(

this IServiceCollection services,

IConfiguration cfg)

{

// DbContext

services.AddDbContext<ApplicationDbContext>(o =>

o.UseSqlServer(cfg.GetConnectionString("Default")));

// Identity

services.AddIdentityCore<AppUser>(opt =>

{

opt.Password.RequireDigit = true;

})

.AddRoles<IdentityRole<Guid>>()

.AddEntityFrameworkStores<ApplicationDbContext>();

// Repositories & UoW

services.AddScoped<IDeviceRepository, DeviceRepository>();

services.AddScoped<IMeasurementRepository, MeasurementRepository>();

services.AddScoped<IUnitOfWork, UnitOfWork>();

// Infra services

services.AddSingleton<IDateTimeProvider, DateTimeProvider>();

services.AddTransient<IEmailSender, EmailSender>();

// Domain events dispatcher

services.AddScoped<IDomainEventDispatcher, DomainEventDispatcher>();

// Background

services.AddHostedService<AlertEngineHostedService>();

return services;

}

}

Хостовете (Api, Web, Worker) извикват само:

builder.Services

.AddInfrastructure(builder.Configuration)

.AddApplication();

**10. Миграции и база**

dotnet ef migrations add Initial \

--project BuzzWatch.Infrastructure \

--startup-project BuzzWatch.Api

dotnet ef database update --startup-project BuzzWatch.Api

* **EF Migrations** живеят в Infrastructure, защото тук е DbContext.
* Можеш да добавиш IInterceptor за soft-delete, аудит timestamps.

**11. Logging & Telemetry**

* **Serilog** конфигуриран в Infrastructure (enrichment, sinks).
* **OpenTelemetry** може да се включи с AddOpenTelemetry().WithSqlClientInstrumentation().

**12. Чести капани**

| **Грешка** | **Какво да правиш** |
| --- | --- |
| Използваш Entity-класове в Controllers | Всички IO типове трябва да са DTO (в **Contracts**). |
| Бизнес-валидация в EF конфигурация | Дръж я в Domain / FluentValidation. |
| Static DateTime.UtcNow в Handler | Инжектирай IDateTimeProvider от Infrastructure. |
| Логика в миграционни SQL скриптове | Използвай Seed/Background job или Domain Events. |

**TL;DR**

**BuzzWatch.Infrastructure** е “връзката с контакта”:

* Дава **ApplicationDbContext** + EF Fluent мапване
* Домъква **Identity** (потребители/роли)
* Имплементира всички абстракции (репота, UoW, e-mail, clock)
* Стартира Hosted Services (аларми, архив)
* Грижи се за логове, миграции и транзакции

Всичко, което е **техноло­гично** и може да се смени без да пипа бизнес-правилата, живее тук.  
  
  
**Проект BuzzWatch.Contracts — „общият речник“ на системата**

*(споделени типове за обмен между всички хостове)*

**1. Защо изобщо има отделен Contracts проект?**

| **Проблем без Contracts** | **Решение с Contracts** |
| --- | --- |
| UI и API директно реферират Entity-класовете от **Domain** ➜ изтичане на вътрешни полета, крехки зависимости. | Вместо това обменяме **DTO** обекти, дефинирани в Contracts. |
| При промяна на EF/Domain трябва да прекомпилираме UI + мобилни клиенти. | Contracts се поддържа **Backward-Compatible** – изолираме счупванията. |
| JSON-модел ≠ Entity-модел (напр. комбинирани полета, форматиране). | DTO-тата от Contracts описват *точно* JSON‐формата. |

**TL;DR:** Contracts е *публичният API-модел*, домейнът е *вътрешният*.

**2. Какъв тип проект е това?**

* **Class Library (.NET 8)**
* Без препратки към EF Core, MediatR, ASP.NET – **чисти POCO/record класове**.
* Допустими леки пакети: System.Text.Json, Swashbuckle.Annotations за [JsonConverter], [SwaggerSchema] ако желаеш.

**3. Организация на папките/файловете**

BuzzWatch.Contracts

├─ Devices

│ ├─ DeviceDto.cs

│ └─ DeviceCreateRequest.cs

├─ Measurements

│ ├─ MeasurementDto.cs

│ ├─ MeasurementCreateRequest.cs

│ └─ MeasurementChartPoint.cs

├─ Alerts

│ ├─ AlertRuleDto.cs

│ ├─ AlertEventDto.cs

│ └─ CreateAlertRuleRequest.cs

├─ Auth

│ ├─ LoginRequest.cs

│ ├─ LoginResponse.cs

│ └─ ApiKeyDto.cs

└─ Common

├─ PagedResult.cs

└─ ProblemDetailsExtensions.cs

**4. Примери за DTO / Request / Response**

**4.1 DeviceDto**

public record DeviceDto

{

public Guid Id { get; init; }

public string Name { get; init; } = default!;

public string? Location { get; init; }

public DateTimeOffset CreatedAt { get; init; }

}

**4.2 MeasurementCreateRequest**

public record MeasurementCreateRequest

(

DateTimeOffset RecordedAt,

decimal? TempInsideC,

decimal? HumInsidePct,

decimal? TempOutsideC,

decimal? HumOutsidePct,

decimal? WeightKg

);

**Важно:**

* *Request* класовете представят JSON входа от клиента.
* *Dto* / *Response* класовете — JSON, който API връща обратно.

