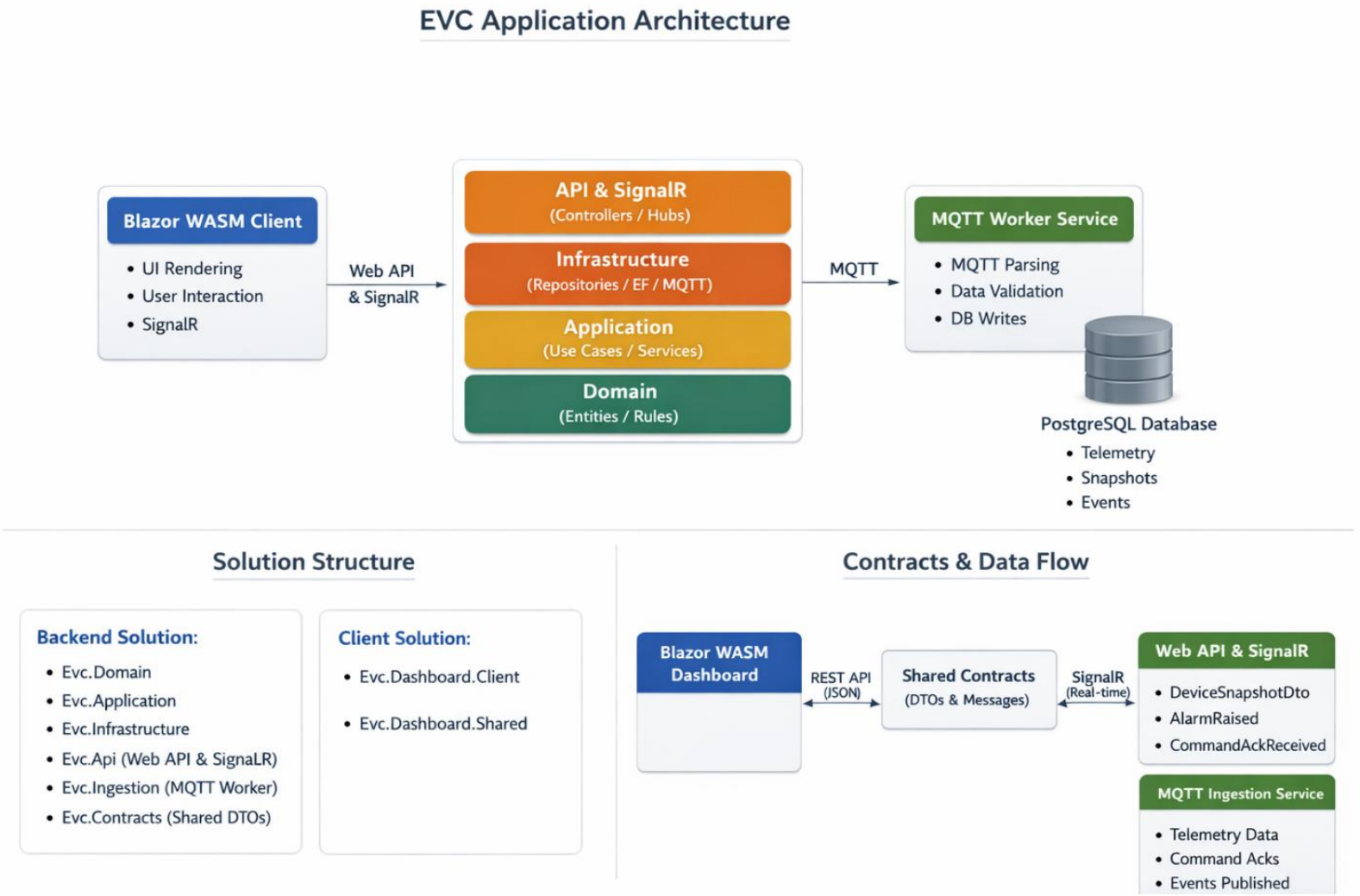
# Code guideline for EVC 2 Application




1. Client: Blazor WebAssembly dashboard (remote web app)
2. Server: ASP.NET Core Web API + SignalR
3. Ingestion: separate MQTT Worker Service dumping to PostgreSQL + notifying server

team engineering standard.

