



[Classi di base]

Sistema
Domotico su
piattaforma
Arduino



AUTORE: ANDREA LANDO

REVISIONE 1.0.0

DATA ULTIMA REVISIONE: 20.05.2026

Sommario

Sommario

Prefazione	5
Architettura Generale del Sistema	7
Visione ad Alto Livello.....	7
Il Ciclo Principale Deterministico	9
Il Ruolo Centrale del Buffer	9
È completamente definito tramite configurazione esterna	9
Engines: Architettura a Motori Indipendenti	9
Configurazione Dinamica e Parametrica	10
DeviceProfiles e Astrazione dell'Hardware	10
AEE – Automation Execution Engine	11
FrontendEngine — Orchestrazione del Frontend e Integrazione con DomoManager.....	11
Il Ruolo del FrontendEngine nel Sistema.....	11
Pipeline Operativa del Frontend.....	12
Analisi del Setup Reale (setup()).....	12
Sicurezza, Robustezza e Fail-Safe	16
Scalabilità e Estendibilità	16
Sintesi dell'Architettura	16
Analisi delle Classi.....	17
Classi di Orchestrazione Principale.....	17
Classe FrontendConfig.....	17
Classe DMFrontendEngine.....	20
Classe DomoManager	24
Classi di Dominio Funzionale	27
Classe WeatherEngine	27
Classe PowerSupervisorEngine	27
Classe PowerEngine	27
Classe DMHVAC	29
Classe WatchModule	33
Classi di gestione Servizi di Sistema e Infrastruttura	34
Classe DMFnCs	34
Classi di Gestione Sensori e Sicurezza	39
SecurityEngine – Gestione Avanzata dei Sensori di Sicurezza	39
Classe DMWiredSensors.....	40
Classi di Supporto e Servizi	41

Classe DMSignal	41
Classe ICommandTransport (DMAAdapters)	43
Classe OptaRTC	44
Classe DMPLC	48
Classe DMLogger	52
Classe DMRS485Node	55
Classe DMBridge	59
Classe AutomationEngine	61
Classe AutomationBuilder	62
Classe DMBaseClass	65
Classe Buffer	69
Classe DMAEE	73
Classe BridgeEngine	77
Classe GenericSensorReader	77
Classe Diagnostic	77
Classi di Attuatori e Driver Fisici	79
DMEquipment	79
Sintesi delle Dipendenze	82
Disclaimer Legale – Limitazione di Responsabilità	85

Prefazione

La progettazione e lo sviluppo di un sistema complesso come **DomoManager** richiedono un approccio metodico, strutturato e pienamente consapevole delle implicazioni architetturali, funzionali e operative che caratterizzano un ecosistema embedded moderno. L'obiettivo di questo documento è fornire una visione completa, rigorosa e coerente dell'intero insieme di classi che compongono il sistema, analizzandone la struttura, le responsabilità, le dipendenze e il ruolo all'interno dell'architettura generale.

In un contesto in cui l'affidabilità, la modularità e la scalabilità rappresentano requisiti imprescindibili, la documentazione tecnica assume un valore strategico: essa non è soltanto un supporto allo sviluppo, ma un vero e proprio strumento di governo dell'evoluzione del software. La capacità di comprendere a fondo le relazioni tra i moduli, la natura dei flussi informativi, i meccanismi di astrazione e le logiche di configurazione dinamica consente infatti di mantenere il sistema robusto nel tempo, riducendo il rischio di regressioni, incoerenze o comportamenti non deterministici.

Il presente documento nasce con l'intento di offrire una **mappa completa e navigabile** dell'intero dominio software, organizzata in modo da permettere sia una lettura sequenziale sia una consultazione mirata. Ogni classe viene analizzata secondo criteri uniformi:

- responsabilità primaria e secondaria;
- interfacce pubbliche e contratti funzionali;
- dipendenze interne ed esterne;
- configurazioni dinamiche e parametri operativi;
- meccanismi di sicurezza, fallback e gestione degli errori;
- interazione con sensori, attuatori, protocolli e moduli di comunicazione;
- impatto sul ciclo di vita del sistema e sulle prestazioni complessive.

Questa struttura permette di cogliere non solo la funzione di ogni componente, ma anche la logica che guida l'intero ecosistema. In particolare, l'adozione di un modello **completamente parametrico**, privo di valori hardcoded e basato su configurazioni esterne, rappresenta uno dei pilastri fondamentali dell'architettura. Tale approccio consente di ottenere un sistema flessibile, adattabile a contesti diversi e facilmente estendibile senza compromettere la stabilità del codice esistente.

Un altro elemento cardine è la presenza di un **lettore sensori generico**, progettato per astrarre le differenze tra dispositivi fisici, protocolli e modalità di acquisizione. Questa scelta architettureale permette di integrare nuovi sensori o nuove tecnologie senza modificare il core del sistema, riducendo drasticamente il costo di manutenzione e aumentando la longevità della piattaforma. La stessa filosofia è applicata ai moduli di comunicazione, ai supervisor di potenza, ai gestori delle soglie dinamiche e ai motori di elaborazione dei dati ambientali.

Il documento include inoltre un'attenzione particolare agli aspetti di **robustezza, affidabilità e sicurezza**, elementi essenziali in un sistema che interagisce con il mondo fisico e che deve garantire continuità operativa anche in condizioni non ideali. Le classi dedicate alla diagnostica, al logging, al controllo dei buffer e alla gestione degli stati anomali sono analizzate con cura, evidenziando le strategie adottate per prevenire errori, degradazioni prestazionali o comportamenti imprevedibili.

Per quanto riguarda il **Capitolo 5**, dedicato alla trattazione approfondita dei profili di dispositivo e delle loro modalità di classificazione, si precisa che tale contenuto è stato già affrontato in un documento precedente. In questa sede, pertanto, esso verrà richiamato solo come riferimento, senza ripetere l'analisi già svolta. Tale scelta

permette di evitare ridondanze e di mantenere la documentazione snella, coerente e focalizzata sugli aspetti non ancora trattati.

L'intero documento è pensato per essere uno strumento di lavoro concreto, utile sia agli sviluppatori che operano quotidianamente sul codice, sia agli architetti che devono valutare l'evoluzione del sistema, sia agli integratori che necessitano di comprendere i meccanismi di configurazione e adattamento. La chiarezza espositiva, la coerenza terminologica e la struttura modulare sono state scelte per facilitare la consultazione e per rendere immediatamente accessibili le informazioni più rilevanti.

In conclusione, questa prefazione introduce un percorso di analisi completo e rigoroso, che accompagnerà il lettore attraverso ogni componente del sistema, fornendo una visione d'insieme e al tempo stesso un livello di dettaglio sufficiente per comprendere a fondo il funzionamento di ciascuna classe. L'obiettivo finale è costruire una documentazione che non sia solo descrittiva, ma che diventi parte integrante del processo di sviluppo, contribuendo alla qualità, alla stabilità e all'evoluzione futura di DomoManager.

Architettura Generale del Sistema

L'architettura di *DomoManager* nasce da un principio cardine: **un sistema domotico deve essere completamente parametrico, modulare, indipendente dall'hardware e deterministico nel comportamento**. Tutto il progetto è costruito attorno a questo concetto, e ogni componente — dalle classi di basso livello ai moduli di orchestrazione — riflette questa filosofia.

La struttura complessiva può essere vista come un insieme di **motori indipendenti**, coordinati da un **ciclo principale deterministico**, che condividono un'unica fonte di verità: **il Buffer**.

Visione ad Alto Livello

L'architettura è organizzata in **cinque macro-strati**, ciascuno con responsabilità chiare e indipendenti.

1. **Hardware Abstraction Layer (HAL)**: Racchiude tutti i componenti che interagiscono con il mondo fisico:

- driver digitali e analogici
- lettori sensori generici
- moduli I/O
- protocolli fisici (Modbus, RS485, CAN)
- timer, watchdog, periferiche Opta
- moduli esterni e dispositivi remoti
- Il HAL fornisce un'interfaccia uniforme e completamente astratta, permettendo al sistema di funzionare su hardware eterogeneo senza modifiche al codice.

2. **Core Engines** I motori funzionali: Sono i moduli che implementano la logica operativa del sistema:

- WeatherEngine
- PowerSupervisorEngine
- SecurityEngine
- HVACEngine
- MediaEngine
- FrontendEngine
- MQTTEngine
- AEE – Automation Execution Engine

Ogni motore:

- è indipendente dagli altri
- ha un ciclo di aggiornamento proprio
- legge e scrive esclusivamente nel Buffer
- non contiene valori hardcoded
- è completamente configurabile

3. **Buffer Centrale (Single Source of Truth)** Una struttura dati condivisa, completamente parametrica, che contiene:

- stati sensori

- stati attuatori
- variabili di automazione
- configurazioni dinamiche
- diagnostica
- comandi in ingresso
- telemetrie in uscita

Il Buffer garantisce:

- **coerenza dei dati**
- **assenza di dipendenze circolari**
- **debug semplificato**
- **scalabilità illimitata**
- **astrazione totale dall'hardware**

Tutte le classi interagiscono con il Buffer tramite funzioni di:

- lettura
- scrittura
- diagnostica
- validazione

4. **Configuration Layer** Tutti i parametri del sistema sono esterni:

- *FrontendConfig*
- *DeviceProfiles*
- *ZoneConfig*
- *ThresholdConfig*
- *PowerConfig*
- *WeatherConfig*
- *SecurityConfig*

Questo approccio elimina ogni valore hardcoded e permette:

- aggiornamenti senza ricompilare
- versioning delle configurazioni
- override dinamico
- adattamento a impianti diversi

5. **Interfacce Esterne**

- MQTT
- Dashboard
- App
- Sistemi terzi

- Moduli Modbus / RS485 / CAN / I/O digitali

Tutte le interfacce lavorano esclusivamente tramite il Buffer, garantendo isolamento e coerenza

Il Ciclo Principale Deterministico

Il cuore del sistema è un **main loop a pesi**, dove ogni motore viene eseguito con una frequenza definita:

- WeatherEngine → ogni 500 ms
- PowerSupervisor → ogni 200 ms
- SecurityEngine → ogni 50 ms
- HVACEngine → ogni 1 s
- MediaEngine → ogni 1 s
- FrontendEngine → ogni 100 ms
- MQTTEngine → ogni 100 ms
- AEE → ogni 20 ms

Questo modello garantisce:

- assenza di concorrenza
- assenza di race condition
- tempi di risposta prevedibili
- stabilità anche sotto carico elevato

Ogni motore:

- legge dal Buffer
- elabora la propria logica
- scrive nel Buffer

Nessun motore comunica direttamente con un altro.

Il Ruolo Centrale del Buffer

Il Buffer è il **fulcro dell'intera architettura**. Tutto ruota attorno a esso.

Il Buffer garantisce:

- **coerenza dei dati**
- **assenza di dipendenze circolari**
- **facilità di debug**
- **scalabilità illimitata**
- **astrazione totale dall'hardware**

È completamente definito tramite configurazione esterna

Engines: Architettura a Motori Indipendenti

Ogni motore è una classe autonoma, con:

- **setup()**

- **update()**
- **diagnostics()**
- **applyConfig()**

Ogni motore:

- legge i dati dal Buffer
- applica logiche interne
- aggiorna stati e comandi
- scrive i risultati nel Buffer

Questo permette:

- sostituire un motore senza toccare gli altri
- aggiungere nuovi motori senza modificare il core
- testare ogni motore in isolamento
- mantenere un'architettura pulita e stabile

Configurazione Dinamica e Parametrica

Uno dei pilastri del progetto è l'eliminazione totale dei valori hardcoded.

Tutto è configurabile:

- sensori
- attuatori
- zone
- soglie
- logiche
- timer
- profili dispositivi
- mappe I/O
- parametri di sicurezza
- parametri energetici
- parametri climatici

Ogni configurazione è caricata all'avvio e può essere aggiornata dinamicamente.

DeviceProfiles e Astrazione dell'Hardware

Il sistema non conosce l'hardware. Conosce solo *profili*.

Un DeviceProfile definisce:

- tipo dispositivo
- ingressi
- uscite
- modalità operative

- parametri
- diagnostica
- comandi supportati

Questo permette:

- sostituire un dispositivo fisico senza toccare il codice
- aggiungere nuovi dispositivi semplicemente creando un nuovo profilo
- mantenere un sistema stabile e scalabile

AEE – Automation Execution Engine

L'AEE è il motore che esegue le automazioni.

Caratteristiche:

- completamente parametrico
- basato su condizioni, eventi, stati, timer
- nessuna logica hardcoded
- supporto per automazioni complesse
- integrazione con tutti i motori tramite Buffer

L'AEE è il cuore logico del sistema.