**5. PagedResult helper**

public record PagedResult<T>(IReadOnlyList<T> Items,

int TotalCount,

int Page,

int Size);

Използва се от почти всички GET /list endpoints; Web UI знае как да рендерира стандартно.

**6. Версиониране**

* **Contracts** e централната библиотека за версиониране на API:
  1. v1 namespace (папка) – стабилни типове.
  2. При breaking change създаваш нова папка/namespace v2, задържаш v1 за съвместимост.
* Swagger ще показва две версии; Web клиентът може постепенно да мигрира.

**7. JSON атрибути**

public record AlertRuleDto

{

[JsonPropertyName("id")]

public Guid Id { get; init; }

[JsonPropertyName("metric")]

public string Metric { get; init; } = default!;

[JsonConverter(typeof(JsonStringEnumConverter))]

public ComparisonOperator Operator { get; init; }

public decimal Threshold { get; init; }

public int DurationSeconds { get; init; }

}

* Contracts могат да съдържат JsonPropertyName, JsonConverter – така контролираш wire-формата **само тук**.

**8. Mapping: Domain ↔ Contracts**

| **Къде се случва** | **Инструмент** |
| --- | --- |
| **Application** или **Api** | AutoMapper / Mapster профил |

TypeAdapterConfig<Device,DeviceDto>.NewConfig();

TypeAdapterConfig<MeasurementHeader,MeasurementDto>.NewConfig()

.Map(d => d.TempInsideC, s => s.TempIn!.ValueC);

UI не знае за Entity-структурата; вижда само DeviceDto, MeasurementDto.

**9. Употреба в проектите**

// Api endpoint

app.MapPost("/api/v1/measurements",

async (MeasurementCreateRequest req, IMediator mediator) =>

{

var cmd = req.Adapt<CreateMeasurementCommand>();

var id = await mediator.Send(cmd);

return Results.Created($"/api/v1/measurements/{id}", new { id });

});

// MVC controller

public async Task<IActionResult> Details(Guid id)

{

var dto = await \_api.GetAsync<DeviceDto>($"/api/v1/devices/{id}");

return View(dto);

}

**10. Тестове за сериализация (по избор)**

[Fact]

public void MeasurementRequest\_Should\_Serialize\_To\_SnakeCase()

{

var req = new MeasurementCreateRequest(DateTimeOffset.UtcNow, 12.3m, null, null, null, 7.8m);

var json = JsonSerializer.Serialize(req, MyJson.Options);

json.Should().Contain("\"tempInsideC\":12.3");

}

*Държим Contracts стабилно – бързо научаваме, ако JSON-форматът се е счупил.*

**11. Избягване на грешки**

| **Антипатърн** | **Правилно** |
| --- | --- |
| Изнасяш Entity от Domain към UI | Изнасяш **DTO** от Contracts |
| Добавяш логика/валидация в DTO | DTO-то е пасивно; валидацията е във Validator / Domain |
| Contracts реферира EF или MediatR | Contracts остава чисто POCO |

**TL;DR**

* **BuzzWatch.Contracts** = *здравият публичен JSON/DTO слой* – един източник на истина за формата на съобщенията.
* Използва се от **Api** (за вход/изход) и от **Web** (за консумация).
* Дава независимост на UI и API от вътрешната Entity-структура и от инфраструктурните промени.

**Проект BuzzWatch.Api — публичният REST-сървър на системата**

*(входна точка за IoT-устройствата и всички външни клиенти)*

**1. Роля в цялата архитектура**

| **Функция** | **Какво означава** |
| --- | --- |
| **Приема телеметрия** | IoT устройствата изпращат POST заявки; API валидира, препраща към Application слоя и записва. |
| **Излага данни** | UI-то (MVC), мобилно приложение или външни интеграции теглят JSON чрез GET/POST. |
| **Сигурност** | Защита чрез JWT за интерактивни потребители и **API Key** за устройства. |
| **Контракт** | Swagger/OpenAPI описва всеки endpoint; версия на API се поддържа през URL (/api/v1/…). |

**2. Какъв шаблон използваме**

* **ASP.NET Core Web API**
  + опция --use-minimal-apis → по-малко ceremony, четливи lambda-ендпойнти
  + включен AddEndpointsApiExplorer() и AddSwaggerGen()

**3. Зависимости на проекта**

| **NuGet пакет** | **Защо е нужен** |
| --- | --- |
| **MediatR** | Изпраща командите/заявките към Application. |
| **FluentValidation.AspNetCore** | Автоматична Model-валидация. |
| **Swashbuckle.AspNetCore** | Swagger UI + OpenAPI JSON. |
| **AspNetCore.Authentication.JwtBearer** | JWT за Identity cookie → token flow. |
| (custom) **AspNetCore.Authentication.ApiKey** | Лек middleware за устройство-ключове. |

В проектните reference-и: **BuzzWatch.Application**, **BuzzWatch.Infrastructure**, **BuzzWatch.Contracts**.