**1) Core Principles**

**1.1 Clean boundaries (non-negotiable)**

1. Client UI: rendering + user interaction only

2. Server API: auth, validation, business rules, querying, orchestration
3. Ingestion Worker: MQTT parsing, validation, normalization, DB writes, event publish
4. Database: source of truth (telemetry, snapshots, events, commands, users, audit)

Never put business rules inside UI components.
Never let UI write telemetry. Only ingestion writes telemetry.

**1.2 One direction of dependencies**

Use a layered structure (Clean Architecture style):

**Domain (Models,entities)** ← **Application (use cases, controllers)** ← **Infrastructure (repo)** ← **API/Worker/UI**

Domain has no dependency on EF, SignalR, MQTTnet, ASP.NET.

**2) Solution / Project Structure (recommended)**

**2.1 Backend solution layout**

Evc.sln

/src

Evc.Domain

Evc.Application

Evc.Infrastructure

Evc.Api (Web API + SignalR)

Evc.Ingestion (MQTT Worker)

Evc.Contracts (shared DTOs + SignalR message contracts)

/tests

Evc.UnitTests

Evc.IntegrationTests

**Why Evc.Contracts**
Keep DTOs/events shared between client and server stable and versionable.

**2.2 Client solution layout (Blazor WASM)**

Evc.Dashboard.sln

/src

Evc.Dashboard.Client

Evc.Dashboard.Shared  (shared UI utilities/components, optional)

**3) Naming Conventions (C#)**

**3.1 General naming**

1. Namespaces: Company.Product.Area → Rma.Evc.Telemetry
2. Types / Classes / Records: PascalCase
3. Methods / Properties: PascalCase
4. Local variables / params: camelCase
5. Private fields: _camelCase
6. Constants: PascalCase (public const int MaxRetries = 3;)
7. Interfaces: IName (e.g., IDeviceRepository)
8. Generics: T, TKey, TResult

**3.2 Async naming rules**

1. Async methods end with **Async**: GetDeviceAsync
2. Always accept CancellationToken ct
3. Don't write async void (except UI events)

**3.3 File rules**

1. One public type per file
2. File name matches type name: DeviceSnapshot.cs
3. Use file-scoped namespaces (modern C#)

**4) Code Style Rules (Team Wide)**

**4.1 Must-have settings**

1. Enable **Nullable Reference Types** everywhere: <Nullable>enable</Nullable>

2. Turn warnings to errors in CI:
   <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
3. Use .editorconfig + dotnet format in CI
4. Add analyzers:

   Microsoft.CodeAnalysis.NetAnalyzers

   (optional) StyleCop.Analyzers

**4.2 Clean code rules**

Methods ≤ ~30 lines (split if bigger)

No “god services” (if class > 500 lines → refactor)

Prefer immutability for contracts: use record DTOs

Prefer early returns, avoid deep nesting

No magic strings: centralize topic names, event names, etc.

**5) Backend Architecture Guidelines**

**5.1 Domain Layer (Evc.Domain)**

**Contains**

1. Entities: Device, Command, Alarm, Snapshot
2. Value objects: DeviceId, UtcTimestamp, QualityFlag
3. Domain rules (pure)

**Rules**

1. No EF Core attributes here
2. No logging here
3. No SignalR/MQTT here

**5.2 Application Layer (Evc.Application)**

**Contains**

1. Use-cases/services: CreateCommand, GetDeviceHistory
2. Interfaces: IDeviceRepository, ICommandPublisher, IClock

3. Validation: FluentValidation validators
4. Mapping: Mapster or AutoMapper (keep mapping in one place)

**Rules**

Only depends on Domain + Contracts

No direct EF/SignalR/MQTT usage (via interfaces)

**5.3 Infrastructure Layer (Evc.Infrastructure)**

**Contains**

EF Core DbContext + migrations

Repositories implementing interfaces

Redis / Postgres LISTEN/NOTIFY integration

MQTT publishing adapter (if needed)

**EF Core Rules**

Use IEntityTypeConfiguration<T> per entity

Avoid lazy loading

Use AsNoTracking() for reads

Always index (DeviceId, Timestamp) for telemetry

Keep a device_snapshot table for fast dashboard

**5.4 API Layer (Evc.Api) – Web API + SignalR**

**Rules**

1. Controllers are thin: validate → call Application → return DTO
2. No database code in controllers
3. Use ProblemDetails for errors (global exception middleware)

**API conventions**

1. Version your API: /api/v1/…
2. DTOs only (never expose EF entities directly)
3. Standard response envelope for paged results

**SignalR rules**

1. Hub is thin: auth + group join + simple forward
2. Do NOT put business logic in Hub methods
3. Use IHubContext<T> from services to broadcast

**Groups strategy**

1. tenant:{tenantId}
2. device:{deviceId}
3. site:{siteId}
   Broadcast to smallest group possible.