FrontendEngine — Orchestrazione del Frontend e Integrazione con DomoManager

Il **FrontendEngine** è il motore centrale del livello frontend di DomoManager. È responsabile di:

- inizializzare i moduli funzionali (Weather, Power, HVAC, Security, Watch, Averages)
- leggere i sensori (WiredSensors, GenericSensor, DMAEE)
- aggiornare lo stato visibile del sistema
- gestire i comandi provenienti da UI, pannelli o bridge
- sincronizzare il frontend con il backend industriale (DomoManager)
- orchestrare l'intero ciclo logico del sistema

A differenza di MQTTEngine — che è opzionale — **FrontendEngine è sempre attivo** ed è parte integrante dell'architettura.

Il Ruolo del FrontendEngine nel Sistema

FrontendEngine è il **punto di convergenza** tra:

- **FrontendConfig** (configurazione esterna)
- **Moduli funzionali** (Weather, Power, HVAC, Security, ecc.)
- **DomoManager** (backend industriale)
- **Buffer Centrale** (single source of truth)
- **UI / Dashboard / Bridge / API locali**

FrontendEngine non controlla direttamente l'hardware: scrive nel Buffer → DomoManager applica i comandi.

Pipeline Operativa del Frontend

La pipeline completa è:

Codice

FrontendConfig



DMFrontendEngine::Config



DMFrontendEngine::Setup()



Moduli Funzionali (Weather, Power, HVAC, Security, Watch, Averages)



Buffer Centrale



DomoManager (backend industriale)



Modbus / RS485 / I/O fisici

Il FrontendEngine è quindi il **ponte logico** tra configurazione, logica funzionale e backend.

Analisi del Setup Reale (setup())

Il setup allegato mostra chiaramente la pipeline di inizializzazione.

Inizializzazione hardware Opta

```
Serial.begin(19200);

pinMode(I1, INPUT);
pinMode(I2, INPUT);
pinMode(I3, INPUT);
pinMode(I4, INPUT);
```

- inizializzazione seriale
- configurazione ingressi digitali Opta
- nessuna logica applicativa ancora attiva

Costruzione del FrontendConfig

Il blocco:

```
static FrontendConfig mainConfig = [](){
    FrontendConfig c;
    ...
}
```

```
return c;  
}();
```

costruisce **tutta la configurazione del sistema**:

- rete
- bridge UDP
- Modbus TCP → RTU
- HVAC
- Weather
- Power Supervisor
- Security
- Averages
- Diagnostics
- Watch
- Web API
- Jobs
- DomoManagerConfig (aree, dispositivi, automazioni, watchdog, ecc.)

Questa è la **sorgente unica di verità** per tutto il frontend.

Setup del FrontendEngine

```
DomoManagerFrontendEngine::Setup(mainConfig);
```

Questa chiamata:

1. crea il FrontendEngine
2. crea il DomoManager
3. inizializza il Buffer
4. costruisce tutti i moduli funzionali
5. applica la configurazione
6. collega FrontendEngine ↔ DomoManager tramite DMAEE
7. avvia i motori interni (Weather, Power, HVAC, Security, ecc.)

È il punto in cui il sistema diventa operativo.

Analisi del Main Loop Reale (loop())

Il main loop è estremamente semplice:

```
void loop() {  
    DomoManagerFrontendEngine::loop();  
}
```

Questa chiamata esegue:

FrontendEngine.update(now)

- lettura sensori

- aggiornamento moduli funzionali
- aggiornamento diagnostica
- aggiornamento medie
- aggiornamento automazioni (AEE)
- aggiornamento stato visibile

DomoManager.loop(now)

- polling Modbus
- polling RS485
- aggiornamento buffer industriale
- applicazione comandi agli attuatori
- gestione watchdog
- gestione errori dispositivi
- gestione automazioni runtime

Nessuna concorrenza

Il loop è deterministico:

- nessun thread
- nessun interrupt complesso
- nessuna race condition

FrontendEngine come Orchestratore dei Moduli Funzionali

Durante update(now) FrontendEngine esegue:

1. **WiredSensorsManager**
2. **GenericSensorReader**
3. **WeatherEngine**
4. **PowerEngine**
5. **PowerSupervisorEngine**
6. **HVACEngine**
7. **SecuritySensorEngine**
8. **WatchEngine**
9. **AveragesEngine**
10. **AutomationEngine (AEE)**
11. **DiagnosticEngine**
12. **BridgeEngine / WebAPI (se abilitati)**

Ogni modulo:

- legge dal Buffer
- elabora
- scrive nel Buffer

Non esistono dipendenze dirette tra moduli.

FrontendEngine come Ponte verso la UI

FrontendEngine prepara lo stato visibile:

- temperature
- potenze
- stati HVAC
- allarmi
- diagnostica
- variabili AEE
- comandi disponibili

E li rende disponibili a:

- dashboard web
- pannelli remoti (via BridgeEngine)
- API locali
- sistemi esterni (se abilitati)

FrontendEngine senza MQTT

Nel setup allegato:

```
c.mqtt.enabled = false;
```

MQTTEngine è completamente disattivato.

Il sistema continua a funzionare al 100%:

- FrontendEngine gestisce UI, pannelli, bridge
- DomoManager gestisce I/O fisico e automazioni
- nessuna dipendenza da MQTT
- nessuna perdita di funzionalità core

MQTT è un modulo opzionale.

Sintesi del Ruolo del FrontendEngine

FrontendEngine è:

- **il coordinatore del frontend**
- **il punto di convergenza tra configurazione, moduli funzionali e backend**
- **il motore che rende visibile lo stato del sistema**
- **il gestore dei comandi utente**
- **il ponte tra logica e hardware tramite DomoManager**
- **il cuore dell'orchestrazione di alto livello**

È uno dei moduli più importanti dell'intera architettura.

Sicurezza, Robustezza e Fail-Safe

L'architettura include:

- watchdog hardware
- watchdog software
- validazione sensori
- fallback automatici
- diagnostica continua
- logging interno
- protezione da valori incoerenti
- protezione da cicli infiniti
- protezione da configurazioni errate

Ogni motore è progettato per fallire in modo sicuro.

Scalabilità e Estendibilità

L'architettura permette:

- aggiungere nuovi motori
- aggiungere nuovi sensori
- aggiungere nuovi attuatori
- aggiungere nuovi protocolli
- aggiungere nuove logiche
- aggiungere nuove zone
- aggiungere nuovi profili dispositivi

Senza modificare il core.

Sintesi dell'Architettura

L'architettura generale di *DomoManager* è:

- **modulare**
- **deterministica**
- **parametrica**
- **scalabile**
- **robusta**
- **completamente configurabile**
- **indipendente dall'hardware**

È un'architettura progettata per durare nel tempo, crescere, adattarsi e rimanere stabile anche in condizioni critiche.

Analisi delle Classi

Il presente capitolo fornisce un'analisi sistematica e approfondita di tutte le classi che compongono il sistema. L'obiettivo è descrivere in modo chiaro **ruolo**, **responsabilità**, **interazioni**, **dipendenze**, **punti di estensione**, **vincoli**, e **criticità potenziali**. La trattazione segue un approccio *top-down*, partendo dai moduli di orchestrazione fino ai componenti periferici e ai servizi di supporto.

Classi di Orchestrazione Principale

Classe FrontendConfig

FrontendConfig è **la sorgente unica di verità** per tutto il frontend e parte del backend. È il *cuore parametrico* del sistema: tutto ciò che il software fa dipende da ciò che è definito qui.

Non contiene logica. Non contiene codice. Non contiene callback. 🖐️ **Contiene solo dati, strutturati in modo rigoroso.**

È il punto di ingresso dell'intero workflow:

```
file cfg → FrontendConfig → DMFrontendEngine → Moduli Funzionali → DMAEE → AEE →  
UI / MQTT / Bridge
```

Ruolo Esteso

FrontendConfig definisce:

1. Sensori

- sensori cablati (WiredSensors)
- sensori generici (GenericSensor)
- sensori virtuali (buffer)
- sensori funzione (fnInt, fnFloat, fnBool)
- sensori AEE (FrontendToModule / ModuleToFrontend)

2. Zone

- zone HVAC
- zone sicurezza
- zone meteo
- zone energetiche
- zone custom

3. Aree

- aree buffer
- aree virtuali
- aree diagnostiche
- aree di routing
- aree di automazione

4. Attuatori

- mapping aree → device
- mapping aree → Modbus
- mapping aree → pannello
- mapping aree → RS485

5. Soglie dinamiche

Per:

- WeatherEngine
- PowerEngine
- PowerSupervisorEngine
- HVACEngine
- SecuritySensorEngine
- Diagnostic

6. Profili di comportamento

- HVAC (isteresi, modalità, zone, priorità)
- Sicurezza (ritardi, zone, tamper, H24)
- Power (limiti, priorità carichi, fasce orarie)
- Weather (soglie vento/pioggia, debounce, scale)
- Watch (aree da monitorare, timing)

7. Mapping fisico → logico

- pin digitali
- pin analogici
- buffer area
- device Modbus
- canali RS485
- ingressi virtuali

8. Configurazione AEE

- variabili
- direzioni
- scaling
- minDelta
- mapping buffer → AEE

9. Configurazione Automazioni

- regole
- scene
- sequenze
- condizioni
- trend
- timer

10. Configurazione diagnostica

- aree critiche
- soglie spike
- timing callback

Caratteristiche Fondamentali

1. Totalmente esterna al codice logico

FrontendConfig è caricata da file:

- JSON
- binario
- EEPROM
- remoto

Il software non contiene valori fissi.

2. Permette la sostituzione dell'hardware

Cambiare:

- sensori
- device Modbus
- ingressi digitali
- pin
- zone
- attuatori
- non richiede modifiche al codice.

3. Garantisce assenza totale di hardcoded values

Tutto è dinamico:

- soglie
- scale
- mapping
- zone
- sensori
- automazioni
- profili

4. È la base per l'orchestrazione

FrontendConfig alimenta:

- **DMFrontendEngine**
- **WeatherEngine**
- **PowerEngine**
- **PowerSupervisorEngine**
- **HVACEngine**
- **SecuritySensorEngine**
- **DMAEE**
- **AEEManagement**
- **GenericSensor**
- **WiredSensors**

Perché FrontendConfig è così importante?

Perché separa **completamente**:

- *cosa fa il sistema*
- *da come è configurato*

Questo permette:

- modularità totale
- sostituzione hardware
- aggiornamenti senza ricompilare
- versioning delle configurazioni
- override dinamico
- editing da UI
- editing da file esterno
- editing remoto

Struttura Interna (semplificata)

```
struct FrontendConfig {  
    Pins pins;  
    Net net;  
    Bridge bridge;  
    Modbus modbus;  
    MQTT mqtt;  
  
    Weather weather;  
    Power power;  
    PowerSupervisor powerSupervisor;  
    HVAC hvac;  
    Security security;  
    Watch watch;  
  
    Sensors sensors;  
    Medie medie;  
    DiagnosticConfig diagnostic;  
  
    AEE aee;  
    WebApi webApi;  
};
```

Ogni blocco è un modulo indipendente.

Relazione con le altre classi

FrontendConfig

↓ genera

DMFrontendEngine::Config

↓ guida

DMFrontendEngine

↓ comunica tramite buffer/AEE

DomoManager (backend)

FrontendConfig è quindi **il primo anello della catena di orchestrazione**.

Classe DMFrontendEngine

DMFrontendEngine è **il motore centrale del frontend**, il punto in cui convergono:

- configurazioni dinamiche

- sensori
- moduli funzionali
- automazioni
- diagnostica
- protocolli esterni

È il coordinatore dell'intero ciclo di alto livello.

Ruolo Esteso

DMFrontendEngine svolge quattro funzioni fondamentali:

1. Gestione del ciclo di aggiornamento dei moduli

Esegue in sequenza:

- WeatherEngine.update()
- PowerEngine.update()
- PowerSupervisorEngine.update()
- HVACEngine.update()
- SecuritySensorEngine.update()
- WatchEngine.update()
- DMAEE.update()
- AutomationEngine.update()
- diagnostica e medie

È il **main loop logico** del sistema.

2. Integrazione delle configurazioni provenienti da FrontendConfig

DMFrontendEngine riceve una configurazione composta:

- WeatherConfig
- PowerConfig
- PowerSupervisorConfig
- HVACConfig
- SecurityConfig
- WatchConfig
- AEEConfig
- SensorsConfig

Ogni modulo funzionale viene costruito **dinamicamente** in base a questi blocchi.

3. Smistamento dei dati verso i moduli specializzati

DMFrontendEngine:

- legge sensori (GenericSensor, WiredSensors, DMAEE)
- normalizza i valori

- li inoltra ai moduli di dominio:
 - HVACEngine
 - WeatherEngine
 - PowerEngine
 - SecuritySensorEngine
 - WatchEngine

Ogni modulo riceve solo ciò che gli compete.