**4. Файлова структура (пример)**

BuzzWatch.Api

├─ Endpoints

│ ├─ DeviceEndpoints.cs

│ ├─ MeasurementEndpoints.cs

│ ├─ AlertEndpoints.cs

│ └─ AuthEndpoints.cs

├─ Middleware

│ └─ ApiKeyMiddleware.cs

├─ Extensions

│ ├─ EndpointGroups.cs

│ └─ SwaggerOptions.cs

└─ Program.cs

**5. Program.cs — най-важното**

var builder = WebApplication.CreateBuilder(args);

builder.Services

.AddInfrastructure(builder.Configuration) // DbContext, Identity, repos

.AddApplication() // MediatR, validators, pipeline

.AddAuthentication(opt =>

{

opt.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

opt.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(JwtOptions.Build(builder.Configuration))

.AddApiKeySupport(); // custom extension

builder.Services.AddAuthorization();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen(SwaggerOptions.Configure);

var app = builder.Build();

app.UseSwagger().UseSwaggerUI();

app.UseAuthentication();

app.UseAuthorization();

app.MapBuzzWatchEndpoints(); // групирано регистриране

app.Run();

**6. Регистриране на endpoint-ите (Minimal API)**

public static class EndpointGroups

{

public static void MapBuzzWatchEndpoints(this IEndpointRouteBuilder r)

{

var v1 = r.MapGroup("/api/v1").WithOpenApi();

DeviceEndpoints.Map(v1);

MeasurementEndpoints.Map(v1);

AlertEndpoints.Map(v1);

AuthEndpoints.Map(v1);

}

}

**7. Примерен endpoint - Measurements**

internal static class MeasurementEndpoints

{

public static void Map(RouteGroupBuilder g)

{

g.MapPost("/devices/{deviceId:guid}/measurements",

[Authorize(AuthenticationSchemes = ApiKeyDefaults.AuthenticationScheme)]

async (Guid deviceId,

MeasurementCreateRequest body,

IMediator mediator,

CancellationToken ct) =>

{

var cmd = body with { DeviceId = deviceId }

.Adapt<CreateMeasurementCommand>(); // Mapster

var result = await mediator.Send(cmd, ct);

return result.IsSuccess

? Results.Created($"/api/v1/measurements/{result.Value}", new { id = result.Value })

: Results.BadRequest(result.Errors);

})

.Produces(201)

.ProducesProblem(400)

.WithName("CreateMeasurement");

}

}

**Ключови моменти**

1. **[Authorize]** – този POST работи само с **API-key** схема.
2. **Mapster** или AutoMapper – мапира DTO → Command.
3. **Mediator** – препраща към Application слоя, който вече записва.

**8. Аутентикация и авторизация**

* **JWT** (Bearer)
  + Получава се от POST /api/v1/auth/login → връща токен ± refresh.
  + Използва се от MVC UI, мобилен клиент, Postman.
* **API Key** (header X-Api-Key)
  + Валидира се в custom middleware (проверява таблица ApiKey, ExpiresAt, Scope).
  + Перфектно за еднопосочно изпращане от устройство (няма clock skew проблеми).
* **Policies**
  + "OwnsDevice" → проверява дали JWT-потребителят има ред в UserDevice за deviceId.
  + "Admin"/"Moderator" роли от Identity за админ endpoints.

**9. Swagger / OpenAPI**

public static void Configure(SwaggerGenOptions o)

{

o.SwaggerDoc("v1", new() { Title = "BUZZWatch API", Version = "v1" });

o.AddSecurityDefinition("ApiKey", new()

{

In = ParameterLocation.Header,

Name = "X-Api-Key",

Type = SecuritySchemeType.ApiKey

});

o.AddSecurityDefinition("Bearer", new()

{

Type = SecuritySchemeType.Http,

Scheme = JwtBearerDefaults.AuthenticationScheme,

BearerFormat = "JWT"

});

o.OperationFilter<SecurityRequirementsOperationFilter>();

}

* **Swagger UI** достъпен само в Development (може да сложиш UseSwaggerUI(c => c.RoutePrefix = "docs")).
* Отбелязва изискваните схеми (ApiKey/Bearer) за всяка операция.

**10. Обработка на грешки**

app.UseExceptionHandler(exApp =>

{

exApp.Run(async ctx =>

{

var feat = ctx.Features.Get<IExceptionHandlerPathFeature>()!;

var problem = new ProblemDetails

{

Title = "Server error",

Detail = feat.Error.Message,

Status = 500

};

ctx.Response.StatusCode = 500;

ctx.Response.WriteAsJsonAsync(problem);

});

});

* **ValidationException** от FluentValidation → 400 с ProblemDetails.
* **NotFoundException** (кастъм) → 404.

**11. Тестове (BuzzWatch.Tests)**

* **WebApplicationFactory** → ин-мемори хост за API unit/integration.
* Тест: POST measurement и assert DB count.

**12. Dev/Prod настройки**

| **Околна среда** | **Особени настройки** |
| --- | --- |
| **Development** | Swagger, CORS *localhost*, dotnet watch. |
| **Staging/Prod** | Swagger може да е зад BasicAuth; HTTPS enforced; HSTS. |
| **Docker** | Dockerfile multi-stage: dotnet publish → slim runtime image (alpine). |

**13. Често срещани грешки**

| **Грешка** | **Решение** |
| --- | --- |
| Регистрация на DbContext и Application два пъти | Използвай разширенията AddInfrastructure / AddApplication по веднъж. |
| Мапваш Entity към JSON директно | Винаги DTO от **Contracts**. |
| Забравен ApiKey header → 401 | Swagger „Authorize“ с X-Api-Key: {guid} за тестове. |
| Clock skew за JWT | Конфигурирай ClockSkew = TimeSpan.FromMinutes(1) в JwtBearer. |

**TL;DR**

**BuzzWatch.Api** е лекият, документиран gateway:

1. Проверява *кой* (JWT/ApiKey)
2. Валидира *какво* (FluentValidation)
3. Обръща се към Application (MediatR)
4. Връща стандартизиран JSON (ProblemDetails, DTO-та)

Всичко инфраструктурно (DbContext, Identity) идва през DI, а бизнес-логиката се намира в **Application**/**Domain**.  
  