**6) MQTT Ingestion Worker Guidelines (Evc.Ingestion)**

**6.1 Structure**

Evc.Ingestion

/Mqtt

MqttClientFactory

TopicRouter

/Parsing

TelemetryParser

CommandAckParser

/Validation

TelemetryValidator

/Persistence

TelemetryWriter

SnapshotWriter

/Events

UpdateEventPublisher (Redis/Notify/HTTP)

**6.2 Rules**

1. Parsing must be strict (reject unknown schema versions unless handled)
2. Validation must include:

   timestamp sanity

   numeric range checks

   device authentication/identity
3. Writes must be idempotent where possible
4. Never block MQTT callback thread:
   - push to a Channel<T> / queue and process in background
5. Publish update event after DB commit (or use outbox)

**7) Client (Blazor WASM) Guidelines**

**7.1 Folder structure (feature-based)**

Evc.Dashboard.Client

/Features

/Live

LivePage.razor

Components/

Services/

/Reports

/Devices

/Alarms

/Shared

/Services

ApiClient/

Realtime/

Auth/

/State

### 7.2 Component naming rules

1. Pages: SomethingPage.razor
2. Components: DeviceCard.razor, AlarmTable.razor
3. Code-behind (optional): DeviceCard.razor.cs
4. Avoid huge pages: break into components

### 7.3 Client responsibilities

1. Render data
2. Call Web API for queries/commands
3. Connect SignalR for live updates
4. Local state only (filters, selections, UI state)

**Do not**

1. implement validation rules that differ from server
2. compute business-critical values on client

### 7.4 SignalR client wrapper (required)

Have one service:

- RealtimeConnectionService

    connect/reconnect

    join/leave groups

    throttle UI updates if needed

    expose events/streams to UI

## 8) Error Handling & Logging Standards

### 8.1 Logging

Use structured logging (recommended: **Serilog**).

Log fields: deviceId, tenantId, messageType, correlationId

Never log secrets, tokens, passwords, raw private payloads

Use log levels correctly:

Information: normal operations

Warning: recoverable issues

Error: failure needing attention

### 8.2 Exception strategy

Throw exceptions only for exceptional cases

Application layer returns a Result pattern where appropriate:

Result<T> / OneOf etc.

API maps failures to ProblemDetails

## 9) Security Guidelines (Minimum Baseline)

Use **JWT** for API + SignalR auth

Enforce RBAC: Admin / Operator / Auditor

Device auth: broker credentials + server-side device registry checks

Rate-limit critical endpoints (commands/login)

Validate all inputs server-side

Use wss everywhere; MQTT over TLS if possible

## 10) Database Guidelines (PostgreSQL)

### 10.1 Table naming

Use snake_case in DB

Use EF mapping to keep C# PascalCase

### 10.2 Telemetry strategy

Keep:

telemetry_history (append-only)

device_snapshot (1 row/device fast access)

Index:

(device_id, ts_utc DESC)

Partition large telemetry by time if volume grows

## 11) API Contract & Versioning

Contracts live in Evc.Contracts

Backward compatible changes only:

add optional fields (ok)

don’t rename/remove fields without version bump

SignalR message names stable:

DeviceSnapshotUpdated

AlarmRaised

CommandAckReceived

## 12) Testing & Quality Gates

### 12.1 Unit tests (fast)

Domain rules

Parsers

Validators

Use-cases

### 12.2 Integration tests

Postgres integration

Ingestion write + snapshot update

SignalR broadcast verification (basic)

### 12.3 CI requirements

dotnet format (or verify formatting)

dotnet build

dotnet test

analyzer warnings treated as errors

## 13) Code Review Checklist (use every PR)

1. No business logic in UI/Hub/Controller

2. Async + CancellationToken done right
3. DTOs used (no EF entity leakage)
4. Logging has context and no secrets
5. Validation server-side exists for new endpoints
6. DB queries use indexes / AsNoTracking for reads
7. SignalR broadcasts only needed payload (not huge objects)
8. Tests included for critical logic