4. Pubblicazione dello stato verso l'esterno

DMFrontendEngine aggiorna:

- AEEManagement
- MQTT
- WebAPI
- Bridge UDP
- pannello HMI (via AEEProtocol)

È il punto in cui lo stato del sistema diventa **visibile all'esterno**.

Punti Chiave Estesi

1. Nessun valore hardcoded

Tutte le soglie, scale, mapping, zone, sensori e profili provengono da:

- FrontendConfig

DMFrontendEngine non contiene logica fissa.

2. Forte dipendenza da FrontendConfig

È progettato per essere:

- modulare
- sostituibile
- configurabile
- indipendente dall'hardware

Cambiare un sensore, un pin, un device Modbus o una zona **non richiede modifiche al codice**.

3. Interazione diretta con i moduli di dominio

Ogni modulo è aggiornato tramite interfacce statiche e metodi deterministici:

- update(now)
- getState()
- apply()

Non esistono callback nascosti o dipendenze circolari.

4. Punto di convergenza tra Frontend e Backend

DMFrontendEngine:

- legge dal buffer (via DMAEE)
- aggiorna le variabili AEE
- invia comandi verso DomoManager
- riceve eventi da WiredSensors

È il **ponte logico** tra:

- frontend (logica)
- backend (runtime industriale)

Configurazione Reale

```
DMFrontendEngine::Config dmCfg {
    .weather = weatherCfg,
    .power   = powerCfg,
    .hvac    = hvacCfg,
    .security = securityCfg,
    .watch   = watchCfg,
    .aee     = aeeCfg,
    .sensors = sensorsCfg
};
```

Ogni blocco deriva da FrontendConfig.

Utilizzo Reale

Istanziamento

```
DMFrontendEngine dm(dmCfg);
```

Loop operativo

```
void frontendLoop(unsigned long now) {
    dm.update(now);
}
```

Il metodo corretto è **update()**.

Relazione con le altre classi

FrontendConfig

↓ genera

DMFrontendEngine::Config

↓ guida

DMFrontendEngine

↓ comunica tramite buffer/AEE

DomoManager (backend)

DMFrontendEngine è quindi **la seconda colonna dell'orchestrazione principale**, insieme a:

- FrontendConfig

- DomoManager

Classe DomoManager

Ruolo

DomoManager è il **motore centrale del backend**, responsabile di tutta la parte “fisica” del sistema:

- polling Modbus TCP dei device
- aggiornamento del buffer
- gestione errori IP / device
- gestione toggles, splits, thresholds
- gestione automazioni runtime
- gestione routing Modbus
- gestione RS485
- gestione watchdog e diagnostica
- sincronizzazione con il frontend tramite DMAEE / AEE

È il **PLC virtuale** del sistema.

In sintesi:

👉 DMFrontendEngine = logica di alto livello 👉 DomoManager = runtime industriale + I/O + buffer + automazioni

Responsabilità principali

- **Gestione del ciclo backend**
 - lettura device
 - scrittura device
 - aggiornamento buffer
 - timing e spike detection
- **Gestione del buffer**
 - change-tracking
 - aree virtuali
 - reverse
 - ToPanel / FromPanel
- **Gestione automazioni**
 - AutomationEngine
 - AutomationBuilder
 - Scheduler
- **Gestione Modbus**

- client (device)
- server (pannello HMI)
- routing dinamico
- **Gestione sensori indiretti**
 - GenericSensor (via buffer)
 - DMAEE (buffer → AEE)
- **Supervisione**
 - Watchdog
 - DiagnosticWatchAreas
 - DeviceManager error tracking
 - IpManager error tracking

Configurazione reale

DomoManager riceve una configurazione **composita**:

```
DomoManagerConfig cfg;

cfg.devices = deviceCfg;
cfg.routes = routeCfg;
cfg.areas = bufferAreasCfg;
cfg.toggles = toggleCfg;
cfg.splits = splitCfg;

cfg.automation.json = automationJson;
cfg.watchdog = watchdogCfg;
```

Questa configurazione deriva **indirettamente** da FrontendConfig.

Utilizzo reale

```
DomoManager dm(AREA_COUNT, ledPins);

dm.setup(
    somethingChangedCallback,
    activityLoopCallback,
    cfg
);

dm.enableWatchdog();
```

Nel loop principale:

```
dm.loop(client, modbusTCPClient, now);
```

Punti chiave

- **Nessun valore hardcoded:** tutto deriva da DomoManagerConfig.
- **Nessuna istanza statica:** tutto è dinamico.
- **Nessuna dipendenza circolare:** DMFrontendEngine e DomoManager comunicano tramite buffer + AEE.
- **Backend e frontend sono separati:**
 - DMFrontendEngine = logica
 - DomoManager = runtime

Relazione con DMFrontendEngine e FrontendConfig

FrontendConfig

↓ genera

DMFrontendEngine::Config

↓ guida

DMFrontendEngine

↓ comunica tramite buffer/AEE

DomoManager (backend)

DomoManager è quindi **la terza colonna dell'orchestrazione principale**, insieme a:

- **DMFrontendEngine** (frontend orchestrator)
- **FrontendConfig** (configurazione)
- **DomoManager** (backend orchestrator)

Classi di Dominio Funzionale

Classe WeatherEngine

Responsabile dell'elaborazione dei dati meteo provenienti dalla stazione interna ed esterna.

Funzioni principali:

- Lettura sensori tramite GenericSensorReader.
- Calcolo di medie, trend, variazioni rapide.
- Validazione sensori con logging diagnostico.
- Applicazione soglie dinamiche (umidità, temperatura, vento).
- Pubblicazione dei dati verso i moduli che ne dipendono (HVAC, sicurezza finestre).

Punti critici:

- Necessità di gestione robusta dei sensori non disponibili.
- Dipendenza forte dalla qualità del reader generico.

Per approfondire: WeatherEngine

Classe PowerSupervisorEngine

Supervisiona consumi, produzione, batterie, priorità di carico.

Responsabilità:

- Lettura dinamica dei canali di potenza.
- Applicazione dei profili energetici configurati.
- Gestione soglie di sovraccarico e shedding automatico.
- Integrazione con sistemi esterni (fotovoltaico, UPS, generatori).

Caratteristiche:

- Completamente parametrico.
- Supporta scenari multi-sorgente.
- Fornisce eventi verso il sistema di sicurezza e automazione.

Per approfondire: PowerSupervisorEngine

Classe PowerEngine

Il PowerEngine gestisce il bilanciamento energetico dell'abitazione, il controllo dei carichi, la previsione solare e l'ottimizzazione dinamica dei parametri.

Obiettivi del Modulo

Il PowerEngine gestisce:

- carichi elettrici con priorità
- limiti energetici soft/hard
- previsioni solari
- carico termico
- auto-tuning dei parametri

- suggerimenti energetici

Architettura

Il modulo è composto da:

- **PowerManager**: motore principale
- **LoadManager**: gestione carichi
- **ThermalLoad**: gestione carico termico
- **ForecastEngine**: previsione solare
- **AutoTuneEngine**: ottimizzazione dinamica

Carichi Dinamici

Ogni carico è definito tramite:

- nome
- priorità
- potenza nominale
- minOn
- minOff

Il sistema garantisce che i carichi rispettino i tempi minimi e non superino i limiti energetici.

Thermal Load

Il modulo supporta un carico termico con:

- setpoint
- min
- max
- minOn
- minOff

Il controllo termico è integrato con la previsione solare e la temperatura media.

Auto-Tune

Il sistema può ottimizzare automaticamente i parametri energetici in base ai dati storici.

Forecast Solare

La previsione solare è basata su:

- mese
- ora
- luminosità
- temperatura esterna

Eventi e Callback

Il modulo genera eventi per:

- load change
- limit warning

- limit exceeded
- error
- suggestion

Classe DMHVAC

DMHVAC è il motore funzionale HVAC di alto livello, responsabile della gestione:

- riscaldamento
- raffrescamento
- fan-coil
- compressore
- valvola a 3 vie
- ACS (acqua calda sanitaria)
- anti-legionella
- defrost
- gestione finestre aperte
- logiche multi-zona

Internamente utilizza HeatPumpController, un motore HVAC completo e deterministico progettato per Opta e sistemi embedded.

Ruolo

DMHVAC fornisce:

- gestione completa della pompa di calore
- controllo multi-zona con isteresi
- gestione fan-coil per zona
- gestione ACS con priorità
- gestione anti-legionella programmata
- gestione defrost automatica
- gestione valvola a 3 vie
- gestione finestra aperta
- protezioni termiche e temporali
- callback verso attuatori fisici (compressore, fan-coil, valvole, pompe)

È uno dei moduli funzionali più complessi del sistema.

Componenti principali

1. HeatPumpController (motore HVAC puro)

Modalità

- OFF
- MANUAL
- HEATING
- COOLING
- AUTO
- DEFROST

Fan speed

- LOW
- MEDIUM

- HIGH

HydraulicTarget

- SPACE_HEATING
- DHW (ACS)

Config

Include parametri critici:

- isteresi
- isteresi ventola
- soglie differenziali
- defrost threshold
- durata defrost
- min/max outdoor temp
- min/max tempi ON/OFF
- timeout finestra aperta
- post-circulation
- min circulation cycle

2. ZoneHVAC (zona termica)

Ogni zona ha:

- nome
- setpoint
- temperatura attuale
- fan-coil associato
- richiesta caldo/freddo
- isteresi indipendente

Logica zona

```
heatRequest = (temperature < setpoint - hysteresis);
coolRequest = (temperature > setpoint + hysteresis);
```

3. ACS (Acqua Calda Sanitaria)

Funzionalità:

- priorità ACS
- setpoint ACS
- lettura temperatura ACS
- valvola a 3 vie
- anti-legionella programmata
- durata massima ciclo anti-legionella

4. Callback attuatori

DMHVAC non controlla direttamente gli attuatori: 🖐️ usa callback verso DMEquipment (Valve, Pump, Heater, ecc.)

Callback disponibili:

- compressorCallback(bool)
- fanCoilCallback(String zone, int id, bool on)

- circulationCallback(bool)
- threeWayValveCallback(HydraulicTarget)

Configurazione

La configurazione avviene tramite DMHVAC::HVACConfig:

```
DMHVAC::HVACConfig cfg;

cfg.enabled = true;
cfg.intervalMs = 5000;

cfg.zones = zoneArray;
cfg.zoneCount = 3;

cfg.readIndoorTemp = [](){ return buffer.getValueFast(AREA_TEMP_INT); };
cfg.readOutdoorTemp = [](){ return buffer.getValueFast(AREA_TEMP_EXT); };
cfg.readWindowOpen = [](){ return buffer.getValueFast(AREA_FINES); };

cfg.initialMode = HeatPumpController::Mode::AUTO;
cfg.initialSetpoint = 21.0f;
cfg.hpConfig.hysteresis = 0.5f;

// ACS
cfg.acs.enabled = true;
cfg.acs.threeWayValve = &valveACS;
cfg.acs.dhwSetpoint = 50.0f;
cfg.acs.readDhwTemp = [](){ return buffer.getValueFast(AREA_ACS_TEMP); };
```

Utilizzo

Setup

```
DMHVAC hvac;
hvac.setup(cfg);
```

Loop operativo

```
hvac.loop(
    now,
    hvacTime,    // struct HVACTime
    zoneTemps,   // array temperature zone
    indoorTemp,
    outdoorTemp,
```

```
windowOpen
```

```
);
```

Funzionalità principali del motore HVAC

1. Controllo riscaldamento

- attivazione compressore
- gestione isteresi
- gestione fan-coil
- gestione circolazione
- protezioni temporali

2. Controllo raffrescamento

- logica simile al riscaldamento
- gestione fan-coil
- protezioni

3. Defrost

- attivazione automatica
- durata configurabile
- blocco compressore durante defrost

4. ACS

- priorità ACS
- valvola 3 vie
- gestione setpoint ACS
- gestione temperatura ACS
- anti-legionella programmata

5. Finestra aperta

- timeout configurabile
- spegnimento compressore
- spegnimento fan-coil

6. Protezioni

- min/max outdoor temp
- min/max tempi ON/OFF
- timeout circolazione
- timeout defrost
- sicurezza ACS

Punti chiave

- Motore HVAC completo e deterministico
- Nessuna allocazione dinamica ricorrente
- Pensato per Opta e sistemi embedded
- Multi-zona con isteresi indipendente
- ACS integrata con valvola 3 vie
- Anti-legionella programmata
- Defrost automatico
- Callback verso attuatori fisici

- Integrato con DMFrontendEngine e DomoManager

Relazione con altri moduli

- DMFrontendEngine → fornisce configurazione e richiama loop
- DomoManager → gestisce attuatori fisici tramite callback
- Buffer → fornisce temperature e stati
- DMEquipment → attuatori fisici (Valve, Pump, Heater)
- AutomationEngine → può attivare scene HVAC

Classe WatchModule

Modulo di sorveglianza e monitoraggio.