**Проект BuzzWatch.Web — уеб-интерфейсът за хората**

*(ASP.NET 8* ***MVC*** *+ Razor + Identity cookie)*

**1. Цел и място в системата**

| **Роля** | **Подробност** |
| --- | --- |
| **UI** | Показва измерванията в графики, таблици, карти; позволява CRUD върху устройства, аларми, потребители. |
| **Портал за роли** | • **Admin** – управлява потребители, роли, кошери, правила.• **Moderator** – вижда всичко, но с ограничени промени.• **User** – вижда и редактира *само* „своите“ устройства (чрез UserDevice). |
| **Auth фронт** | Използва ASP.NET Identity cookie (Forms) ⇒ single-sign-on за UI; Web извиква API с **Bearer JWT**, получен чрез silent flow. |
| **Live dashboard** | Получава push-данни чрез SignalR хъб от API или Worker за „живи“ графики без рефреш. |

**2. Шаблон и проекти зависимости**

* **Шаблон „ASP.NET Core Web App (Model-View-Controller)“**
  + --auth Individual → scaffold-нати Register/Login/Logout Razor Pages.
* **References**: BuzzWatch.Application (за Mediator, ако желаем direct use), BuzzWatch.Infrastructure (за Identity store), BuzzWatch.Contracts (DTO), плюс NuGet Microsoft.AspNetCore.SignalR.Client.

**3. Папки и съдържание**

BuzzWatch.Web

├─ Controllers

│ ├─ DashboardController.cs

│ ├─ DeviceController.cs

│ ├─ MeasurementController.cs

│ ├─ AlertController.cs

│ └─ AdminController.cs

├─ Models (ViewModels / DTO wrappers)

│ ├─ DashboardVm.cs

│ ├─ DeviceDetailsVm.cs

│ └─ AlertListVm.cs

├─ Services

│ ├─ ApiClient.cs (HttpClient wrapper)

│ └─ SignalRClient.cs (live updates)

├─ Views

│ ├─ Shared

│ │ ├─ \_Layout.cshtml

│ │ ├─ \_NavBar.cshtml

│ │ └─ \_ValidationScriptsPartial.cshtml

│ ├─ Dashboard

│ │ └─ Index.cshtml

│ ├─ Device

│ │ ├─ List.cshtml

│ │ └─ Details.cshtml

│ ├─ Alert

│ │ └─ Index.cshtml

│ └─ Admin

│ ├─ Users.cshtml

│ └─ Rules.cshtml

├─ TagHelpers

│ └─ DeviceStatusTagHelper.cs (онлайн/офлайн бейдж)

├─ wwwroot

│ ├─ css (Tailwind / Bootstrap)

│ └─ js (Chart.js, SignalR, custom)

└─ Program.cs

**4. Аутентикация / авторизация**

builder.Services.AddAuthentication(opt =>

{

opt.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;

})

.AddCookie(opt =>

{

opt.LoginPath = "/Account/Login";

opt.AccessDeniedPath = "/Account/Denied";

});

builder.Services.AddAuthorization(opts =>

{

opts.AddPolicy("IsOwner", p => p.Requirements.Add(new OwnsDeviceRequirement()));

opts.AddPolicy("AdminOnly", p => p.RequireRole("Admin"));

});

builder.Services.AddSingleton<IAuthorizationHandler, OwnsDeviceHandler>();

* **Login** и **Register** Razor Pages остават scaffold-нати.
* След успешен вход MVC UI *тихо* вика /api/v1/auth/token да получи JWT, пази го в session IHttpContextAccessor или IJSRuntime (localStorage) за Ajax повиквания.

**5. HttpClient → API**

builder.Services.AddHttpClient<ApiClient>(c =>

{

c.BaseAddress = new(cfg["Api:BaseUrl"]); // https://api.buzzwatch.local

})

.AddHttpMessageHandler<JwtDelegatingHandler>(); // добавя Bearer JWT

ApiClient има помощни методи:

public async Task<T?> GetAsync<T>(string url) =>

await \_http.GetFromJsonAsync<T>(url);

public async Task<Result<Guid>> PostAsync<TReq>(string url, TReq body) { … }

**6. Пример: DashboardController**

[Authorize]

public class DashboardController : Controller

{

private readonly ApiClient \_api;

public DashboardController(ApiClient api) => \_api = api;

public async Task<IActionResult> Index()

{

var devices = await \_api.GetAsync<IReadOnlyList<DeviceDto>>("/api/v1/devices");

var alerts = await \_api.GetAsync<IReadOnlyList<AlertEventDto>>("/api/v1/alerts/open");

var vm = new DashboardVm(devices!, alerts!);

return View(vm);

}

}

View използва Chart.js компоненти за визуализацията.

**7. SignalR за live-ъпдейти**

public class SignalRClient

{

private readonly HubConnection \_conn;

public event Action<MeasurementDto>? MeasurementReceived;

public SignalRClient(NavigationManager nav, ITokenProvider tok)

{

\_conn = new HubConnectionBuilder()

.WithUrl(nav.ToAbsoluteUri("/hubs/measurements"),

opt => opt.AccessTokenProvider = tok.GetTokenAsync)

.WithAutomaticReconnect()

.Build();

\_conn.On<MeasurementDto>("MeasurementAdded",

dto => MeasurementReceived?.Invoke(dto));

}

public Task StartAsync() => \_conn.StartAsync();

}

UI-то добавя точките в графиката в реално време.

**8. View примери (Tailwind + Razor)**

<div class="grid md:grid-cols-3 gap-4">

@foreach (var d in Model.Devices)

{

<div class="card">

<h2>@d.Name</h2>

<span device-status device-id="@d.Id"></span> <!-- custom tag helper -->

<canvas id="chart-@d.Id" width="320" height="160"></canvas>

</div>

}

</div>

DeviceStatusTagHelper добавя клас „bg-green-500“ за online.

**9. Администраторска зона (Areas)**

Areas/Admin/Controllers/UserController.cs

Areas/Admin/Views/User/Index.cshtml

[Area("Admin")] + [Authorize(Roles="Admin")].

**10. Обработка на грешки и validation**

* MVC ModelState + ValidationScriptsPartial.
* Ако API връща ProblemDetails, JavaScript показва toast с подробност.
* Globally handle HttpRequestException → лог-аут ако 401.

**11. Локализация (по желание)**

builder.Services.AddLocalization();

app.UseRequestLocalization(opt =>

{

opt.SetDefaultCulture("bg");

opt.AddSupportedCultures("bg","en");

});

Razor @Localizer["Temperature"].

**12. Бъндлинги / производителност**

* **BundlerMinifier** или Vite да пакетират Tailwind/JS.
* Response caching за статични ресурси, gzip enabled.

**13. Тестове на UI слоя**

* **Playwright** integration:
  1. Логин. 2. Очаквай „Hive-01“ tile. 3. Изкуствено инжектирай нова стойност чрез SignalR mock → assert графиката се обнови.
* **Razor Pages unit tests** (bUnit) – за TagHelper-и.

**14. Чести грешки**

| **Симптом** | **Причина / Fix** |
| --- | --- |
| CORS error между Web :5002 и API :5001 | Enable CORS в Api за https://localhost:5002. |
| 401 при Ajax, но UI е логнат | JWT в localStorage е изтекъл – добави refresh flow или Sliding cookie. |
| Чупи се при offline устройства | Контролер филтрира по IsOwner policy; върни 404 вместо 500. |
| Дублира DTO код | Изнасяй в **Contracts**. |

**TL;DR**

**BuzzWatch.Web** е „таблото за управление“:

1. **Cookie Auth** → Identity UI
2. **HttpClient + JWT** → комуникира с API
3. **Razor MVC** показва устройства, измервания, аларми
4. **SignalR** вкарва live данни
5. **Role-based** менюта (Admin, Moderator, User)

Той е *чист клиент* – никакви EF/DbContext, цялата бизнес-логика идва от **BuzzWatch.Api** и **Application**.

**Проект BuzzWatch.Worker — “невидимият” бекграунд-двигател**

*(.NET 8* ***Worker Service****) – изпълнява задачи, които не трябва да блокират уеб-заявки*

**1. Защо е отделен процес?**

| **Причина** | **Обяснение** |
| --- | --- |
| **Непрекъснати задачи** | Алармено проверяване, агрегации, чистене на стари данни работят 24/7, без да товарят API или UI. |
| **Независимо мащабиране** | Ако имаш 1 млн. измервания/час, пускаш 3 реплики Worker, а API остава 1-2 екземпляра. |
| **Без тайм-аут на HTTP** | Дълги операции (експорт 100 MB CSV, месечен архив) вървят извън web-request контекста. |
| **Crash-група** | Ако Worker падне, front-end остава онлайн; в Kubernetes се рестартира отделно. |

**2. Основни отговорности**

| **Категория** | **Подробности** |
| --- | --- |
| **Alert Engine** | На всеки N секунди сканира нови измервания → проверява AlertRule → отваря/затваря AlertEvent и изпраща уведомления. |
| **Continuous Aggregates** | Изчислява 5-минутни / 1-часови усреднени стойности и ги записва в отделни таблици/Materialized View. |
| **Data Housekeeping** | • Триe сурови данни по retention политика.• Компресира стари партиции.• Триe изтекли ApiKey. |
| **Outbox Dispatcher** *(по желание)* | Чете outbox таблица и публикува събития към RabbitMQ/Azure Service Bus. |
| **E-mail Queue** | Забавени/повтарящи се опити при SMTP грешка; не спира API. |

**3. Шаблон и зависимости**

dotnet new worker -n BuzzWatch.Worker

**Project References**

* BuzzWatch.Application (за MediatR, DTO)
* BuzzWatch.Infrastructure (DbContext, repos, Identity)

**NuGet**

* Microsoft.Extensions.Hosting (включен)
* Microsoft.Extensions.Logging (Serilog)
* Quartz.Extensions.Hosting *(ако искаш cron-стил job schedule)*
* MediatR (publish Domain Events)

**4. Файлова структура (пример)**

BuzzWatch.Worker

├─ Services

│ ├─ AlertEngine.cs

│ ├─ AggregateJob.cs

│ └─ MaintenanceJob.cs

├─ Schedulers

│ └─ QuartzConfigurator.cs

├─ Hosted

│ └─ OutboxDispatcher.cs

├─ Extensions

│ └─ ServiceCollectionExtensions.cs

└─ Program.cs

**5. Program.cs (съкратен)**

var builder = Host.CreateApplicationBuilder(args);

builder.Services

.AddInfrastructure(builder.Configuration) // DbContext, Identity

.AddApplication() // MediatR

.AddWorkerJobs(builder.Configuration); // наш custom DI

builder.Services.AddHostedService<OutboxDispatcher>();

using var host = builder.Build();

await host.RunAsync();