Funzioni:

- Supervisione heartbeat dei moduli.
- Rilevamento freeze o ritardi.
- Generazione eventi di recovery.
- Logging diagnostico.

Per approfondire: WatchModule

Classi di gestione Servizi di Sistema e Infrastruttura

Classe DMFncs

Il modulo *DMFncs* raccoglie una serie di componenti fondamentali per il funzionamento del sistema:

- gestione modalità utente (Owner/GUEST/Developer)
- gestione IP e priorità Modbus
- mapping area → device
- gestione tempo (RTC + fallback)
- gestione Modbus TCP client/server
- routing dinamico
- diagnostica intelligente

Queste classi non appartengono al dominio funzionale (HVAC, Weather, Power, Security) e non sono orchestrazione (DMFrontendEngine, DomoManager). 🙌 Sono **servizi infrastrutturali** utilizzati da tutto il sistema.

OwnerManager

(Gestione modalità utente: Owner, Guest, Developer)

Ruolo

OwnerManager gestisce il *livello di permessi* del sistema:

- OWNER → accesso completo
- GUEST → accesso temporaneo
- DEVELOPER → accesso esteso + log attivi

Funzionalità

- cambio modalità con callback
- scadenza automatica della modalità Guest
- richieste IOT di accesso temporaneo
- pausa e ripristino dei log
- controllo permessi (modifica impostazioni, automazioni, log)

Esempio

```
owner.setGuest(now + 60000);  
owner.update(now);
```

IpManager

(Gestione IP Modbus, errori, cooldown, auto-esclusione)

Ruolo

IpManager gestisce:

- lista IP dei device Modbus
- errori consecutivi
- cooldown automatico
- auto-esclusione dopo troppi errori
- priorità di polling
- mapping device → IP

Funzionalità

- ShouldQuery() → decide se interrogare un IP
- ReportError() → incrementa errori e attiva cooldown
- ReportSuccess() → riduce cooldown
- Exclude() → esclusione manuale
- BuildIps() → costruzione lista IP dai device
- GetDevicesByIP() → lookup veloce

Comportamento intelligente

- log ridotti (solo cambi stato)
- cooldown progressivo
- auto-esclusione dopo troppi errori

AreaDeviceMap

(Mapping area → device/channel/item)

Ruolo

Costruisce una mappa $O(1)$ che permette di sapere:

- quale device Modbus controlla una certa area
- quale canale
- quale item

Funzionalità

- buildOnce() → costruzione iniziale
- find(area) → lookup istantaneo

È fondamentale per:

- ModbusManager
- routing
- automazioni

TimeManager

(Gestione tempo: RTC hardware + fallback)

Ruolo

TimeManager fornisce:

- tempo affidabile (epoch)
- sincronizzazione da IOT
- fallback su millis()
- accesso a struct tm
- validazione RTC

Funzionalità

- updateFromEpoch()
- getEpoch()
- getDateTime()
- nowMs()

Comportamento

- se RTC è valido → usa RTC
- se RTC non è valido → usa epoch + millis()

ModbusManager

(Gestione completa Modbus TCP client + server)

Ruolo

ModbusManager è il **motore Modbus del backend**:

- connessione ai device Modbus TCP
- lettura registri
- scrittura registri
- gestione errori IP
- routing dinamico
- integrazione con Buffer
- integrazione con ToggleManager, SplitOutManager, ThresholdManager
- server Modbus per pannello HMI

Funzionalità principali

Client

- RunClient()
 - connessione sicura
 - lettura device
 - scrittura device

- gestione cambi buffer
- routing
- split
- toggle
- threshold

Server

- RunServer()
 - sincronizzazione pannello
 - scrittura ToPanel
 - lettura FromPanel

Connessione

- chiusura connessione precedente
- delay minimo per Opta/NINA
- gestione errori tramite IpManager

RouteManager

(Routing dinamico area → area)

Nota: RouteManager è incluso indirettamente in ModbusManager.

Ruolo

- applicare logiche di routing tra aree
- eseguire callback per aree specifiche
- supportare automazioni avanzate

Punti chiave del modulo DMFncs

- È **infrastruttura di sistema**, non dominio
- Nessun valore hardcoded
- Nessuna allocazione ricorrente
- Tutto deterministico
- Integrato con Buffer, Modbus, Automazioni, DomoManager
- Fondamentale per stabilità e affidabilità

Relazione con altri moduli

- **DomoManager** → usa ModbusManager, IpManager, AreaDeviceMap
- **DMFrontendEngine** → usa TimeManager, OwnerManager
- **AutomationEngine** → usa TimeManager
- **Buffer** → usato da ModbusManager

- **ToggleManager / SplitOutManager / ThresholdManager** → integrati in ModbusManager

Classi di Gestione Sensori e Sicurezza

SecurityEngine – Gestione Avanzata dei Sensori di Sicurezza

Il SecurityEngine è stato ampliato in modo significativo rispetto alla versione precedente. La nuova implementazione introduce un modello più robusto, modulare e completamente parametrico, in linea con l'architettura generale del sistema.

Obiettivi del Modulo

Il SecurityEngine gestisce:

- sensori cablati e virtuali
- aggregazione logica degli allarmi
- gestione zone e tipologie
- reset automatico dei sensori in errore
- comandi enable/engage dinamici
- integrazione con automazioni e buffer

Il modulo garantisce un comportamento deterministico, sicuro e privo di falsi positivi.

Architettura Generale

Il modulo è composto da tre componenti principali:

- **WiredSensorsManager** Gestisce la lettura dei sensori, il debouncing, i timer interni, la validazione e la generazione degli allarmi.
- **ZoneDispatcher** Sistema di callback che permette di reagire a eventi specifici per zona, tipo o combinazione zona+tipo.
- **SystemManager** Aggregatore centrale che produce un bitmask di stato del sistema di sicurezza, utilizzato dal buffer e dalle automazioni.

Startup Inhibit

Il sistema supporta un periodo di inibizione iniziale configurabile tramite startupInhibitMs. Durante questo intervallo:

- gli allarmi non vengono generati
- i sensori vengono stabilizzati
- eventuali transitori vengono ignorati

Reset Automatico dei Sensori

Se tutti i reader associati a un sensore restituiscono false, il sensore viene automaticamente resettato:

- memoria interna
- timer
- stato engaged
- stato enable
- stato di allarme

Questo evita che un sensore in errore mantenga uno stato incoerente.

Aggregazione Logica (SystemManager)

Il SystemManager produce un bitmask composto dai seguenti campi:

- ALLARME_INTRUSIONE
- ALLARME_INTRUSIONE_H24
- ALLARME_ALLAGAMENTO
- ALLARME_FUMO

- `FINESTRE_APERTE`
- `PORTE_APERTE`
- `ALLARME_TAMPER`

Ogni campo è gestito tramite `Cell<bool>` con change tracking. Quando uno qualsiasi dei campi cambia, il `SystemManager`:

- aggiorna il buffer
- genera diagnostica
- notifica eventuali automazioni

Comandi di Sicurezza

Il modulo supporta due modalità:

- **`ApplySecurityCommands(area)`** Applica i comandi provenienti dal buffer (enable/engage) per il singolo sensore.
- **`ForceSecurityCommands(engage)`** Forza l'ingaggio di tutti i sensori e aggiorna il buffer.

Integrazione con il Buffer

Il `SecurityEngine` aggiorna automaticamente l'area:

- `AREA_SECURITY_STATUS`

contenente il bitmask aggregato del sistema.

Diagnostica

Il modulo fornisce report completi su:

- stato sensori
- stato zone
- comandi applicati
- inconsistenze
- bitmask aggregato

Classe `DMWiredSensors`

Gestisce tutti i sensori cablati.

Caratteristiche:

- Accesso tramite function-based readers (mai buffer diretto).
- Mappatura dinamica sensore → area → zona.
- Supporto per sensori digitali, analogici, contatti, reed, PIR.
- Diagnostica integrata (open, short, tamper).

Per approfondire: `DMWiredSensors`

Classi di Supporto e Servizi

Classe DMSignal

Il modulo **DMSignal.hpp** fornisce un insieme di primitive fondamentali per la gestione dei segnali digitali e temporali in ambiente embedded. Queste primitive sono progettate per:

- sistemi deterministici
- logiche PLC-like
- Opta, PLC industriali e microcontrollori
- assenza totale di allocazioni dinamiche
- esecuzione non-bloccante
- massima efficienza e prevedibilità

Il modulo include:

- gestione toggle
- timer industriali (TON, TOF, TP)
- debounce lento e veloce
- rilevamento fronti (rising, falling, change)

Queste primitive costituiscono la base per la costruzione di logiche di automazione, filtri digitali, gestione ingressi, e implementazioni di sicurezza.

ToggleSignal — Gestione Toggle su Fronte di Salita

ToggleSignal implementa un meccanismo di toggle digitale estremamente leggero e deterministico.

Funzionamento

- rileva variazioni di stato
- effettua il toggle **solo sul fronte di salita**
- utilizza un singolo byte di memoria
- nessuna operazione bloccante

Caratteristiche

- change(statusIn, value) restituisce true solo quando avviene un fronte di salita
- value viene invertito tramite XOR (più veloce di un if)
- ideale per pulsanti, comandi manuali, ingressi digitali

TimerBase — Classe Base per Timer Industriali

TimerBase è la classe astratta da cui derivano TON, TOF e TP.

Caratteristiche principali

- supporto formati temporali: millisecondi, secondi, minuti, ore
- nessuna allocazione dinamica
- nessun blocco
- tempo trascorso calcolato tramite millis()
- comportamento deterministico

Metodi principali

- SetPreset(preset, fmt)
- Start() / Stop()
- ET() → elapsed time
- Q() → stato del timer

TON — Timer On-Delay

Il TON attiva l'uscita **dopo** che l'ingresso rimane attivo per un tempo minimo.

Comportamento

- se IN = true → avvia timer
- se ET ≥ preset → Q = true
- se IN = false → reset immediato

Utilizzo tipico

- ritardi di attivazione
- filtri temporali
- logiche di sicurezza

TOF — Timer Off-Delay

Il TOF mantiene l'uscita attiva **per un tempo definito dopo** che l'ingresso torna a zero.

Caratteristiche avanzate

- gestione del primo ciclo (inizializzazione)
- stato `_everTrue` per evitare attivazioni spurie
- comportamento identico ai TOF PLC industriali

Comportamento

- se IN = true → Q = true
- se IN = false → avvia temporizzazione
- Q = true finché ET < preset

TP — Pulse Timer

Il TP genera un impulso di durata fissa sul fronte di salita dell'ingresso.

Comportamento

- fronte di salita → Q = true per preset ms
- nessun retrigger finché l'impulso è attivo

Utilizzo tipico

- generazione impulsi
- comandi monostabili
- trigger per automazioni

Debounce — Debounce Lento con TON

Debounce implementa un filtro antirimbalo basato su TON.

Caratteristiche

- tempo configurabile
- comportamento deterministico
- ideale per pulsanti fisici e ingressi rumorosi

Funzionamento

- se il valore cambia → avvia TON
- se il valore rimane stabile per preset ms → aggiorna stato

FastDebounce — Debounce Veloce

Versione ottimizzata per ingressi ad alta frequenza.

Caratteristiche

- nessun timer dedicato
- utilizza solo confronti temporali
- latenza minima

Edge — Rilevamento Fronti

Edge fornisce tre funzioni:

- Rising(in) → fronte di salita
- Falling(in) → fronte di discesa
- Change(in) → qualsiasi variazione

Utilizzo tipico

- pulsanti
- encoder
- segnali digitali veloci
- trigger per automazioni

Punti Chiave del Modulo

- progettato per Opta e PLC industriali
- zero allocazioni dinamiche ricorrenti
- completamente non-bloccante
- comportamento deterministico
- timer PLC-like (TON, TOF, TP)
- debounce robusti
- rilevamento fronti affidabile
- ideale per logiche di automazione e sicurezza

Relazione con Altri Moduli

- **DMFrontendEngine** → utilizza DMSignal per logiche interne
- **DomoManager** → usa TON/TOF/TP per automazioni runtime
- **DMAEE** → può usare Edge e Debounce per variabili digitali
- **WiredSensors** → integra Debounce e Edge per ingressi fisici
- **AutomationEngine** → costruisce logiche basate su TON/TOF/TP

Classe ICommandTransport (DMAdapters)

Ruolo

La classe rappresenta il livello di astrazione dei trasporti di comando. Ogni adapter implementa l'interfaccia ICommandTransport e fornisce un canale di comunicazione non-bloccante verso sistemi esterni o moduli interni.