**6. Регистрация на задачи**

public static class WorkerDi

{

public static IServiceCollection AddWorkerJobs(

this IServiceCollection s, IConfiguration cfg)

{

// Quartz scheduler

s.AddQuartz(q =>

{

q.SchedulerId = "buzzwatch-scheduler";

q.UseMicrosoftDependencyInjectionJobFactory();

q.ScheduleJob<AlertEngineJob>(t =>

t.WithIdentity("alert-engine")

.WithSimpleSchedule(x => x

.WithIntervalInSeconds(30)

.RepeatForever()));

q.ScheduleJob<AggregateJob>(t =>

t.WithIdentity("aggregate-hourly")

.WithCronSchedule("0 5 \* \* \* ?")); // 5 мин след всеки час

q.ScheduleJob<MaintenanceJob>(t =>

t.WithIdentity("housekeeping-daily")

.WithCronSchedule("0 0 2 \* \* ?")); // ежедневно 02:00

});

s.AddQuartzHostedService(opts => opts.WaitForJobsToComplete = true);

return s;

}

}

**7. Реализация на AlertEngineJob**

public class AlertEngineJob : IJob

{

private readonly ILogger<AlertEngineJob> \_log;

private readonly IAlertEvaluator \_evaluator;

public AlertEngineJob(ILogger<AlertEngineJob> log, IAlertEvaluator eval) =>

(\_log,\_evaluator) = (log, eval);

public async Task Execute(IJobExecutionContext ctx)

{

var count = await \_evaluator.ProcessAsync(ctx.CancellationToken);

\_log.LogInformation("AlertEngine processed {Count} measurements", count);

}

}

IAlertEvaluator живее в Infrastructure и прави:

1. Чете всички AlertRule ▶ Active=true.
2. Търси последните M измервания за даден device/metric.
3. Ако условието се държи DurationSeconds, създава/затваря AlertEvent.
4. Публикува AlertTriggeredEvent ➜ Outbox ➜ E-mail/SignalR.

**8. AggregateJob (пример)**

public class AggregateJob : IJob

{

private readonly ApplicationDbContext \_db;

private readonly ILogger<AggregateJob> \_log;

public AggregateJob(ApplicationDbContext db, ILogger<AggregateJob> log) =>

(\_db,\_log) = (db,log);

public async Task Execute(IJobExecutionContext ctx)

{

var from = DateTimeOffset.UtcNow.AddHours(-1);

var to = DateTimeOffset.UtcNow;

// Пример: hourly avg temp\_inside

var query = \_db.Measurements

.Where(m => m.RecordedAt >= from && m.RecordedAt < to)

.Select(m => new { m.DeviceId, m.TempIn!.ValueC });

var grouped = await query

.GroupBy(x => x.DeviceId)

.Select(g => new HourlyAggregate

{

DeviceId = g.Key,

Period = from.DateTime,

AvgTempC = g.Average(x => x.ValueC)

})

.ToListAsync(ctx.CancellationToken);

\_db.HourlyAggregates.AddRange(grouped);

await \_db.SaveChangesAsync(ctx.CancellationToken);

\_log.LogInformation("Aggregated {Count} devices for {Period}", grouped.Count, from);

}

}

**9. OutboxDispatcher (optional)**

Патерн *Outbox* гарантира, че събитията се изпращат **само ако** транзакцията с данните е успешна.

protected override async Task ExecuteAsync(CancellationToken ct)

{

while (!ct.IsCancellationRequested)

{

using var scope = \_provider.CreateScope();

var db = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();

var pending = await db.OutboxMessages

.Where(x => x.Processed == null)

.OrderBy(x => x.OccurredAt)

.Take(50).ToListAsync(ct);

foreach (var msg in pending)

{

try

{

await \_bus.PublishAsync(msg.Topic, msg.Payload, ct);

msg.Processed = DateTimeOffset.UtcNow;

}

catch (Exception ex)

{

\_log.LogWarning(ex, "Outbox dispatch failed for {Id}", msg.Id);

}

}

await db.SaveChangesAsync(ct);

await Task.Delay(1000, ct);

}

}

**10. Разгръщане / DevOps**

| **Среда** | **Настройка** |
| --- | --- |
| **Docker** | FROM mcr.microsoft.com/dotnet/aspnet:8.0-alpineENTRYPOINT ["dotnet","BuzzWatch.Worker.dll"] |
| **Kubernetes** | Deployment с replicas: 2, restartPolicy: Always, readiness probe /healthz |
| **Systemd** (on-prem) | dotnet BuzzWatch.Worker.dll под Type=notify за graceful shutdown |

**11. Мониторинг**

* **Serilog** → Loki/ELK за лога.
* **OpenTelemetry** → Prometheus: метрики job\_duration\_seconds, alert\_events\_total.
* Health endpoint (/healthz) – DbContext ping + pending outbox count.

**12. Чести грешки**

| **Симптом** | **Причина / Fix** |
| --- | --- |
| CPU spikes | Cron/loop твърде къс интервал без sleep, добави Task.Delay. |
| Дублирани AlertEvent | Забравена проверка за вече „open“ събитие. |
| Worker няма достъп до DB | Kubernetes secret / connection string не е маунтнат. |
| Умиращ HostedService | Wrapи кода в try/catch и логни; Quartz сам ще рестартира job-а. |

**TL;DR**

**BuzzWatch.Worker** е отделен процес, който:

1. **Сверява алармени правила** на всеки ~30 сек.
2. **Създава агрегати** (часови, дневни) за бърз UI.
3. **Чисти и архивира** данни според политики.
4. **Изпраща уведомления** и публикува домейн-събития от outbox.

Всичко това върви независимо от web-сървърите, подсигурявайки мащабируемост и стабилност на системата.  
  
  
**Проект BuzzWatch.Tests — гаранцията за качество**

*(xUnit-базирана тестова колекция, покриваща Domain → Application → API/UI)*

**1. Защо отделен тестов проект?**

| **Полза** | **Обяснение** |
| --- | --- |
| **Изолация** | Тестовете не попадат в production сборките — бърз билд и чисти зависимости. |
| **Модулност** | Можеш да имаш различни „test suites“ (unit, integration, e2e) в подпапки. |
| **CI** | GitHub Actions стартира само dotnet test BuzzWatch.Tests и отчита покритието. |

**2. Зависимости и пакети**

dotnet new xunit -n BuzzWatch.Tests

dotnet add reference BuzzWatch.Domain

dotnet add reference BuzzWatch.Application

dotnet add reference BuzzWatch.Infrastructure # за интеграционни

dotnet add package FluentAssertions

dotnet add package Microsoft.EntityFrameworkCore.InMemory

dotnet add package Respawn # reset DB между тестове

dotnet add package Microsoft.AspNetCore.Mvc.Testing

dotnet add package PlaywrightSharp # e2e UI (по избор)

**FluentAssertions** – четими асерции.  
**InMemory** – бърз DbContext за unit; за по-реалистично – Testcontainers SQL.

**3. Папкова структура**

BuzzWatch.Tests

├─ Unit

│ ├─ Domain

│ │ ├─ DeviceTests.cs

│ │ └─ MeasurementValidationTests.cs

│ └─ Application

│ ├─ CreateMeasurementHandlerTests.cs

│ └─ AlertRuleValidatorTests.cs

├─ Integration

│ ├─ RepositoryTests.cs

│ └─ Api

│ ├─ ApiFactory.cs

│ ├─ MeasurementEndpointTests.cs

│ └─ DeviceEndpointTests.cs

└─ E2E

└─ DashboardUiTests.cs (Playwright)

**4. Unit-тестове (Domain)**

public class DeviceTests

{

[Theory]

[InlineData("")]

[InlineData(" ")]

public void Create\_Should\_Throw\_When\_NameEmpty(string badName)

{

Action act = () => Device.Create(badName, null);

act.Should().Throw<ArgumentException>()

.WithMessage("\*name\*");

}

}

* Никаква база, никакъв MediatR — само чисти класове.

**5. Unit-тестове (Application Handler)**

public class CreateMeasurementHandlerTests

{

[Fact]

public async Task Returns\_Failure\_When\_Device\_NotFound()

{

var repo = Substitute.For<IDeviceRepository>(); // NSubstitute

repo.GetAsync(Arg.Any<Guid>(), default)

.Returns(Task.FromResult<Device?>(null));

var handler = new CreateMeasurementHandler(

repo, new FakeMeasurementRepo(),

Substitute.For<IUnitOfWork>(),

new FakeClock());

var cmd = new CreateMeasurementCommand(Guid.NewGuid(), DateTimeOffset.UtcNow,

35, null, null, null, null);

var result = await handler.Handle(cmd, default);

result.IsFailed.Should().BeTrue();

result.Errors.Should().Contain(e => e.Message.Contains("not found"));

}

}

**6. Интеграционни тестове (репо + DB)**

public class RepositoryTests : IAsyncLifetime

{

private readonly SqliteConnection \_sqlite;

private readonly ApplicationDbContext \_db;

private readonly DeviceRepository \_repo;

public RepositoryTests()

{

\_sqlite = new SqliteConnection("Filename=:memory:");

\_sqlite.Open();

var opt = new DbContextOptionsBuilder<ApplicationDbContext>()

.UseSqlite(\_sqlite).Options;

\_db = new ApplicationDbContext(opt);

\_db.Database.EnsureCreated();

\_repo = new DeviceRepository(\_db);

}

public async Task InitializeAsync()

{

// seed sample device

var d = Device.Create("Hive-A", null);

\_repo.Add(d);

await \_db.SaveChangesAsync();

}

[Fact]

public async Task Can\_Read\_Device\_By\_Id()

{

var device = await \_repo.GetAsync(\_db.Devices.First().Id, default);

device.Should().NotBeNull();

device!.Name.Should().Be("Hive-A");

}

public Task DisposeAsync() => \_sqlite.DisposeAsync().AsTask();

}

**7. Integration + API (WebApplicationFactory)**

public class MeasurementEndpointTests : IClassFixture<ApiFactory>

{

private readonly HttpClient \_client;

public MeasurementEndpointTests(ApiFactory factory)

{

\_client = factory.CreateClient(); // in-memory TestServer

}

[Fact]

public async Task Post\_Measurement\_Returns\_201()

{

var body = new MeasurementCreateRequest(

DateTimeOffset.UtcNow, 36.5m, null, null, null, 42.2m);

var resp = await \_client.PostAsJsonAsync("/api/v1/devices/11111111-1111-.../measurements", body);

resp.StatusCode.Should().Be(HttpStatusCode.Created);

}

}

**ApiFactory** наследява WebApplicationFactory<BuzzWatch.Api.Program> и override-ва DbContext към InMemory или Testcontainer SQL.