Gli adapter disponibili sono:

- **MqttAdapter**
- **BridgeAdapter**
- **HttpAdapter**
- **UdpAdapter**
- **RS485NodeAdapter**

Responsabilità

- Uniformare l'invio di comandi e richieste asincrone
- Gestire callback di risposta
- Fornire un meccanismo di connettività indipendente dal protocollo
- Integrare DMAEE con protocolli esterni (MQTT, HTTP, UDP, RS485)
- Garantire un loop non-bloccante per ogni trasporto

Caratteristiche principali

- Nessuna operazione bloccante
- Pattern request/response unificato
- Timeout integrati (HTTP, RS485)
- Supporto ACK/NACK tramite RS485NodeAdapter
- Compatibilità diretta con DMFrontendEngine e DeviceMessageEngine

Relazioni

CommandTransportAdapters è utilizzato da:

- DMFrontendEngine
- DMAEE
- BridgeEngine
- DeviceMessageEngine
- moduli esterni (HMI, gateway, app)

Classe OptaRTC

OptaRTC è il **motore temporale del sistema**, progettato per fornire:

- tempo affidabile (epoch)
- sincronizzazione NTP via Ethernet o WiFi
- callback periodici (secondo, minuto, ora, giorno, settimana, mese)
- scheduler CRON integrato
- diagnostica completa dello stato RTC
- compatibilità totale con Opta e sistemi embedded

È un componente fondamentale per:

- automazioni temporali
- HVAC (anti-legionella, fasce orarie)
- logging
- sincronizzazione pannello
- gestione eventi programmati

Ruolo

OptaRTC fornisce:

1. Tempo affidabile

- lettura epoch
- conversione in struct tm
- gestione fuso orario
- fallback automatico se RTC non valido

2. Sincronizzazione

- NTP via Ethernet
- NTP via WiFi
- aggiornamento via UDP (bridge o gateway)
- sincronizzazione manuale (setEpoch, setDateTime)

3. Callback periodici

- ogni secondo
- ogni minuto
- ogni ora
- ogni giorno
- ogni settimana
- ogni mese

4. Scheduler CRON integrato

Supporta espressioni cron standard:

```
min hour day month weekday
```

Esempi:

- "0 3 * * *" → ogni giorno alle 03:00
- "*/5 * * * *" → ogni 5 minuti
- "30 18 * * 1-5" → 18:30 dal lunedì al venerdì

5. Diagnostica

- ultima sincronizzazione
- RTC mai sincronizzato
- ultima data valida
- numero cron registrati

Funzionalità principali

1. Sincronizzazione RTC

Set epoch

```
rtc.setEpoch(1716220000);
```

Set data/ora

```
rtc.setDateTime(2026, 5, 20, 14, 30, 0);
```

Sync da UDP

```
rtc.updateFromUDP("1716220000");
```

Sync NTP

```
rtc.ntpSync("pool.ntp.org", 1);
```

Sync multiplo

```
rtc.syncNTP({"pool.ntp.org", "time.google.com"}, 1);
```

2. Lettura tempo

Epoch

```
uint32_t e = rtc.getEpoch();
```

Struct tm

```
struct tm t;  
rtc.getDateTime(t);
```

Stringhe formattate

```
rtc.getDateString();      // "2026-05-20"  
rtc.getTimeString();      // "14:30:05"  
rtc.getDateTimeString();  // "2026-05-20 14:30:05"
```

3. Callback periodici

```
rtc.onEverySecond([](){ ... });  
rtc.onEveryMinute([](){ ... });  
rtc.onEveryHour([](){ ... });  
rtc.onEveryDay([](){ ... });  
rtc.onEveryWeek([](){ ... });  
rtc.onEveryMonth([](){ ... });
```

4. CRON scheduler

Aggiunta cron

```
rtc.addCron("0 3 * * *", [](){  
    // eseguito ogni giorno alle 03:00  
});
```

Matching cron

Gestito automaticamente in update().

5. Loop operativo

```
rtc.update(nowMs);
```

Il loop:

- evita aggiornamenti troppo frequenti
- rileva cambio secondo/minuto/ora/giorno/settimana/mese

- esegue callback
- esegue cron
- aggiorna diagnostica

Punti chiave

- Compatibile Opta (Ethernet + WiFi)
- Zero allocazioni ricorrenti
- Deterministico
- CRON integrato
- Callback periodici
- Diagnostica completa
- Fallback automatico se RTC non valido
- Sincronizzazione robusta (NTP + UDP)

Relazione con altri moduli

- **AutomationEngine** → usa cron e callback temporali
- **DMHVAC** → usa tempo per ACS e anti-legionella
- **DMFrontendEngine** → usa tempo per logiche di alto livello
- **DomoManager** → usa tempo per diagnostica e cicli
- **Buffer** → timestamp scritte

Esempio completo

```
OptaRTC rtc;

rtc.beginEth();
rtc.syncNTP({"pool.ntp.org", "time.google.com"}, 1);

rtc.onEveryMinute([](){
    LOG_IF("RTC", "Tick minuto");
});

rtc.addCron("0 3 * * *", [](){
    LOG_IF("RTC", "Esecuzione anti-legionella");
});

void loop() {
    rtc.update(millis());
}
```

Classe DMPLC

Il modulo **DMPLC** fornisce l'intero strato di astrazione tra:

- **hardware Modbus TCP**
- **buffer logico del sistema**
- **routing dinamico**
- **toggle e forward**
- **sensori generici**
- **profili device**
- **gestione priorità**
- **device manager**

È il layer che permette al sistema di trattare dispositivi fisici come **entità logiche uniformi**, indipendenti da:

- marca
- modello
- numero di canali
- tipo di registri
- indirizzi Modbus

Ruolo del modulo DMPLC

DMPLC fornisce:

1. Astrazione dei dispositivi Modbus

- definizione canali (AI, AO, DI, DO)
- mapping area → device/channel/item
- gestione errori per device
- priorità di polling
- profili predefiniti per device comuni

2. Sensori generici (GenericSensor)

- lettura analogica
- lettura digitale
- lettura da buffer
- costanti
- scaling

3. ToggleManager

- gestione aree toggle
- forward dinamico
- mapping areaRead → aree forward
- integrazione con ModbusManager

4. RouteManager

- routing dinamico area → area
- esecuzione azioni condizionali
- protezione targetArea
- integrazione con buffer

5. DeviceManager

- gestione lista device
- assegnazione automatica aree
- assegnazione manuale aree
- diagnostica errori device
- calcolo max read size

6. DeviceProfiles

- profili built-in per device noti
- definizione canali e registri
- supporto per profili custom

7. GenericPrgDevice

- rappresentazione di un device Modbus
- lettura registri
- scrittura registri
- mapping area
- gestione errori tramite classe Errors

Componenti principali

1. GenericSensor

(Sensore generico multi-sorgente)

Permette di leggere un valore da:

- pin analogico
- pin digitale
- area buffer
- costante

Funzionalità

```
GenericSensor::read(cfg, buffer);  
GenericSensor::readRaw(cfg, buffer);
```

Usi tipici

- sensori esterni
- ingressi digitali

- sensori virtuali
- override da pannello

ToggleManager

(Gestione toggle e forward)

Gestisce:

- aree toggle
- forward dinamico
- mapping areaRead → aree forward
- integrazione con ModbusManager

Funzionalità

- add(areaRead, forwards)
- isForwardArea(area)
- getForwardValue(area)

RouteManager

(Routing dinamico area → area)

Permette di definire regole:

```
se area X = valore Y → scrivi su aree Z
```

Funzionalità

- load() → carica configurazione
- hasRoute(area)
- execute(srcArea, value, buffer)

Usi tipici

- logiche di sicurezza
- sincronizzazione aree
- mapping avanzati

GenericPrgDevice

(Rappresentazione di un device Modbus)

Ogni device contiene:

- IP
- device address
- canali (AI, AO, DI, DO)
- mapping aree
- priorità
- gestione errori

Funzionalità

- Read()

- Write()
- GetArea(channel, item)
- FindChannelByArea()
- IsInError()

DeviceManager

(Gestione lista device + assegnazione aree)

Funzionalità

- addDeviceAuto() → aree assegnate automaticamente
- addDeviceManual() → aree definite dall'utente
- getDevices()
- DeviceHasErrors()
- GetMaxReadSize()

Ruolo

- costruisce la lista dei device
- assegna aree in modo deterministico
- fornisce diagnostica

AreaDeviceMap

(Lookup area → device/channel/item)

Funzionalità

- buildOnce(devices)
- find(area, out)

Lookup O(1), fondamentale per ModbusManager.

DeviceProfiles

(Profili built-in per device comuni)

Contiene profili per:

- N4DIH32
- MA01_XACX0440
- MA01_AXCX4020
- MA01_AXCX0080
- LE_01MQ
- R421A08
- GENERIC_4DI_4DO
- CWT_SLTH_6W_S
- CWT_THCO_2K
- IR_210
- PTA8C04

- RESI_LED
- LSA_H4P40YBM
- PROBE01

Funzionalità

- get(name)
- add(name, channels)
- list()

Punti chiave del modulo DMPLC

- È il **layer PLC/I/O del sistema**
- Nessuna logica di dominio
- Nessuna orchestrazione
- Zero allocazioni ricorrenti
- Deterministico
- Integrato con ModbusManager e DomoManager
- Fondamentale per la gestione dei device Modbus

Relazione con altri moduli

- **DomoManager** → usa DeviceManager, ToggleManager, RouteManager
- **ModbusManager** → usa GenericPrgDevice, AreaDeviceMap, ToggleManager, RouteManager
- **DMFrontendEngine** → usa GenericSensor
- **AutomationEngine** → legge aree gestite da DMPLC
- **Buffer** → memorizza valori letti/scritti dai device

Classe DMLogger

Il modulo **DMLogger.hpp** fornisce l'infrastruttura di logging interna del sistema DomoManager. È progettato per ambienti embedded e industriali, con particolare attenzione a:

- **determinismo**
- **assenza di allocazioni dinamiche ricorrenti**
- **leggerezza e velocità**
- **visibilità immediata degli eventi critici**
- **diagnostica strutturata**

Il modulo è composto da tre elementi principali:

- **Logger** → motore di logging
- **LogBuffer** → buffer circolare per eventi critici
- **LogManager** → abilitazione/disabilitazione globale del logging

Inoltre include un set completo di **macro ad alte prestazioni** per semplificare l'uso del logger in tutto il sistema.

LogBuffer — Buffer Diagnostico degli Eventi

LogBuffer è un buffer circolare ottimizzato per registrare solo gli eventi più rilevanti:

- **ERROR**
- **WARN**

Gli eventi **INFO** e **DEBUG** vengono stampati ma non memorizzati, evitando sprechi di memoria.

Caratteristiche principali

- capacità massima: **10 eventi**
- deduplicazione automatica degli eventi identici
- conteggio delle occorrenze
- timestamp dell'ultima occorrenza
- sostituzione intelligente degli eventi meno rilevanti

Struttura di un evento

Ogni evento contiene:

- livello (ERROR/WARN)
- nome classe
- nome funzione
- messaggio
- timestamp
- numero di occorrenze

Funzionalità

- `add()` → aggiunge o aggiorna un evento
- `report()` → stampa un report ordinato per occorrenze
- `reset()` → svuota il buffer

Il report è pensato per:

- diagnostica in campo
- analisi post-evento
- debug di problemi intermittenti

LogManager — Abilitazione Globale del Logging

LogManager fornisce un controllo globale:

- `enable()`
- `disable()`
- `isEnabled()`

Questo permette di:

- disattivare completamente il logging in produzione
- attivarlo solo in fase di diagnostica
- ridurre il rumore nei sistemi ad alte prestazioni

Logger — Motore di Logging

Logger è il cuore del sistema di logging.

Funzionalità principali

- stampa su Serial con timestamp
- formattazione compatta e leggibile
- supporto a messaggi formattati (printf-style)
- integrazione con LogBuffer
- livelli: ERROR, WARN, INFO, DEBUG

Livelli di log

Livello	Uso
ERROR	errori critici, malfunzionamenti
WARN	condizioni anomale, fallback
INFO	stato operativo, eventi normali
DEBUG	diagnostica dettagliata

Il livello massimo compilato è definito da:

```
#define LOG_LEVEL LogLevel::INFO
```

Questo permette di:

- compilare firmware “silenziosi”
- abilitare DEBUG solo in fase di sviluppo

Macro di Logging — Alte Prestazioni

Il modulo fornisce macro che catturano automaticamente:

- nome classe
- nome funzione

Esempi:

```
LOG_I("HVAC", "Avvio motore HVAC");  
LOG_WF("Power", "Soglia superata: %d W", watt);  
LOG_E("MQTT", "Connessione persa");  
LOG_D("AEE", "Valore variabile aggiornato");
```

Categorie di macro

- **LOG_X** → messaggio semplice
- **LOG_XF** → messaggio formattato
- **LOG_XN** → senza registrazione nel buffer
- **LOG_XFN** → formattato senza buffer

Dove **X** ∈ {E, W, I, D}.