**8. End-to-End UI тестове (Playwright)**

[Fact]

public async Task Dashboard\_Shows\_Devices()

{

using var pw = await Playwright.CreateAsync();

var browser = await pw.Chromium.LaunchAsync(new BrowserTypeLaunchOptions{ Headless = true });

var page = await browser.NewPageAsync();

await page.GotoAsync("https://localhost:5002/Account/Login");

await page.FillAsync("#Input\_Email", "admin@demo.local");

await page.FillAsync("#Input\_Password", "Pa$$word1!");

await page.ClickAsync("button[type=submit]");

await page.WaitForSelectorAsync("text=Hive-01");

var temp = await page.InnerTextAsync("#latestTemp");

temp.Should().MatchRegex(@"[0-9]+\.[0-9] °C");

}

**9. Test fixtures и helpers**

* **FakeClock** – имплементира IDateTimeProvider → връща фиксирано време.
* **Substitute/Mock** – NSubstitute или Moq за лесно мокване на репо.
* **Respawn** – при integration: чисти DB между фактове без повторно EnsureCreated.

**10. CI интеграция**

.github/workflows/ci.yml

- name: Run unit & integration tests

run: dotnet test BuzzWatch.Tests --logger trx --results-directory ./TestResults

- name: Upload test results

uses: actions/upload-artifact@v4

with:

name: test-results

path: ./TestResults

Добави coverlet за code-coverage (dotnet test /p:CollectCoverage=true).

**11. Покритие по слоеве**

| **Слой** | **Покритие** | **Вид тест** |
| --- | --- | --- |
| **Domain** | ≥ 90 % | Unit |
| **Application** | 70 – 80 % | Unit (+ integration с InMemory) |
| **Infrastructure** | репозитории и DbContext конфигурация | Integration |
| **API** | Critical endpoints (POST, GET/latest) | Integration |
| **UI** | Smoke сценарии (Login, Dashboard load) | E2E |

**12. Чести капани**

| **Симптом** | **Решение** |
| --- | --- |
| Random failures от едновременни тестове | Използвай Testcontainers или уникални DB имена за всеки тест. |
| Тестовете бавни (> 1 s unit) | Увери се, че Domain/Application тестове не тръгват срещу SQL. |
| JSON camelCase mismatch | Тествай сериализацията с JsonSerializer в Contracts тест. |

**TL;DR**

**BuzzWatch.Tests** осигурява:

1. **Unit validation** на бизнес-правила (Domain + Application).
2. **Интеграционна гаранция** че EF мапинг/репо-та и API ендпойнти работят.
3. **E2E уверение** че истинският потребител вижда коректен UI.
4. Лесна интеграция в CI/CD – “червен build” при регресия.

**BUZZWatch — обобщение на цялата система и проектите в решението**

| **Ниво** | **Какво съдържа** | **Защо съществува** |
| --- | --- | --- |
| **Domаin** (📗) | Чисти Entity-класи, Value Objects, бизнес-инварианти, Domain Events | Една непроменлива „истина“ за кошера, независима от база, UI и протоколи |
| **Application** (📘) | CQRS + MediatR команди/заявки, валидатори, мапинг | Прилага use-case-ите (“създай измерване”, “вземи графика”), без да знае за SQL или HTTP |
| **Infrastructure** (⚙️) | EF Core DbContext + Identity, репозитории, Unit-of-Work, e-mail, Hosted Services | Реализира портовете на Application със SQL Server, SMTP, Serilog, Quartz |
| **Contracts** (🔗) | DTO-та и enum-и за обмен (JSON) | Единен, обратно-съвместим публичен модел между API и всички клиенти |
| **Api** (🌐) | ASP.NET Minimal API, Swagger, JWT + ApiKey | Вход за IoT устройствата и външни клиенти; маршрутизира към Application |
| **Web** (💻) | ASP.NET MVC + Razor + Identity cookie, SignalR | Човешки UI: dashboard, графики, управление на устройства/потребители/аларми |
| **Worker** (⏳) | .NET Worker Service, Quartz jobs | Фонови задачи — проверка на аларми, агрегиране, почистване, outbox |
| **Tests** (✅) | xUnit пакети: Unit, Integration, E2E (Playwright) | Автоматично валидиране на всички слоеве в CI |

**Поток на данните**

1. **Устройството** изпраща пакет → **Api** (ApiKey)
2. Api ⇢ **Application** (CreateMeasurementCommand) ⇢ **Infrastructure** (DbContext) ⇢ SQL
3. **Worker** сверява AlertRule, отваря AlertEvent, праща нотификация
4. **Web** (JWT) чете /api/v1/..., показва live-графики (SignalR)
5. **Tests** гарантират, че всичко горе работи при всяка промяна.

**Ключови предимства**

* **Чиста разделимост** по отговорности ⇒ лесна поддръжка и мащабиране
* **Обратно-съвместим контрак­тен слой** ⇒ API може да расте без да чупи клиенти
* **Фонов engine** ⇒ уеб-натоварването и алармите не си пречат
* **Пълно покритие с тестове** ⇒ уверено развитие и CI защита

Така BUZZWatch осигурява надеждно събиране, съхранение, визуализация и алармиране на кошерните данни – готово за реални потребители и голям брой устройства.