InfoBox — Messaggi ad Alta Visibilità

Per eventi importanti è disponibile:

```
LOG_IH("SYSTEM", "Avvio completato");
```

```
LOG_IFH("HVAC", "Setpoint aggiornato: %.1f °C", setpoint);
```

Genera un riquadro ASCII ad alta leggibilità, utile per:

- avvii di sistema
- cambi di stato importanti
- diagnostica in campo

Punti Chiave del Modulo

- progettato per Opta e PLC industriali
- zero allocazioni dinamiche ricorrenti
- completamente non-bloccante
- logging strutturato
- buffer diagnostico intelligente
- macro ad alte prestazioni
- supporto messaggi formattati
- timestamp integrato
- compatibile con tutti i moduli del sistema

Relazione con Altri Moduli

- **DMFrontendEngine** → log di configurazione e aggiornamenti
- **DomoManager** → log di automazioni, buffer, Modbus
- **WeatherEngine / PowerSupervisorEngine / HVACEngine** → diagnostica operativa
- **WatchModule** → log di freeze, ritardi, anomalie
- **DMAEE** → log di variabili e automazioni

Il logger è utilizzato in tutto il sistema come strumento di:

- diagnostica
- analisi
- validazione operativa
- troubleshooting

Classe DMRS485Node

Il modulo **DMRS485Node** fornisce due componenti avanzati progettati per sistemi Opta e PLC industriali:

- **RS485Node** → motore di comunicazione RS485 affidabile, con protocollo custom, ACK/NACK, retry e state machine non-bloccante
- **HotStandbyManager** → gestione completa di ridondanza attiva/passiva (master/slave) con failover e fallback automatici

Entrambi i componenti sono progettati per ambienti industriali caratterizzati da:

- comportamento **deterministico**
- assenza totale di operazioni **bloccanti**
- **zero allocazioni dinamiche ricorrenti**
- robustezza contro errori di linea

- supporto nativo a sistemi **ridondati** (dual Opta, PLC + Opta, ecc.)

RS485Node — Motore di comunicazione RS485

RS485Node implementa un protocollo RS485 leggero, robusto e ottimizzato per sistemi embedded. Il protocollo è basato su frame compatti e facilmente verificabili.

Struttura del frame

```
START    = 0x7E
LEN      = numero byte (TYPE + PAYLOAD)
TYPE     = tipo pacchetto
PAYLOAD  = dati variabili
CHKSUM   = XOR di LEN, TYPE e PAYLOAD
```

Questa struttura garantisce:

- parsing semplice
- robustezza contro errori
- overhead minimo
- compatibilità con microcontrollori a bassa memoria

Funzionalità principali

- invio pacchetti con ACK opzionale
- retry automatico fino a 3 tentativi
- timeout ACK configurato
- state machine RX completamente non-bloccante
- gestione automatica di ACK (0xFE) e NACK (0xFF)
- callback per pacchetti, errori e timeout
- memorizzazione dell'ultimo pacchetto per ritrasmissione

State Machine RX

La ricezione è implementata tramite una state machine deterministica:

- **WAIT_START**
- **WAIT_LEN**
- **WAIT_TYPE**
- **WAIT_PAYLOAD**
- **WAIT_CHKSUM**

Ogni stato elabora un singolo byte, garantendo:

- nessun blocco
- latenza minima
- robustezza contro disturbi di linea

Callback disponibili

- onPacket(type, payload, len)
- onError(code)
- onTimeout()

Utilizzo tipico

- comunicazione tra due Opta

- reti RS485 multi-nodo
- protocolli industriali custom
- sincronizzazione tra moduli remoti
- trasporto per sistemi ridondati

HotStandbyManager — Ridondanza attiva/passiva

HotStandbyManager implementa un sistema completo di **hot-standby** tra due nodi Opta o PLC, garantendo continuità operativa anche in caso di guasto del nodo principale.

Il modulo gestisce:

- ruolo **master attivo**
- ruolo **slave in standby**
- **failover automatico** se il master non risponde
- **failback automatico** quando il master ritorna operativo
- sincronizzazione **stato, dati di processo e timestamp**

Pacchetti supportati

- 0x01 → HEARTBEAT
- 0x02 → STATE_SYNC
- 0x03 → PROCESS_DATA
- 0x04 → TIMESTAMP_SYNC

Meccanismi principali

1. Heartbeat

- inviato ogni **500 ms**
- timeout a **1500 ms**
- se lo slave non riceve heartbeat → diventa master

Callback:

- onPeerAlive()
- onPeerLost()

2. Failover

- lo slave assume il ruolo di master se il master non risponde
- callback: onBecomeMaster()

3. Failback

- se il master torna operativo → lo slave ritorna slave
- callback: onBecomeSlave()

4. State Sync

- sincronizzazione variabili di stato ogni **300 ms**
- callback: onStateSync(data, len)

5. Process Data Sync

- replica dati di processo ogni **200 ms**
- callback: onProcessData(data, len)

6. Timestamp Sync

- sincronizzazione epoch ogni **1000 ms**

- integrazione diretta con **TimeManager**
- callback: onTimestampSync(epoch)

Funzionamento interno

RS485Node

- invio pacchetti tramite sendPacket()
- gestione ACK/NACK
- retry automatico
- state machine RX robusta
- nessuna operazione bloccante

HotStandbyManager

- utilizza RS485Node come trasporto
- implementa logica master/slave deterministica
- sincronizza stato e dati di processo
- aggiorna l'RTC dello slave tramite timestamp del master
- garantisce continuità operativa anche in caso di guasto

Esempi di utilizzo

RS485Node

```
RS485Node node(true); // master
node.begin(9600);

node.onPacket([](uint8_t type, uint8_t* data, uint8_t len){
// gestione pacchetto
});

node.sendPacket(0x10, payload, len, true);
node.poll();
```

HotStandbyManager

```
HotStandbyManager hsm(true, &timeManager);
hsm.begin();

hsm.onBecomeMaster([](){ LOG_IF("HSM", "Sono master"); });
hsm.onBecomeSlave([](){ LOG_IF("HSM", "Sono slave"); });

void loop() {
    hsm.poll();
}
```

Punti chiave del modulo

- progettato per Opta e PLC industriali

- zero allocazioni dinamiche ricorrenti
- completamente non-bloccante
- comportamento deterministico
- failover e failback automatici
- sincronizzazione stato, dati e tempo
- logging dettagliato
- protocollo RS485 robusto e leggero

Relazione con altri moduli

- **TimeManager** → sincronizzazione timestamp
- **DMFrontendEngine** → può utilizzare RS485Node per pannelli remoti
- **DomoManager** → può replicare stato/process data su nodo standby
- **Buffer** → dati di processo replicabili via HotStandby
- **AutomationEngine** → può essere sincronizzato tra nodi

Classe DMBridge

(Categoria: Supporto e Servizi)

DMBridge è il **sottosistema di comunicazione esterna** del framework. Fornisce un layer di trasporto e protocollo affidabile per sincronizzare le variabili AEE tra frontend, backend e sistemi remoti (HMI, pannelli, bridge, gateway, supervisor).

Non elabora logiche funzionali, non coordina moduli e non partecipa al ciclo operativo dei motori: 🖱️ è un **servizio infrastrutturale** che garantisce comunicazione robusta, asincrona e con conferma.

Ruolo

DMBridge fornisce:

- **Trasporto:** UDP o RS485 tramite interfaccia astratta IBridgeTransport.
- **Protocollo:** parsing JSON, gestione seq/ack, resend automatico.
- **Monitoraggio peer:** rilevamento ONLINE/OFFLINE.
- **Integrazione AEE:** invio e ricezione variabili tramite AEERegistry.
- **Affidabilità:** ritrasmissione pacchetti non confermati.

È il layer che collega il sistema a:

- pannelli HMI
- bridge esterni
- gateway di supervisione
- sistemi remoti
- UI personalizzate

Componenti Principali

1. IBridgeTransport

Interfaccia astratta per il trasporto:

- begin()
- available()
- read()

- send()

Implementazioni:

- **BridgeTransportUDP**
- **BridgeTransportRS485**

2. BridgeProtocol

Gestisce:

- parsing JSON
- integrazione con AEEProtocol::parseJSON()
- generazione ACK automatici

3. AEEProtocolEngine

Implementa:

- gestione sequenze (seqCounter)
- gestione ACK (lastAckSeq)
- resend automatico
- serializzazione variabili AEE

4. BridgePeerMonitor

Rileva:

- peer ONLINE
- peer OFFLINE
- timeout attività

5. BridgeEngine

Il “motore” del bridge:

- inizializzazione trasporto
- loop operativo
- gestione pacchetti
- resend
- monitoraggio peer

Configurazione

DMBridge non ha una configurazione complessa: richiede solo:

- un trasporto (IBridgeTransport)
- un registro AEE (AEERegistry)

Esempio:

```
BridgeTransportUDP udp;
AEERegistry reg;
```

```
BridgeEngine::Init(&udp, reg);
```

Utilizzo

Loop operativo

```
unsigned long now = millis();  
BridgeEngine::Loop(now);
```

Invio manuale

```
BridgeEngine::SendRaw("{\"cmd\":\"ping\"}");
```

Verifica stato peer

```
if (BridgeEngine::isPeerOnline()) {  
    // peer attivo  
}
```

Punti Chiave

- **Non appartiene al dominio funzionale** Non elabora meteo, HVAC, power, sicurezza.
- **Non appartiene all'orchestrazione principale** Non coordina moduli, non gestisce cicli logici.
- **È un servizio infrastrutturale** Fornisce comunicazione affidabile e asincrona.
- **Totalmente modulare** Può usare UDP, RS485 o altri trasporti futuri.
- **Integrato con AEE** È il canale principale per sincronizzare variabili verso l'esterno.

Classe AutomationEngine

AutomationEngine è il **motore di automazione generica** del sistema. Fornisce un framework completo per definire ed eseguire:

- regole condizionali
- scene
- sequenze temporizzate
- scheduled rules (orari, intervalli, timeout)
- automazioni dinamiche basate su callback

Non appartiene al dominio funzionale (HVAC, Weather, Power, Security) e non coordina moduli: 🖱️ è un **servizio trasversale** che opera sopra il buffer e il time manager.

Ruolo

- Valuta condizioni su aree del buffer
- Esegue azioni (scrittura aree)
- Attiva scene predefinite
- Gestisce sequenze temporizzate
- Esegue automazioni periodiche
- Supporta regole basate su orario (RTC)
- Integra medie, buffer e time manager tramite SchedulerContext

Componenti principali

- **Condition** → confronti su valori del buffer
- **Action** → scrittura di un valore in un'area
- **Rule** → insieme di condizioni + azioni THEN/ELSE
- **Scene** → insieme di azioni atomiche
- **ActionSequence** → sequenze temporizzate
- **ScheduledRule** → regole basate su orario o intervallo
- **DynamicAutomation** → callback periodici
- **SchedulerContext** → accesso condiviso a buffer, medie, tempo

Configurazione

AutomationEngine non ha una configurazione propria: viene inizializzato con i servizi condivisi:

```
automation.attachScheduler(&scheduler, &buffer, &averages, &timeManager);
```

Utilizzo

Aggiornamento ciclico

```
automation.update(now);
```

Aggiunta scena

```
auto* s = automation.addScene("LuciOn");  
s->addAction({ AREA_LUCE_SALOTTO, 1 });
```

Regola giornaliera

```
automation.addDailyRule(18, 30, s); // alle 18:30
```

Automazione dinamica

```
automation.addDynamicAutomation(1000, [](auto* ctx, unsigned long now){  
    // logica personalizzata  
});
```

Punti chiave

- Non contiene logica di dominio
- Non contiene valori hardcoded
- Non dipende da hardware
- Opera esclusivamente tramite buffer
- È un servizio generico e riutilizzabile
- È richiamato da DMFrontendEngine o DomoManager

Classe AutomationBuilder

AutomationBuilder è il **motore di costruzione dinamica delle automazioni**. Non esegue logiche, non valuta condizioni, non attiva scene: 🖐 **genera** le automazioni a partire da una configurazione esterna (JSON).

È il complemento naturale di **AutomationEngine**, ma con un ruolo completamente diverso:

- **AutomationEngine** → esegue automazioni

- **AutomationBuilder** → costruisce automazioni

Ruolo

AutomationBuilder:

- interpreta un file JSON di configurazione
- costruisce dinamicamente:
 - scene
 - regole dinamiche
 - sequenze temporizzate
 - composite rules
 - trend rules
 - threshold rules
 - bitmask rules
 - multi-condition rules
 - time-based rules
- registra tutto dentro **AutomationEngine**
- risolve nomi area → ID tramite AreaRegistry
- fornisce diagnostica completa della configurazione

È un **parser + factory** per automazioni.

Componenti principali

1. Parser JSON

parseJson() converte un file JSON in:

- SceneCfg
- RuleCfg
- SequenceCfg
- CompositeCfg
- TrendCfg
- DebounceCfg
- BitmaskInputCfg

Ogni elemento viene validato e convertito in strutture interne.

2. Scene Builder

Costruisce scene statiche:

cpp

```
auto* scene = engine.addScene("LuciOn");
```

```
scene->addAction({ AREA_LUCE, 1 });
```

3. Dynamic Rule Builder

Registra automazioni dinamiche:

- TrendRule
- ThresholdRule
- BitmaskRule
- MultiRule
- TimeRule
- SequenceRule
- CompositeRule

Ogni regola viene trasformata in una **DynamicAutomation** con callback.

4. Sequence Builder

Costruisce sequenze temporizzate:

```
seq->addStep({ area, value }, delayMs);
```

5. Composite Rule Engine

Supporta input complessi:

- debounce
- trend
- bitmask
- simple
- TOF (timer on fallback)

E produce un output bitmask + scena.

6. Diagnostica

ReportAutomationConfig() stampa:

- scene
- regole
- sequenze
- parametri
- warning su aree non valide

Configurazione

AutomationBuilder riceve una struttura:

```
AutomationBuilder::AutomationConfig cfg;
AutomationBuilder::parseJson(json, cfg);
```

Utilizzo

Costruzione completa

```
AutomationBuilder builder;

builder.build(
    engine,
```

```
buffer,  
medie,  
timeManager,  
cfg  
);
```

Risultato

- scene registrate
- regole dinamiche attive
- sequenze operative
- composite rules funzionanti

Punti chiave

- Non contiene logica di dominio
- Non esegue automazioni
- Non appartiene all'orchestrazione principale
- È un **servizio di costruzione**
- Permette automazioni completamente dinamiche
- Supporta configurazioni JSON complesse
- È indipendente dall'hardware

Relazione con altre classi

- **AutomationEngine** → esecuzione
- **DomoManager** → runtime backend
- **DMFrontendEngine** → orchestrazione frontend
- **Buffer** → sorgente dati
- **AverageCalculator** → trend
- **TimeManager** → regole temporali

AutomationBuilder è quindi un **modulo di supporto** che abilita automazioni dinamiche basate su configurazioni esterne.

Classe DMBaseClass

DMBaseClass è la **classe base astratta** da cui derivano tutti i moduli del sistema. Fornisce un set uniforme di funzionalità comuni, garantendo:

- identificazione univoca del modulo
- logging standardizzato
- ciclo di vita coerente
- diagnostica integrata
- struttura deterministica

Ruolo

- Base comune per tutti i motori (Weather, HVAC, Power, Security, ecc.)
- Fornisce nome modulo, logging, diagnostica
- Garantisce uniformità tra componenti
- Evita duplicazione di codice

Punti chiave

- Nessuna logica di dominio
- Nessun valore hardcoded
- Nessuna dipendenza da hardware
- È un *foundation layer* del framework

AsyncScheduler

AsyncScheduler è un **motore di workflow asincrono**, progettato per eseguire:

- sequenze di step
- branch condizionali
- skip dinamici
- delay temporali
- job con priorità

Ruolo

- Eseguire job complessi senza bloccare il loop
- Permettere automazioni avanzate (sequenze, condizioni, rami)
- Fornire un sistema di scheduling deterministico

Funzionalità

- Step normali e branch
- Condizioni dinamiche
- Skip condizionale
- Delay per step
- Priorità dei job
- Callback di completamento

Utilizzo

```
AsyncScheduler scheduler;
scheduler.setContext(&ctx);
scheduler.run(now);
```

Cell<T>

Cell è una **variabile reattiva** con change-tracking integrato.

Ruolo

- Memorizzare un valore
- Segnalare quando cambia

- Evitare aggiornamenti inutili

Funzionalità

- `setIfDiff()` → aggiorna solo se diverso
- `hasChanged()` → rileva cambiamenti
- `preserveGet()` → lettura senza reset

Group & AverageCalculator

Queste due classi lavorano insieme per fornire:

- medie mobili
- trend (INCREASING / DECREASING / CONSTANT)
- variazioni successive
- buffer circolari

Group

Gestisce:

- buffer circolare di misure
- buffer variazioni
- calcolo media
- calcolo trend

AverageCalculator

Gestisce:

- gruppi multipli
- creazione automatica
- trend per nome gruppo

Utilizzo

```
medie.addMeasurement("TempEsterna", valore);
auto trend = medie.groupTrend("TempEsterna");
```

AnalogThresholdManager

Gestisce soglie dinamiche per aree analogiche.

Ruolo

- Fornire soglie low/high
- Adattare soglia in base al valore
- Supportare fallback

SplitOutManager

Gestisce logiche di split:

- un'area principale
- più aree di uscita
- timer TON

- callback start/stop

Ruolo

- Attivare più aree da un singolo trigger
- Gestire timeout
- Evitare conflitti tra aree

Errors

Gestisce:

- conteggio errori
- finestre temporali
- reset automatico
- stato di errore persistente

Ruolo

- Stabilizzare moduli che possono fallire
- Evitare retry continui
- Fornire un meccanismo di auto-recovery

Watchdog

Watchdog monitora:

- activity loop
- update cycle
- spike
- blocchi
- overload

Ruolo

- Rilevare anomalie runtime
- Segnalare instabilità
- Attivare callback di sicurezza

LedController

Gestisce LED di sistema:

- READ
- WRITE
- PANEL
- ERR

Funzionalità

- on/off
- blink non-blocking
- pulse

- active-low

AreaRegistry

Gestisce:

- mapping nome → area
- risoluzione simbolica
- dump aree
- supporto per automazioni e AEE

Ruolo

- Permettere configurazioni simboliche
- Evitare hardcoded numerici
- Supportare JSON e FrontendConfig

Classe Buffer

La classe **Buffer** è il **cuore dati condiviso** del sistema. È il punto di scambio tra:

- **Backend (DomoManager)**
- **Frontend (DMFrontendEngine)**
- **DMAEE / AEEProtocol**
- **Automazioni**
- **Modbus**
- **Pannello HMI**
- **Sensori virtuali**
- **Motori funzionali**

È il componente che garantisce:

- consistenza
- atomicità
- change-tracking
- routing
- virtualizzazione
- diagnostica

In pratica: 🖐️ **tutto ciò che accade nel sistema passa dal Buffer.**

Ruolo

Buffer fornisce:

1. Memoria centrale delle aree

Ogni area rappresenta:

- un sensore
- un attuatore

- un valore logico
- un valore virtuale
- un valore proveniente dal pannello
- un valore destinato al pannello

2. Change-tracking avanzato

Per ogni area e tipo (Field, FromPanel, ToPanel) tiene traccia di:

- ultimo valore
- valore precedente
- timestamp
- flag di esistenza
- flag di lettura/scrittura

3. Routing dinamico

Ogni area può:

- leggere dal pannello
- scrivere al pannello
- essere invertita (Reverse)
- scrivere su un'altra area (areaToWrite)

4. Supporto per aree virtuali

Aree non collegate a hardware reale, usate per:

- automazioni
- logiche composite
- valori derivati

5. Diagnostica integrata

Buffer fornisce strumenti per:

- aree mai inizializzate
- aree inizializzate più volte
- aree virtuali
- aree inutilizzate

6. Iterazione efficiente sulle aree cambiate

Grazie a:

- `getChangedMap()`
- `getChangedMapByType()`
- iteratori filtrati

Struttura interna

Buffer gestisce:

BufferInfo

Contiene:

- Data[Field]
- Data[FromPanel]
- Data[ToPanel]
- flag Reverse
- flag ReadFromPanel
- flag WriteToPanel
- nome area
- flag virtuale
- flag wasRead / wasWritten

BufferSourceInfo

Per ogni tipo:

- valore
- valore precedente
- timestamp
- tipo

ChangedEntry

Mappa delle aree cambiate:

```
std::unordered_map<int, ChangedEntry> _changed;
```

AreaTracker

Traccia:

- aree mai inizializzate
- aree inizializzate più volte

Funzionalità principali

1. Inizializzazione

```
buffer.SetElement(area, areaToWrite, WriteToPanel, ReadFromPanel, Reverse, "NomeArea");  
buffer.AddType(area, initialValue, Field);
```

2. Scrittura

```
buffer.WriteElement(area, Field, value, now);
```

Supporta:

- modalità silenziosa
- change-tracking automatico
- aggiornamento timestamp

3. Lettura

Lettura veloce

```
long v = buffer.getValueFast(area);
```

Supporta:

- divideFactor per scalare valori
- flag wasRead

Lettura completa

```
BufferReadElementInfo info = buffer.ReadElement(area, preserve, Field);
```

4. Change-tracking

Tutte le aree cambiate

```
auto& map = buffer.getChangedMap();
```

Solo un tipo

```
for (auto& e : buffer.getChangedMapByType(ToPanel)) { ... }
```

Saltando aree virtuali

```
buffer.getChangedMapByType(Field, true);
```

5. Aree virtuali

```
buffer.SetVirtual(area, true);  
buffer.IsVirtual(area);
```

6. Diagnostica

```
Buffer::Diagnostics::ReportVirtualAreas(buffer);  
Buffer::Diagnostics::ReportNeverInitialized(buffer);  
Buffer::Diagnostics::ReportMultipleInitialized(buffer);
```

Esempio di utilizzo

Inizializzazione

```
Buffer buffer(256);  
  
buffer.SetElement(10, -1, false, false, false, "TempEsterna");  
buffer.AddType(10, 0, Field);
```

Scrittura

```
buffer.WriteElement(10, Field, 235, now);
```

Lettura

```
long t = buffer.getValueFast(10, 10); // scala di 10
```

Iterazione sulle aree cambiate

```
for (auto& e : buffer.getChangedMapByType(Field)) {  
    int area = e.second.area;  
    // gestisci cambiamento
```

```
}
```

Punti chiave

- È il **cuore dati del sistema**
- Nessun valore hardcoded
- Supporta routing dinamico
- Supporta aree virtuali
- Change-tracking avanzato
- Integrato con DMAEE, AEE, DMFrontendEngine, DomoManager
- Deterministico e ottimizzato per sistemi embedded

Relazione con altri moduli

- **DMAEE** → legge dal buffer e aggiorna AEE
- **AEEProtocol** → serializza cambiamenti del buffer
- **DMFrontendEngine** → legge sensori e scrive nel buffer
- **DomoManager** → aggiorna buffer da Modbus
- **AutomationEngine** → legge/scrive aree
- **SplitOutManager** → usa buffer per attivare aree
- **AverageCalculator** → legge valori dal buffer

Classe DMAEE

Automation Exchange Engine; Il modulo DMAEE (Automation Exchange Engine) fornisce l'intero framework di scambio dati tra:

- motori interni
- frontend
- bridge
- MQTT
- automazioni
- protocolli esterni

È un sistema parametrico, tipizzato, scalabile, progettato per:

- tracciare variazioni di variabili
- serializzare e deserializzare JSON
- applicare aggiornamenti provenienti dal Buffer
- gestire soglie, scaling, bitmask
- garantire comunicazione affidabile con ack e resend

DMAEE è composto da quattro blocchi principali:

- **AEEVarDef** – definizione variabile
- **AEEVariable** / **AEEVariableBase** – variabili tipizzate con change-tracking
- **AEERegistry** / **AEEManagement** – registrazione e lookup
- **DMAEE** – motore di aggiornamento da Buffer → AEE

AEEVarDef – Definizione Variabile

Una variabile AEE è definita tramite una struttura completamente parametrica:

- nome
- direzione (FrontendToModule, ModuleToFrontend, Bidirectional)
- sorgente (BufferArea, GenericSensor, Function)
- bitIndex opzionale
- scaling
- soglia minima di variazione (minDelta)

Questa definizione permette di:

- mappare una variabile AEE a un'area buffer
- leggere un sensore generico
- usare una funzione C++ come sorgente
- applicare scaling e filtri

Esempio:

```
{
    .name = "TempEsterna",
    .direction = AEEDirection::ModuleToFrontend,
    .sourceType = AEEVarSourceType::BufferArea,
    .bufferArea = AREA_TEMP_ESTERNA,
    .varType = AEEVarType::FLOAT,
    .scale = 0.1f,
    .minDelta = 0.2f
}
```

AEEVariableBase e AEEVariable<T>

Ogni variabile AEE deriva da AEEVariableBase e implementa:

- change-tracking
- serializzazione JSON
- parsing JSON
- callback onChange

La versione tipizzata AEEVariable<T> gestisce automaticamente:

- tipo (bool, int, float, string)
- soglia minDelta
- scaling
- timestamp dell'ultimo cambiamento

Caratteristiche:

- nessuna RTTI
- nessuna allocazione ricorrente
- aggiornamenti deterministici
- change-tracking ottimizzato

AEERegistry – Registro Variabili

Il registro contiene tutte le variabili AEE attive nel sistema.

Funzioni principali:

- add() – registra una variabile
- find() – lookup per nome
- forEach() – iterazione completa
- forEachChanged() – iterazione solo variabili modificate
- clearAllChanged() – reset globale

Il registro è utilizzato da:

- AEEProtocol (serializzazione)
- BridgeEngine
- MQTT AEE Bridge
- DMAEE (aggiornamenti da buffer)

AEEManagement – Costruzione Variabili

AEEManagement:

- riceve un array di AEEVarDef
- costruisce automaticamente le variabili tipizzate
- le registra nel registry

È il punto di ingresso per definire un set di variabili AEE.

Esempio:

```
static const AEEVarDef vars[] = {
    { "TempEsterna", AEEDirection::ModuleToFrontend, AEEVarSourceType::BufferArea,
    AREA_TEMP_ESTERNA },
    { "Allarme", AEEDirection::FrontendToModule, AEEVarSourceType::BufferArea,
    AREA_ALLARME }
};

AEEManagement mgr(vars, 2);
mgr.registerAll(registry);
```

AEEProtocol – Serializzazione e Parsing JSON

AEEProtocol gestisce:

- serializzazione delle variabili modificate
- parsing dei pacchetti JSON in ingresso
- gestione del campo seq
- protezione contro pacchetti troppo grandi
- validazione dei tipi

Funzionalità:

- invio solo variabili cambiate
- supporto direzioni (FrontendToModule, ModuleToFrontend, Bidirectional)
- JSON compatto e deterministico
- protezione overflow (max 900 bytes)

DMAEE – Motore di Aggiornamento

DMAEE è il cuore del sistema: aggiorna le variabili AEE in base ai cambiamenti del Buffer.

Pipeline:

1. BuildUpdatesFromBuffer()
 - controlla quali aree buffer sono cambiate
 - applica scaling
 - applica bitmask
 - legge sensori generici
 - invoca funzioni custom
2. ApplyUpdates()
 - aggiorna le variabili AEE
 - applica minDelta
 - attiva callback onChange
 - registra timestamp

Questo permette di sincronizzare:

- Buffer → AEE
- AEE → Bridge
- AEE → MQTT
- AEE → Automazioni

In modo deterministico e senza overhead.

Integrazione con BridgeEngine e MQTT

DMAEE è progettato per integrarsi con:

✓ BridgeEngine

- invio pacchetti JSON
- gestione ack
- resend automatico

✓ MQTT AEE Bridge

- pubblicazione variabili AEE come sensori Home Assistant
- mapping dinamico
- supporto bool, int, float

Vantaggi Architettureali

DMAEE introduce:

- zero hardcoded
- tipizzazione forte
- change-tracking efficiente
- scalabilità illimitata
- astrazione completa dal buffer
- protocollo JSON standardizzato
- integrazione con automazioni e frontend

È uno dei moduli chiave per rendere DomoManager:

- modulare
- estensibile

- interoperabile
- robusto

Classe BridgeEngine

Comunicazione Affidabile con Sistemi Esterni, protocollo custom di connessione tra controller Arduino (es configurazioni Hot Standby Master\Slave)

Obiettivo

Il BridgeEngine gestisce la comunicazione bidirezionale tramite:

- UDP
- RS485

AEEProtocolEngine

Gestisce la sincronizzazione affidabile delle variabili AEE tramite JSON con:

- sequence number
- ack
- resend automatico
- timeout
- rate limiting

Serializzazione JSON

Le variabili AEE vengono serializzate solo se changed. Il JSON include sempre il campo seq.

Ritrasmissione

Se un ack non arriva entro il timeout, il pacchetto viene ritrasmesso automaticamente.

Peer Monitor

Il sistema rileva automaticamente:

- peer online
- peer offline

Classe GenericSensorReader

Elemento chiave per l'astrazione hardware.

Caratteristiche:

- Unifica la lettura di qualsiasi sensore.
- Supporta scaling, offset, filtri.
- Gestisce errori e fallback.
- Permette l'aggiunta di nuovi sensori senza modifiche al codice.

Per approfondire: GenericSensorReader

Classe Diagnostic

Responsabile della diagnostica globale.

Funzioni:

- Raccolta errori da tutti i moduli.

- Generazione report statici.
- Logging strutturato.
- Supporto alla manutenzione.

Per approfondire: Diagnostic

Classi di Attuatori e Driver Fisici

DMEquipment

Il modulo *DMEquipment* fornisce una suite completa di driver deterministici per attuatori fisici:

- riscaldamento (PID)
- valvole motorizzate
- pompe
- tapparelle / serrande
- irrigazione multi-zona

Tutti i componenti sono progettati per:

- sistemi embedded Opta-class
- zero allocazioni dinamiche ricorrenti
- loop non-bloccanti
- sicurezza fisica
- comportamento deterministico

EquipmentBase

(Base comune per tutti gli attuatori)

Ruolo

- Fornisce una macchina a stati comune: Idle → Running → Error
- Gestisce timeout
- Gestisce timestamp dei comandi
- Fornisce reset errori
- È la base per Heater, Valve, Pump, Shutter

Funzionalità

- state() → stato corrente
- setTimeout() → timeout di sicurezza
- resetError() → reset errori
- timedOut() → verifica timeout
- markCommand() → registra timestamp comando

Heater

(Controllo termico con PID e protezioni)

Ruolo

- Controllo PID completo
- Soft-start
- Anti-windup
- Rilevamento sensore bloccato
- Limiti di temperatura
- Ramp adattiva in base alla differenza setpoint-ambiente

Funzionalità

- setTarget(temp) → avvia riscaldamento
- stop() → ferma
- update(currentTemp) → restituisce potenza (0..maxPower)

- setMaxTemp(), setMinTemp()
- setStuckDetection()
- enableAdaptiveSoftStart()

Sicurezze

- temperatura fuori range
- sensore bloccato
- timeout
- anti-windup

Valve

(Valvola motorizzata con finecorsa e timeout)

Ruolo

- Gestione apertura/chiusura
- Finecorsa aperto/chiuso
- Timeout movimento
- Stato completo del movimento
- Stati
- VIdle
- Opening
- Closing
- Opened
- Closed
- StoppedByUser
- VError

Funzionalità

- commandOpen()
- commandClose()
- commandStop()
- update(switchStatus)

Pump

(Pompa con protezione dry-run)

Ruolo

- Gestione ON/OFF
- Protezione livello liquido
- Timeout
- Safe shutdown

Funzionalità

- commandStart()
- stop()
- update(levelOk)
- isOn()

Shutter

(Tapparella / serranda con finecorsa e timeout)

Ruolo

- Movimento su/giù
- Finecorsa superiore/inferiore
- Timeout
- Stato completo del movimento

Stati

- Idle
- MovingUp
- MovingDown
- Opened
- Closed
- SError

Funzionalità

- commandUp()
- commandDown()
- stop()
- update(limitUp, limitDown)

IrrigationSystem

(Irrigazione multi-zona con sequenziatore non-bloccante)

Ruolo

- Gestione irrigazione a zone
- Sequenza automatica
- Callback per zone
- Callback per pompa
- Integrazione opzionale con elettrovalvola giardino
- Inibizione tramite sensore esterno
- Nessuna alloc dinamica ricorrente

Stati

3. Idle
4. OpeningGardenValve
5. WaitingValveOpened
6. PumpRunning
7. RunningZones
8. Finished
9. Error

Funzionalità

- configure(zones)
- setInhibit()
- setZoneCallback()
- setPumpCallback()
- attachGardenValve()
- start()
- stopAll()
- update()

Sequenza operativa

1. Apri elettrovalvola giardino
2. Attendi finecorsa
3. Avvia pompa
4. Avvia zona 1
5. Avanza zone sequenziali
6. Spegni pompa
7. Chiudi elettrovalvola

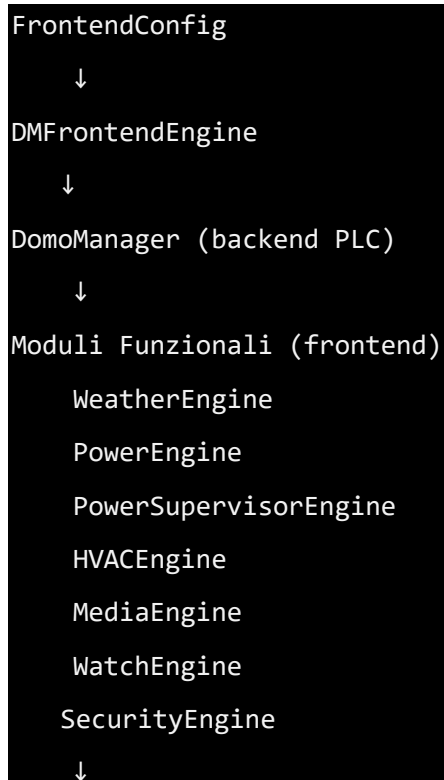
Punti chiave del modulo DMEquipment

- Zero allocazioni dinamiche ricorrenti
- Loop non-bloccanti
- Sicurezza fisica integrata
- Timeout per ogni attuatore
- Stati chiari e deterministici
- Pensato per Opta e sistemi embedded
- Integrabile con Buffer, AutomationEngine, DMFrontendEngine

Relazione con altri moduli

- DMFrontendEngine → invia comandi agli attuatori
- DomoManager → gestisce routing e scrittura fisica
- AutomationEngine → attiva scene che comandano attuatori
- Buffer → memorizza stati e comandi
- GenericSensor → fornisce input per Heater, Pump, Irrigation

Sintesi delle Dipendenze



Acquisizione Dati

DMWiredSensors

GenericSensorReader

↓

Comunicazione e Stato

AEEProtocol

AEEManagement

AEERegistry

DMAEE

↓

Supervisione

Diagnostic

Watchdog

IPManager

DeviceManager

↓

Integrazione Esterna

MQTT

Bridge

WebAPI

Modbus TCP

↓

Core Runtime (PLC)

DomoManager

Buffer

AutomationEngine

AutomationBuilder

ModbusManager

ToggleManager

SplitManager

ThresholdManager

RouteManager

DeviceManager

IpManager

OwnerManager

TimeManager (RTC)

RS485Node

Logger

Disclaimer Legale – Limitazione di Responsabilità

Il software **DomoManager**, la relativa documentazione tecnica e tutti i moduli associati (inclusi ma non limitati a: *DM.hpp*, *DMFrontend.hpp*, *DMSetup.hpp*, *DMDiagnostic.hpp*, automazioni, configurazioni e profili hardware) sono forniti **“così come sono”**, senza alcuna garanzia espressa o implicita.

L'autore e i collaboratori **non assumono alcuna responsabilità** per:

- danni diretti o indiretti
- malfunzionamenti
- perdita di dati
- interruzioni di servizio
- danni a cose, impianti o persone
- utilizzo improprio o non conforme del software
- modifiche non autorizzate al codice o alla configurazione
- integrazioni con dispositivi di terze parti non certificati

L'utente accetta che:

L'installazione, la configurazione e l'uso del software avvengono sotto la propria esclusiva responsabilità.

Il sistema è progettato per applicazioni domotiche e di automazione non critiche. Non deve essere utilizzato in contesti dove un malfunzionamento potrebbe causare:

- rischi per la sicurezza personale
- danni strutturali
- interruzioni di servizi essenziali
- condizioni pericolose per persone o animali

L'utente è tenuto a:

- verificare la correttezza delle configurazioni
- testare ogni automazione prima dell'uso reale
- assicurarsi che i dispositivi collegati siano conformi alle normative vigenti
- implementare adeguate misure di sicurezza elettrica e informatica

L'autore declina ogni responsabilità derivante da:

- modifiche al codice sorgente
- utilizzo in ambienti industriali o critici
- integrazioni con hardware non supportato
- errori di cablaggio o installazione
- uso improprio delle automazioni o delle scene

L'uso del software implica l'accettazione integrale del presente disclaimer.

DOMO Manager è ciò che davvero conta:

*la continuità, la sicurezza, la sostenibilità e la qualità dell'ambiente in cui viviamo ogni
giorno...*
